A Logic-Based Integration of Active and Deductive Databases

Alvaro A.A. Fernandes
Department of Mathematical and Computing Sciences
Goldsmiths College
University of London
New Cross, London SE14 6NW, UK
e-mail: a.fernandes@gold.ac.uk
phone: +44-171-919-7850, fax: +44-171-919-7853

M. Howard Williams
Department of Computing and Electrical Engineering
Heriot-Watt University
Riccarton, Edinburgh EH14 4AS, UK
e-mail: howard@cee.hw.ac.uk
phone: +44-131-451-3430, fax: +44-131-451-3431

Norman W. Paton
Department of Computer Science
University of Manchester
Oxford Road, Manchester M13 9PL, UK
e-mail: norm@cs.man.ac.uk
phone: +44-161-275-6910, fax: +44-161-275-6236

Abstract

A logic-based approach to the specification of active database functionality is presented which not only endows active databases with a well-defined and well-understood formal semantics, but also tightly integrates them with deductive databases. The problem of endowing deductive databases with rule-based active behaviour has been addressed in different ways. Typical approaches include accounting for active behaviour by extending the operational semantics of deductive databases, or, conversely, accounting for deductive capabilities by constraining the operational semantics of active databases. The main contribution of the paper is an alternative approach in which a class of active databases is defined whose operational semantics is naturally integrated with the operational semantics of deductive databases without either of them strictly subsuming the other. The approach is demonstrated via the formalization of the syntax and semantics of an active-rule language that can be smoothly incorporated into existing deductive databases, due to the fact that the standard formalization of deductive databases is reused, rather than altered or extended. One distinctive feature of the paper is its use of a *history*, as defined in the Kowalski-Sergot event-calculus, to define event occurrences, database states and actions on these. This has proved to be a suitable foundation for a comprehensive logical account of the concept set underpinning active databases. The paper thus contributes a logical perspective to the ongoing task of developing a formal theory of active databases.

1 Introduction

Rules in database systems can be of at least two kinds: *deductive rules* (DRs), which are used to express knowledge about application domains in a purely declarative way, and *active rules* (ARs), which are used to express actions to be performed, depending on context, as a reaction to events that may be internal or external to the system.

The availability of deductive capabilities significantly enhances query expressiveness and provides solid foundations [12] for both application development and the principled engineering of optimizers and evaluators for the query languages that applications require. The incorporation of active capabilities into database systems is justified on several grounds [50], of which the most important are: improving interaction between the system and its users, enforcing constraints on database states, and managing the database environment dynamically.

Research into *deductive database systems* (DDBs) started in the late 1970s and has matured into a number of prototype systems [44]. Initially, researchers were keener to exploit the intimate relationship between the relational model and first-order logic, but, more recently, several DDBs [4, 35] have been proposed whose underlying model of data and behaviour is object-oriented. The class of *active database systems* (ADBs) [2, 7, 14, 19, 22, 29, 30, 32, 38, 48] in which active behaviour is expressed as a set of *event-condition-action* (ECA) *rules* has recently attracted great interest. One appealing property of an ECA rule-set is that its operational semantics is independent of the underlying data model. As discussed in [41], ADBs based on ECA rules and displaying significant functional similarity have been built over both relational and object-oriented databases. The question arises whether this independence property also holds for ADBs over DDBs. The answer is not straightforward because DDBs already have an operational semantics of their own, and this, in turn, raises the issue as to whether an ECA rule-set and a DR-set can be given a unified operational semantics and what gains there are in doing so.

The different semantic accounts that can be given to DR-sets [18] are special cases of similar accounts given to logic programs in general. Simplifications are mainly due to additional constraints on the expressiveness of the logic system, e.g., the language of DDBs is constrained to be function-free. The semantics of AR-sets is typically operational, with roots in research into rule-based expert systems [27, 39]. Much of the interest in investigating ways of combining the two kinds of rule-sets lies in the semantic issues that arise from their acknowledged differences and purported similarities.

Typically, research into DRs has been carried out in almost complete isolation from research into ARs, and vice-versa. More recently, a strong interest in investigating whether there is any intuitive, clean and efficient way of reconciling the two kinds of rules in a single database system has resulted in two positions. The first [51] argues that ARs and DRs lie at opposite ends of a spectrum that progressively trades declarativeness for more powerful expression of active behaviour. Intuitively, this leads to the conjecture that AR systems could be constrained to run on sets of DRs. Unfortunately, no formal semantics has yet been proposed to ground this intuition. The second position [52] argues that a fixpoint interpretation extending the classical operational semantics of DR-sets succeeds in providing a common view of DRs and ARs. Intuitively, this leads to the conclusion that DR systems can be naturally extended to run on AR-sets. Unfortunately, it is clear that some of the expressive power of AR systems is not easily captured in a purely declarative framework (e.g., the possibility of specifying particular coupling modes [19]).

This paper presents a third position. It takes the view that the operational semantics of DDBs and that of ADBs are not intrinsically similar, although in some cases they can be made to be so by constraining the expressive power of either, sometimes both, of them. The intended interpretation of DR-sets is different from the intended interpretation of AR-sets. Some of the reasons for this are: active behaviour and deductive behaviour serve different purposes from the point of view of users; also, they have different conceptual bases, require different tools and styles for the formalization of their operational semantics and are realized by different computational mechanisms. It is difficult to see how one kind of behaviour can be subsumed by the other, without the whole resulting more restrictive than the parts that originally went into its making. Paradigm-merging enterprises are only justified if they deliver more than is provided by each paradigm in isolation and if the original paradigms still retain their distinctive strengths in the resulting merger.

It is probably possible to implement a DR language using an AR language, but it is not clear that certain of the implementation technologies associated with DDBs (especially optimization) would carry over easily. Furthermore, this does not yield a formal semantics for ADBs, which is one of the overall aims.

It is probably not possible to specify the functionality of a comprehensive ADB using a DR language (especially as realized in DDBs). However, certain aspects of active behaviour seem obvious candidates for specification using a DR language (especially conditions, and perhaps events). This approach could be facilitated by allowing the DR language to query the results of actions, as recorded in a history.

The approach followed in this paper leaves the deductive aspects of the database logically pure, while providing a natural formalization of certain aspects of active behaviour in a logical framework. The remaining aspects of ADBs requiring formalization (e.g., the interaction between the evaluator and the transaction manager) are few and do not necessarily benefit from a logical approach.

This paper, therefore, argues in favour of keeping the two operational semantics independent of one another. Instead of attempting to show how one operational semantics subsumes the other, the paper details how to integrate the two in a clean and coherent whole that respects the differences in purpose that users discern in AR- and DR-sets. In particular, no major constraints are imposed on either of them. Rather than forcing one semantics to fit uneasily into the other, the paper argues that the two can be cleanly integrated without renouncing their separate identities and properties.

The main advantage of keeping the two kinds of rules orthogonal is that it maximizes reuse of results from the theory of DDBs (e.g., on termination and on optimization) while not blocking the way for more complex active behaviour to be expressed over them. The paper demonstrates this by first giving a detailed logic-based formalization of three specification languages (one for events, one for conditions, and one for actions) and then showing how an AR language is built from them with an operational semantics that assimilates, but does not interfere with, the operational semantics of DR-sets.

It is important to stress that the paper is not principally concerned with the proposal of either a specific ADB architecture or a specific AR language. The paper does, however, sketch examples of both an ADB architecture and an AR language to indicate more concretely how the logic-based approach might spawn results that lie closer to the implementation level. (Section 9 lists a number of implementation issues that arise in ADBs and indicates the expected pros and cons of adopting the approach described here.)

Due to the standard trade-off between expressiveness and efficiency, a prospective ADB implementor who chooses to use the logic-based approach contributed by this paper may be led to design an architecture and a language that differ from the examples presented here in nontrivial respects. What is expected to remain invariant, as a consequence of adopting our approach, is the fact that the space of choices for both the architecture and the language will be largely constrained by the implementation techniques used to implement DDBs and DR languages over them. The example architecture and the example AR language provided in the paper fulfill the role of highlighting this fact only.

The approach to the formalization of ADB functionality contributed by this paper is centred on the idea of a *history* that generates, through the Kowalski-Sergot event calculus [37, 36, 46], an ordered set of database states. One result of the paper is to show that this notion of a history suffices to endow the specification languages with a precise semantics. Broadly, event and condition specifications are given a Datalog-related operational semantics, while action specifications denote the addition of new axioms to the logical theory that is the representation of the history. Overall, the approach taken here allows for a logic-based architecture comprising active and deductive behaviour with fewer compromises in expressiveness than alternative approaches, and has the additional advantages of permitting a great degree of reuse of techniques and methods developed by the theory and practice of deductive and conventional databases in recent years.

The main contribution of the paper is to present a logic-based approach to the formalization of a large set of syntactic and semantic aspects of ADBs. The most important result made possible by this approach is the definition of the operational semantics of event detection, condition verification and action execution in terms of a well-defined, well-studied semantics which can be computed by tractable algorithms, viz., that of DR programs and queries. It follows from this result that, in many nontrivial cases, the analysis and optimization of AR programs can benefit extensively from well-known methods and techniques used to optimize DR programs and queries.

The approach to the formalization of ADBs contributed in this paper is being applied at Heriot-Watt University to design a much more expressive language than the one exemplified here. This language is the basis for current work on extending the ROCK & ROLL deductive object-oriented database system [4, 25, 26] with powerful ADB functionality. While this paper abstracts away from any particular passive DB system (which is why we chose to illustrate the approach with a leaner, less expressive language), the work mentioned above takes the existing, passive ROCK & ROLL architecture into account for all architectural decisions related to AR processing. Apart from the extensions in the language and in the rule processor of ROCK & ROLL, it is also planned to implement an analyzer and an optimizer for rules expressed in this language. The final goal is to develop an ADB over an object-oriented DDB that is not only expressive and efficient but also provides the programmer with extensive facilities for statically investigating the dynamic behaviour of AR sets. It is specially relevant to this latter goal that the logic-based approach described in this paper endows event detection, condition verification and action execution with a simple, well-studied formal semantics that can be computed by well-known tractable algorithms.

The remainder of the paper is structured as follows. Section 2 introduces the syntax and semantics of ADBs in very general terms. Section 3 discusses alternative architectures for ADBs and the advantages of assigning a central role to the history. Section 4 formally defines the history as the extensional database underlying two DDBs that ultimately give rise to an event-specification language, a condition-specification language and an action-specification language. Section 5 presents the version of the Kowalski-Sergot event calculus that is used in the paper to induce database states from the history. Section 6 is devoted to the formal definition of the three specification languages induced by the history and required in ECA rules. Section 7 shows how an ECA-rule language can be derived from the specification languages previously defined and shows the operational semantics it can be given in terms of that defined in Section 2. Section 8 begins with an analysis of the notion of a rule in both DDBs and ADBs that motivates the approach of keeping their operational semantics maximally independent and briefly discusses recent related work in the light of this analysis. Finally, in Section 9 some conclusions are drawn and an indication of implementation issues and future directions is given.

2 Syntactic and Semantic Aspects of Active Databases

This section provides an overview of the syntax and semantics of ADBs. Knowledge of the syntax and semantic aspects of DDBs is assumed at the level of [12], whose terminology and notation are followed. There are a number of other good research surveys on both DDBs, e.g., [18, 28], and ADBs, e.g., [5, 15, 33, 41, 50], that can be profitably consulted in connection with this paper. The exposition here is necessarily more focussed in comparison.

In contrast with DDBs, for which the characterization given in [12] is standard, ADBs have no widely-accepted characterization. There is broad agreement on the syntax and semantics of ARs but only now are detailed proposals concentrating on providing formal foundations for ADBs in general [8]. This section aims at a system-independent characterization of syntactic and semantic aspects of ADBs to the extent needed for the rest of the paper.

2.1 Syntactic Aspects

An *active rule* ρ can be characterized as a pair $\rho = \langle \sigma, \kappa \rangle$. κ, called the *control-part of* ρ, determines the execution conditions of σ, called the *specification-part of* ρ, by parameterizing for ρ the behaviour of the execution model of rule-sets. Since there is no standard operational semantics for ARs, proposals vary with respect to what execution parameters can be set by κ, and as a consequence the form taken by the control-part of ARs varies accordingly. In contrast, there is broad agreement on the form taken by the specification-part σ. The specification-part σ of an AR rule ρ is a triple $\langle E, C, A \rangle$, where E is an *event specification*, C is a *condition specification*, and A is an *action specification*.

The specification-part of an AR rule is often referred to as an *ECA rule*. In this paper, an ECA rule $\sigma = \langle E, C, A \rangle$ is expressed as '$E : C \leadsto A$'. Such expressions are commonly taken to mean that when an occurrence of the event specified by E has been detected and the condition specified by C holds in the current state of the underlying data, the action specified by A is performed, possibly changing that state and generating new events thereby.

Two particular cases of the general form above are also commonly found in the literature. The first is $C \leadsto A$, obtained by omitting the event specification; the second is $E : A$, obtained by omitting the condition specification. The first is sometimes called a *production rule*, from its origins in artificial intelligence research; the second is called a *trigger*, though some also call 'trigger' the complete form of an ECA rule.

An ECA rule $E : C \leadsto A$ is traditionally read as '`on` E `if` C `then` A'. In this paper, the following alternative reading is preferred '`when (`E `and` C`) do` A'. On the basis of this reading, ECA rules can also be referred to as *when-do rules*. The motivation for this less conventional reading is the lesser risk it carries of calling both deductive (i.e., only-if) and active (i.e., when-do) rules "if-then rules", thereby conflating notions that, as argued in Section 8, are essentially different.

Each of the constituent parts of an AR $E : C \leadsto A$ is an expression in a *specification language*. There is thus a distinct specification language for each part of an ECA rule, viz., an *event*-specification language (ESL), a *condition*-specification language (CSL) and an *action*-specification language (ASL). Since all three are dependent on the definition language of the data model underlying a given database B for their lexicon, one could be precise and speak of the ESL of B, denoting it $\mathcal{E}_B$. Similarly, for the CSL $\mathcal{C}_B$ and the ASL $\mathcal{A}_B$. However, in the remainder of this paper, the subscript B is dropped because the underlying database is of no special concern.

If $\mathcal{Q}$ denotes the query language and $\mathcal{M}$ the manipulation language of the data model underlying B, one can choose to set, for every B, $\mathcal{C} = \mathcal{Q}$ and $\mathcal{A} = \mathcal{M}$, thereby identifying a condition specification with a query expression, and an action specification with an expression that manipulates the database environment. This might obviate the need for independently-defined CSLs and ASLs. However, the characterization of ESLs is still an open issue [16, 29, 31], as existing data models make no provision for an independent representation of events. Also open is the question as to how independent the specification languages are from one another. As pointed out, they share the lexicon, and hence the name space composed of variables and values. The mechanisms needed for detecting, or for constructing the set of, event occurrences may or may not require the imposition on event specifications of a sufficiently distinct semantics for the language to be considered fully independent.

A tentative syntactic characterization of ADBs can be made as follows. Let B be a database, whose definition, manipulation and query languages are, respectively, $\mathcal{D}$, $\mathcal{M}$, $\mathcal{Q}$. Similarly, let $\mathcal{E}$, $\mathcal{C}$, $\mathcal{A}$ be the event-, condition- and action-specification languages whose application-dependent lexicon is induced from that of $\mathcal{D}$. Furthermore, let $\mathcal{C} \supseteq \mathcal{Q}$ and $\mathcal{A} \supseteq \mathcal{M}$. An *ECA rule-set* B^+ *over* B is a set of expressions of the form $E : C \rightsquigarrow A$ such that $E \in \mathcal{E}$, $C \in \mathcal{C}$ and $A \in \mathcal{A}$. An *active rule-program* B^* *over* B is a set of pairs $\langle \kappa, \sigma \rangle$ where κ is a set of parameters for the execution of σ and $\sigma \in B^+$. An *active database* is a pair $\langle B, B^* \rangle$ where B is a database and B^* is an AR-program over B. In Section 6 the syntax and semantics of an example ESL, an example CSL and an example ASL for the class of ADBs proposed in this paper is formally defined.

2.2 Semantic Aspects

Describing the operational semantics of an ADB $\langle B, B^* \rangle$ that is general enough to cover the gamut of existing proposals is a daunting task. Work is underway [8, 9] that specifically addresses that question using the formal specification language Object-Z [24]. In the context of this paper, simplifying assumptions are made. This allows the focus to fall on explaining general features of ADBs, but inevitably excludes important aspects of the functionality provided by specific proposals. The assumptions are the following:

1. It is assumed that procedures exist which, given an expression in a language, return either the bindings resulting from successful evaluation of the expression or an arbitrary symbol (e.g., $\bigtriangledown$) to indicate failure to generate any bindings. Two such procedures are required: a procedure *detect* for *event detection* and a procedure *verify* for *condition verification*.
2. It is assumed that a transition-effecting procedure *execute* exists. This procedure is responsible for *action execution*.
3. If $\mathbf{DB}^\star$ is the set of all possible database states, $B \in \mathbf{DB}^\star$ and $\mathbf{E}$ denotes the set of known event occurrences in the environment of B, then the process that constructed $\mathbf{E}$ up to the point in which the database state is B is assumed to be well-defined.
4. Let an event occurrence be called *driven* if all its implications have been carried out through as many runs as required of the rule-set that specifies the event occurrence. An event occurrence is called *undriven* if it is not yet driven. Then, it is assumed that a nonempty $\mathbf{E}$ can be partitioned into a set of undriven event occurrences and a set of driven event occurrences, e.g., by manipulating and consulting a system-wide log. Let these partitions be denoted by, respectively, *undriven*($\mathbf{E}$) and *driven*($\mathbf{E}$).
5. A total ordering on sets of known event occurrences, denoted by *PriorE*, is assumed to exist — for instance, that induced by a time line that is discrete and linear into the future — along with a function $earliest$ that returns the least element of the resulting ordered set.
6. A total ordering on ECA rule-sets, denoted by *PriorR*, is assumed to exist — for instance, that induced by order of occurrence in a textual representation of the set — along with a function $first$ that returns the least element of the resulting ordered set.
7. An assignment function from the set Var of variables in the specification languages to the set Val of values in $\mathbf{DB}^\star \cup \mathbf{E}$ is assumed to exist.
8. A function *bind* is assumed to exist that, given an expression and a substitution [12], uniformly and simultaneously replaces every variable in the expression that occurs in the domain of the substitution by the value of the variable in the codomain of the substitution.

The signatures of the *binding-generation procedures* and the *transition-effecting procedure* are:

$$\begin{array}{rlll} \textit{detect}: & 2^{\mathbf{E}} & \times \mathcal{E} \rightarrow & 2^{(Var \times Val)} \cup \{\bigtriangledown\} \\ \textit{verify}: & \mathbf{DB}^\star & \times \mathcal{C} \rightarrow & 2^{(Var \times Val)} \cup \{\bigtriangledown\} \\ \textit{execute}: & \mathbf{DB}^\star & \times \mathcal{A} \rightarrow & \mathbf{DB}^\star \times 2^{\mathbf{E}} \end{array}$$

Thus, *detect* takes as input a set of known event occurrences and an event specification, and returns either the variable/value pairs resulting from detecting that the specified event has indeed occurred or else the indication of failure to detect. Similarly, *verify* takes as input a database state and a condition specification, and returns either the variable/value pairs resulting from verifying that the specified condition holds or else the indication of failure to verify. Finally, *execute* takes as input a database state and an action specification, and returns a new database state and a new set of known event occurrences resulting from effecting the state transition.

In this paper only the semantics of ECA rules (understood as ARs without a control part) is formalized. However, a prototype has been built which implements an interpreter for the ECA rule language defined in Section 6. This prototype does cater for some control parameters, e.g., programmers may directly specify relative rule priority and coupling modes [19].

The *operational semantics of an ECA rule-set* B^+ *over a database* B *with respect to a set of known event occurrences* $\mathbf{E}$ is defined by the *evaluateECASet* procedure shown in Figure 1.

$\underline{\mathbf{proc}}$ $evaluateECASet$
  $\Leftarrow Rules, PriorR, PriorE, DBState, Events$
  $\Rightarrow DBState, Events \;\equiv$
  $\underline{\mathbf{begin}}$: $\underline{\mathbf{while}}$ $undriven(Events) \neq \emptyset$ $\underline{\mathbf{do}}$
      $driveEvent(Rules, PriorR, PriorE, DBState, Events, NewDBState, NewEvents)$
      $DBState \leftarrow NewDBState;$
      $Events \leftarrow NewEvents$
    $\underline{\mathbf{od}}$
  $\underline{\mathbf{end}}$.

Figure 1: An Operational Semantics for ECA Rule-Sets (a)

In Figure 1, *evaluateECASet* returns the *NewDBState* resulting from driving all the elements in *Events*, including those that may have been generated in the process of driving previous event occurrences. Note that a saturation state is guaranteed by the requirement that every undriven event be driven, i.e., *NewEvents* must be empty for the procedure to stop. The procedure to drive an undriven event is shown in Figure 2.

$\underline{\mathbf{proc}}$ $driveEvent$
  $\Leftarrow Rules, PriorR, PriorE, DBState, Events$
  $\Rightarrow NewDBState, NewEvents \;\equiv$
  $\underline{\mathbf{begin}}$: $DrivingEvent \leftarrow earliest(undriven(Events), PriorE);$
    $RemainingRules \leftarrow Rules;$
    $\underline{\mathbf{while}}$ $RemainingRules \neq \emptyset$ $\underline{\mathbf{do}}$
      $ToConsider \leftarrow first(RemainingRules, PriorR);$
      $touchRule(ToConsider, DrivingEvent, DBState, Events, NewDBState, NewEvents);$
      $RemainingRules \leftarrow RemainingRules \setminus \{ToConsider\};$
      $DBState \leftarrow NewDBState;$
      $Events \leftarrow NewEvents$
    $\underline{\mathbf{od}}$;
    $logAsDriven(DrivingEvent)$
  $\underline{\mathbf{end}}$.

Figure 2: An Operational Semantics for ECA Rule-Sets (b)

The *DrivingEvent* (see Figure 2) is the earliest (under *PriorE*) known undriven event. *driveEvent* takes each rule in turn. The rule *ToConsider* is the first (as defined by *PriorR*) among the rules not yet considered. It is processed by invoking *touchRule*, shown in Figure 3. *touchRule* returns a new (possibly intermediate) database state and a new (possibly intermediate) set of known event occurrences. As the new state and event occurrences are generated they are also taken into account in the next step of the iteration by means of which *driveEvent* proceeds through the entire rule set. The procedure *logAsDriven* simply records the fact that the *DrivingEvent* has been fully attended to.

```
proc touchRule
    ⇐ ToConsider, DrivingEvent, DBState, Events
    ⇒ NewDBState, NewEvents ≡
    begin : E ← eventSpec(ToConsider);
      C ← conditionSpec(ToConsider);
      A ← actionSpec(ToConsider);
      detect(Events, E, BindingsFromEventDetection);
      if BindingsFromEventDetection ≠ {▽}
        then begin : bind(C, BindingsFromEventDetection, boundC);
                  verify(DBState, boundC, BindingsFromConditionVerification);
                  if BindingsFromConditionVerification ≠ {▽}
                    then begin : bind(A, BindingsFromConditionVerification, boundA);
                              execute(DBState, boundA, NewDBState, NewEvents)
                          end
                  fi
              end
      fi
    end.
```

Figure 3: An Operational Semantics for ECA Rule-Sets (c)

As shown in Figure 3, *touchRule* takes the rule *ToConsider* and fetches its specification parts, viz., E, C, A, using the functions *eventSpec*, *conditionSpec*, and *actionSpec*, respectively, for that purpose. Then, *detect* is requested to return bindings using E. If it succeeds in doing so, the bindings are applied to C, and *verify* is requested to return bindings using *boundC*. If it succeeds in doing so, the bindings are applied to A, and *execute* is requested to generate a new database state and a new set of known events using *boundA*.

Clearly, the crucial steps in the procedures above are the invocations of *detect*, *verify* and *execute*. Not only are complexity and termination dependent on them, but they also stand for very high level abstractions, i.e., they can undergo stepwise refinement up to a reasonable degree of intricacy. Setting $\mathcal{C}$ to $\mathcal{Q}$ and $\mathcal{A}$ to $\mathcal{M}$ alleviates the problem, but events remain an issue, as they have to be generated from manipulations and detected, thereby involving functionality which is not present in passive databases.

As already remarked, variations on *evaluateECASet* abound. The best known include:

- taking a set of database states (e.g., the current and the last, or the current and the last n, etc.) for condition evaluation rather than a single one as above;
- dispensing with ordering relations either by the use of unique system-generated identifiers for events and rules, or else by defaulting the ordering relations to a time line and to textual occurrence, respectively;
- taking a single event, or possibly event type, for event detection rather than a set as above.
- taking a differential approach to action execution such that actions in different rules that cancel out one another (e.g., A = include x, A' = delete x, where A is to be executed before A') are not executed at all.

This section has highlighted and described the syntactic and semantic aspects of ADBs that are needed in later sections. Section 3 presents an archetypal architecture for ADBs and an alternative one centred on the notion of a history, and briefly discusses the advantages of assigning to the latter a more central role than previously done.

3 Architectural Alternatives for Active Databases

The main concern of this paper is with the definition of a new class of logic-based ADBs whose perceived benefits are independent not only of the data model underlying the database but also of the architecture and the specification languages adopted in the ADB. Nevertheless, an example architecture and an example language are described (in this section and in Sections 6 and 7, respectively) to indicate how the logic-based approach might spawn more results at, or close to, the implementation level.

This section draws a contrast between an archetypal ADB architecture and a history-centred one. The most important distinctions between them are:

- In the archetypal architecture the component data stores are instances of some conventional, i.e., non-deductive, data model. In contrast, in the history-centred architecture the component data stores (the log being an exception) are required to be DDBs.
- Because in the archetypal architecture the component data stores are not precisely specified, the semantics of event detection, condition verification and action execution are also left for specific systems to define. In contrast, because in the history-centred architecture the component data stores are required to be DDBs, the semantics of event detection, condition verification and action execution are defined in terms of the semantics of querying and updating DDBs.

The above remarks indicate how the class of ADBs defined in this paper is distinctive in its reuse of query-processing techniques in contrast with the introduction of powerful new modelling constructs. Both in the relational and in the object-oriented case, a conventional, i.e., non-deductive, DB can be seen as a particular case of a deductive one ([28] discusses the relational case, while [25] shows how the same property holds in the object-oriented case). This means that the query processing engine of a DDB subsumes that of a conventional DB. In other words, although this paper always refers to the more expressive class, viz., DDBs, the same reasoning applies to the less expressive class, viz., conventional DBs.

Calling the alternative architecture *history-centred* can be motivated as follows. A history, understood as a store of event data, is a standard component, possibly with a different name, of most ADB systems [7, 14, 19, 30, 31]. However, in none of the systems just cited does the history play as fundamental a role as in the alternative architecture presented below. While in the archetypal architecture the history directly determines only the available event data for the ADB engine to decide which rules to fire, in the history-centred architecture, the history serves, in addition, to determine database states (via an event calculus), thereby allowing action execution to be fully carried out by manipulation of the history alone. In a well-defined sense, the history underlies the definition of event detection, condition verification and action execution in the approach to ADBs contributed by this paper.

3.1 An Archetypal ADB Architecture

Figure 4 depicts the conceptual components of an archetypal ADB (based on the one used as an example in [50]). This figure highlights the components that exist within implemented ADB systems, but abstracts over the implementation details that are characteristic of specific systems. Three data components are used: the first to store *event occurrences*; the second to store the *database*; and the third to store internal control data, produced and required by the ECA-rule interpreter, in the form of a *log*. Three groups of software components interact with the data stores: the *environment* records external event occurrences as they occur, *applications* read and write into the database and record their having done so as internal event occurrences; finally the *ECA-rule interpreter* checks and records its workings in the log while controlling *event detection*, *condition verification* and *action execution*. These three subcomponents of the ECA-rule interpreter interact with the data stores as follows: the procedure for event detection scans event occurrences to check whether, as described in the event-specification part of ECA rules, any ECA rule is to be selected for possible scheduling; the procedure for condition verification requires the database to check whether, as described in the condition-specification part of ECA rules, any selected ECA rule is to be scheduled for execution, and, finally, the procedure for action execution of a scheduled ECA rule reads and writes into the database, records its having done so in the form of event occurrences, and records its internal affairs in the log. Naturally, many variations in Figure 4 are possible. Both data components and software components can be combined, conflated, or arranged in hierarchies, in many ways.

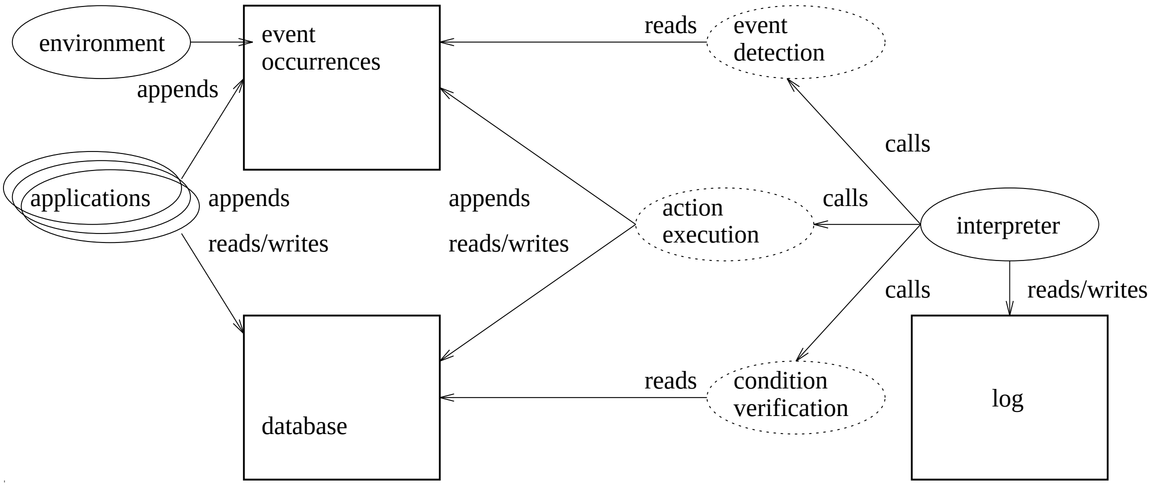

Figure 4: An Archetypal ADB Architecture

A typical cycle in the architecture shown in Figure 4 can be informally described as follows. An application invokes a manipulation, e.g., `put_salary(20)@employee`, which causes an entry to be inserted into the database and an event occurrence to be registered, e.g., `[ev123, 94.04.19.16.44.25.25, put_salary(20)@!456)]`,where `ev123` is the system-assigned event-occurrence identifier, `94.04.19.16.44.25.25` is a system-assigned time stamp, `put_salary(20)@!456)` is an event description and `!456` is the result of dereferencing `employee`. The interpreter must react (or be made to react) to these happenings by fetching the undriven event occurrence(s) and by coordinating the following sequence of computations: event detection *selects* rules whose event-specification part denotes some event occurrence in store; condition verification *schedules* selected rules (if any) whose condition specifications hold true against the database, and, finally, action execution *fires* the action specification of each scheduled rule (if any). If this last step involves writing new data into the database and new event occurrences into their store, then the interpreter must react (or be made to react) to these new happenings until a saturation state is reached in which all known events have been driven. In broad terms, this cycle is captured by the algorithms in Figures 1-3 in Section 2.

3.2 A History-Centred ADB Architecture

This subsection proposes an alternative ADB architecture that, besides being conceptually clearer and potentially free of synchronization problems, provides a clean and intuitive logic-based integration of DDBs with ADBs. It is centred on the notion of a *history* that underlies the Kowalski-Sergot event calculus [36] described later on.

Figure 5 depicts the conceptual components of a history-centred ADB. The data components of the archetypal architecture are still used. The log is construed equivalently in both architectures. However, the store of event occurrences and the database are no longer construed as unanalyzed

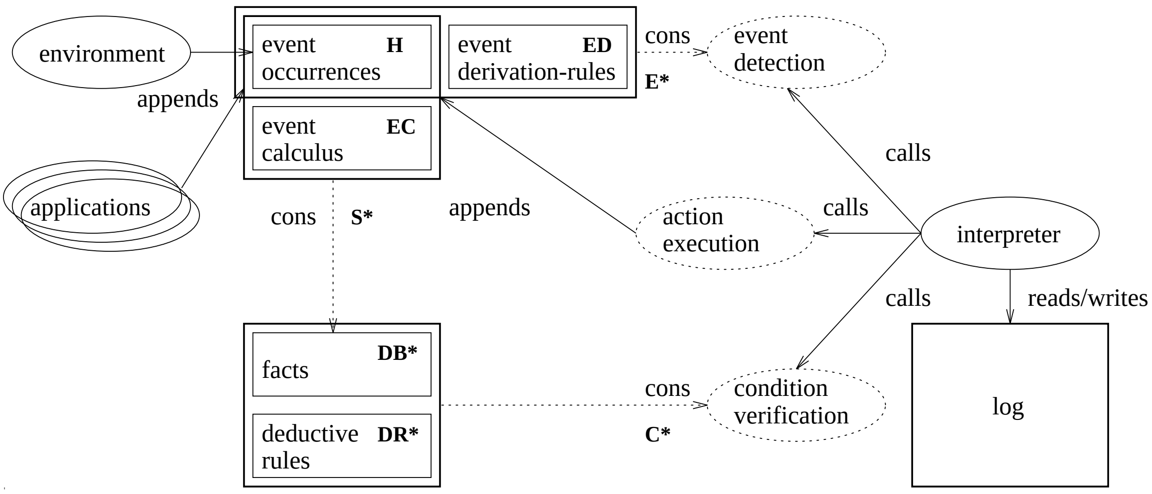

Figure 5: A History-Centred ADB Architecture

abstract data structures.

The first proposal is to consider the store of event occurrences an extensional database (EDB) called the *history* and denoted by **H**. Two intensional databases (IDBs) are defined on top of the history. The first IDB, denoted by **ED**, is made up of *event derivation-rules* over **H**. This determines a first DDB $\mathbf{E}^\star = \mathbf{H} \cup \mathbf{ED}$. Then, the set **E** of *known events* is immediately characterized in terms of the set of all logical consequences of $\mathbf{E}^\star$, denoted by $cons(\mathbf{E}^\star)$. Note that the elements of **E** are potentially complex, in the sense of being characterized by patterns of occurrence (and of recurrence), for instance. Such events are usually referred to in the literature as *composite events*.

The second proposal is to define the database case of the Kowalski-Sergot event calculus as a second IDB, denoted by **EC**, over **H**. This determines a second DDB $\mathbf{S}^\star = \mathbf{H} \cup \mathbf{EC}$. Then, the set $\mathbf{DB}^\star$ of *possible database states* is immediately characterized in terms of the set of all logical consequences of $\mathbf{S}^\star$. (Note that, rather than a single state as is typically the case, the event calculus makes several (potentially all) states accessible.) One can then write a set $\mathbf{DR}^\star$ of *data derivation-rules* over $\mathbf{DB}^\star$. This makes $\mathbf{DB}^\star$ a genuine DDB $\mathbf{S}^\star = \mathbf{DB}^\star \cup \mathbf{DR}^\star$, against which conditions are evaluated.

Figure 5 clearly illustrates that database states are not written into, i.e., there is no in-place update. Conceptually speaking, database states need not even be independently stored, since they follow logically from the history. However, efficiency considerations require that they are (see [36] for a discussion of some advantages of taking this course). The history only needs appending to, in order to record that something has happened in the modelled reality. When action executions model happenings in the modelled reality they do so by appending event occurrences to the history (otherwise they only need to record happenings in the log). The fact that simple append suffices is desirable because logical consequence remains monotonic with respect to $\mathbf{DB}^\star$. Overall, the architecture in Figure 5 minimizes the level of interaction between software and data components and requires no complex synchronization. Well-known deductive mechanisms are responsible for most of the information state of the system as a whole.

By being used extensionally both by event derivation-rules and by the event calculus, the history determines the set of known event occurrences and the set of all database states, and from the latter, the set of all (intensional and extensional) facts can be derived. It follows that for action executions to make events known or to effect state transition they need only append event occurrences to the history. This, in fact, characterizes the precise sense in which the evaluation rules for the specification part of ARs depend solely on the notion of a history.

A typical cycle in this architecture can be informally described as follows: the internal effect of a manipulation, e.g., `put_salary(25)@employee`, is that an event occurrence, e.g., one denoted by `[ev456, 94.04.20.16.44.55.25, put_salary(25)@!789)]` is appended to $\mathbf{H}$ in a clausal form described in Section 4. It follows from $\mathbf{H} \cup \mathbf{ED}$ that the set of known events now includes at least one event occurrence, viz., the one named by `ev456`, and hence, from $\mathbf{H} \cup \mathbf{EC}$, that from `94.04.20.16.44.55.25` onwards, the `salary` property of employee `!789` has value `25`. When the interpreter reacts (or is made to react) to the manipulation it proceeds to coordinate the following sequence of computations: event detection selects rules whose event-specification part, taken as a goal against $\mathbf{E}^{\star}$, evaluates to true; condition verification schedules selected rules (if any) whose condition-specification part, taken as a goal against $\mathbf{C}^{\star}$, evaluates to true; and, finally, action execution fires the action specification of scheduled rules (if any). If this last step involves appending new event occurrences into $\mathbf{H}$ (which induce by logical consequence new known events in $\mathbf{E}$ and new facts in $\mathbf{DB}^{\star}$), then the interpreter must react (or be made to react) to these new happenings until a saturation state is reached in which all known events have been driven.

The goal of this paper is to present a formal, logic-based approach to the problem of integrating deductive and active behaviour in databases. It is important to stress that, in accordance with that goal, the two paradigmatic architectures depicted in this section have no more than a conceptual role to play. In particular, it is not being claimed that, if it were implemented concretely, the idea of a history-based architecture assumed in the rest of the paper would lead to improved performance with respect to the archetypal architecture with which it is being contrasted.

Readers who wish to know more about important recent advances in the topic of concrete architectures for ADBMSs should consult [2, 7, 14, 19, 22, 29, 30, 32, 38, 48].

This section contrasted an archetypal ADB architecture with one that is centred on a history, in the sense that event detection, condition verification and action execution are defined in terms of the history. Thus, motivation has been given for a class of ADBs whose underlying database is deductive without the operational semantics of one interfering extensively with that of the other. Section 4 formally defines the notion of a history, in preparation for a proposal of specification languages for history-centred ADBs, in the sections that follow.

4 A Formal Definition of the History H

Section 3 briefly described and motivated the approach of assigning a more prominent role to the data component that stores event occurrences, referred to as a *history*. This section formally defines this notion as an extensional database which underlies the derivation of complex events and database states.

Conceptually, a history is assumed to be a set of representations of time-stamped event occurrences. This means that, minimally, elements of a history have the form $\langle \tau, \delta \rangle$ where τ denotes a time point and δ describes an event occurrence.

The form assumed by δ and what that form denotes is partly dependent on the nature of the events the history is supposed to record and partly dependent on the execution model of ECA rule-sets. Event types that ADBs are typically interested in include time events (e.g., system-clock ticks, recurrence patterns, relative-time events), structural events (e.g., insert, delete, update), behavioural events (e.g., method invocation, application start-up), transaction events (e.g., start, commit, abort) and environment events (e.g., device signal). A typical case is that of δ taking the form of a DML expression denoting a structural operation, such as setting the salary of an employee.

A more flexible representation is obtained by providing unique, lifelong identifiers for event occurrences, resulting in event occurrences of the form $\langle \upsilon, \tau, \delta \rangle$. Asserting this is equivalent to asserting that the event occurrence denoted by υ happened at time point τ and is described by δ. The remainder of this paper assumes elements of a history to represent event occurrences of the latter form.

In the architecture proposed in Section 3, a history is considered to be an EDB. In this case, the formal language $L(H)$, whose expressions are used to make assertions of the above kind, is the language of a deductive database theory, as defined in [12]. $L(H)$ is defined as follows. Let $EventId$, $TimeLine$, and *EventDescriptor* be nonempty sets of symbols. In $\langle \upsilon, \tau, \delta \rangle$, let $\upsilon \in EventId$, $\tau \in TimeLine$, and let δ be an expression, said to be an event description in *surface form*, over a given domain-specific alphabet. An event description in surface form is assumed to admit a translation, through parsing and dereferencing, into what is called an event description in *internal form* $a_0(a_1, \ldots, a_n)$, where $a_0, a_1, \ldots, a_n$ are elements in *EventDescriptor*; '(', ')', and ',' are punctuation symbols; and notions such as arity are assumed in the usual way when constructing the internal form.

This translation of event descriptions in surface form into internal form is necessarily domain-specific, and, therefore, is not described here, though some examples given later help sharpen intuitions about what the translation is supposed to achieve. Its main purpose is to remove syntactic sugar so that the result is closer to the clausal form in which the history is represented.

Each internal form $a_0(a_1, \ldots, a_n)$ can be translated [20] into a set of clauses Δ with $n+1$ elements, each of which has the form $r_i(\upsilon, a_i)$, where $0 \leq i \leq n$, $\upsilon \in EventId$, and the r_i are domain-specific symbols, interpreted as *roles*. The set of role-denoting symbols is denoted *Roles* and is adjoined to $Const(H)$ to effect the translation of internal forms into clause sets. This set is also application-specific.

A clause set which results from applying this translation procedure to an event description in internal form will be referred to as an event description in *Deliyanni-Kowalski form*, which is a special form of clause set. The advantages of representing complex expressions in this form are presented in [20]. Essentially, there are gains in flexibility, uniformity of representation, and more semantics is made explicit than would otherwise be the case.

For instance, the surface form '`put_salary(20)@self`' yields the internal form 'put(salary, !231, 20)' assuming that dereferencing `self`, which is a symbol in the DML but not in $L(H)$, has yielded '!231'. Then this internal form can be translated into the set of clauses:

referencing-act(ev321, put) $\leftarrow$
referenced-property(ev321, salary) $\leftarrow$
reference-source(ev321, !231) $\leftarrow$
reference-target(ev321, 20) $\leftarrow$

Note that, after parsing and dereferencing, every argument in a internal form, and, hence, in a Deliyanni-Kowalski form, is atomic, since nesting in the surface form is resolved into constant symbols before the translation into internal form.

Every event occurrence [υ, τ, δ] can be represented in the history as a set of clauses of the form $\{$*event(υ)$\leftarrow$, *at(υ, τ)$\leftarrow\}$ $\cup\, \Delta'$, where Δ' is obtained from the Deliyanni-Kowalski form Δ of δ as follows: if $r_i(\upsilon, a_i) \in \Delta$ then *happened(υ, r_i, a_i)$\leftarrow \in \Delta'$. An event occurrence represented as a set of clauses of the above form is said to be in *clausal form*. This last mapping ensures that the representation abstracts over application details by transforming predicate symbols into individual constants.

Thus, the event occurrence `[ev321, 94.04.25.13.06.20.00, put_salary(20)@self]` is expressed in clausal form as follows:

*event(ev231) $\leftarrow$
*at(ev231, 94.04.25.13.06.20.00) $\leftarrow$
*happened(ev231, referencing-act, put) $\leftarrow$
*happened(ev231, referenced-property, salary) $\leftarrow$
*happened(ev231, reference-source, !231) $\leftarrow$
*happened(ev231, reference-target, 20) $\leftarrow$

A **history H** is a set of event occurrences in clausal form. It follows that a history **H** is an extensional database over $Const(H)$ = $EventId \cup TimeLine \cup$ *EventDescriptor* $\cup$ *Roles* and *Pred*(H) = $\{$*event/1, *at/2, *happened/3$\}$. Throughout this paper, p/n expresses the fact that the predicate symbol p has arity n.

This section provided a formal definition for the central notion in the class of ADBs defined by this paper. A history **H** is a particular form of an extensional database containing representations of event occurrences. Section 5 shows how **H** is used with an event calculus to give rise to a sequences of extensional databases in the application.

5 The Event Calculi EC and EC$^+$

This section formally defines event calculi over a history as formally defined in Section 4 and shows how a time-stamped set of database states is generated such that DRs can be used to characterize derived information given any such state.

Over a history defined as a set of event descriptions in Deliyanni-Kowalski form, Kowalski and Sergot [37] defined an *event calculus*. In [37], the event calculus caters for relationships persisting indefinitely into the past as well as into the future. Recently, Kowalski [36] reduced the event calculus to the database case, in which persistence into the past is not assumed. This simplifies the calculus and is the basis for the version, denoted by **EC**, described below. In contrast with the definition in [36], two assumptions made in **EC** are that events have no extended duration and that the relationships that they give rise to hold in the period initiated by the event and contain the said event. These assumptions simplify the formulation and implementation of **EC**, but, otherwise, nothing essential hinges on them and different conventions could be catered for. The calculus is defined by axioms, each of which is a set of clauses.

The **persistence axiom** AxP defines the atoms that hold at a given time point, through the predicate: *holdsAt(Atom, TimePoint). It is defined as the singleton formed by the following clause:

*holdsAt(Atom, TimePoint2) ←
    starts(EventId, Atom),
    happenedAt(EventId, TimePoint1),
    TimePoint1 ≤ TimePoint2,
    ∼(brokenPersistence(EventId, Atom, TimePoint1, TimePoint2))

i.e., *Atom* holds for an open period containing $TimePoint1$ at which the event $EventId$ that gave rise to the relationship denoted by it occurred, unless at some $TimePoint2$ subsequent to $TimePoint1$ the persistence of *Atom* initiated by $EventId$ at $TimePoint1$ is broken. In the above, '∼' can be interpreted as negation-by-finite-failure. Axioms $AxTP$, $AxBP$, and AxS are assumed in the body of the persistence axiom and are defined as follows.

The **time-point axiom** $AxTP$ is defined as the singleton formed by the following clause:

happenedAt(EventId, TimePoint) ←
    *at(EventId, TimePoint)

where $*at$ is the extensional predicate symbol in $L(H)$ defined in Section 4. This shows one circumstance in which **EC** is dependent on the history.

The **broken-persistence axiom** $AxBP$ is defined as the singleton formed by the following clause (assuming axiom $AxIW$ defined below and axiom $AxTP$ defined above):

brokenPersistence(EventId1, Atom, TimePoint1, TimePoint3) ←
    interferesWith(EventId2, Atom),
    happenedAt(EventId2, TimePoint2),
    EventId2 ≠ EventId1,
    TimePoint1 ≤ TimePoint2,
    TimePoint2 ≤ TimePoint3

i.e., the persistence of *Atom* started by $EventId1$ at $TimePoint1$ is broken at $TimePoint3$ if a distinct event $EventId2$, that happened at $TimePoint2$ between $TimePoint1$ and $TimePoint3$, interferes with $Atom$.

The **interference axiom** $AxIW$ is defined as follows (assuming axioms AxS, AxE and AxI defined below):

interferesWith(EventId, Atom1) ←
  ends(EventId, Atom1)
interferesWith(EventId, Atom1) ←
  starts(EventId, Atom1)
interferesWith(EventId, Atom1) ←
  ends(EventId, Atom2),
  incompatible(Atom1, Atom2)
interferesWith(EventId, Atom1) ←
  starts(EventId, Atom2),
  incompatible(Atom1, Atom2)

i.e., $EventId$ interferes with $Atom1$ if it ends or starts it, or else it ends or starts an $Atom2$ that is incompatible with $Atom1$.

Intuitively, two atoms A and B are incompatible if they cannot both be true at the same time point, or, trivially, if they are in fact the same atom. Thus, the **incompatibility axiom** AxI is defined as follows (assuming axiom $AxEx$ discussed below):

incompatible(Atom1, Atom2) ←
  excludes(Atom2, Atom1)
incompatible(Atom1, Atom2) ←
  Atom2 = Atom1

Finally, the **start**, **end** and **exclusion axioms** of **EC**, denoted by AxS, AxE and $AxEx$, respectively are application-specific. The following examples are typical and give an idea of their general form. An element of AxS could be:

starts(EventId, property(Employee, salary, Salary)) ←
  *event(EventId),
  *happened(EventId, referencing-act, put),
  *happened(EventId, referenced-property, salary),
  *happened(EventId, reference-source, Employee),
  *happened(EventId, reference-target, Salary)

where the body uses the extensional predicate symbols $*event$ and $*happened$ in the language of the history defined in Section 4, and, in particular, the roles introduced in the transformation of the surface form `put_salary(Salary)@Employee` to its corresponding Deliyanni-Kowalski form. The expression $property(Employee, salary, Salary)$ denotes an atom in $\mathbf{DB}^\star$ representing a relationship that *starts* to hold after the event occurrence denoted by $EventId$.

The next examples show how AxE and AxS contain clauses that are often linked by the fact that they have shared bodies. Let the surface form of the relevant event descriptions be `make_redundant@Employee`, then:

ends(EventId, property(Employee, status, permanent-contract)) ←
  *event(EventId),
  *happened(EventId, referencing-act, make-redundant),
  *happened(EventId, reference-target, Employee)
starts(EventId, property(Employee, status, redundant)) ←
  *event(EventId),
  *happened(EventId, referencing-act, make-redundant),
  *happened(EventId, reference-target, Employee)

Finally, for properties that are single-valued, one could have in $AxEx$:

excludes(property(Object, Property, Value1), property(Object, Property, Value2)) ←
    Value1 ≠ Value2

An **event calculus EC** is defined as $\mathbf{EC} = AxP \cup AxTp \cup AxBP \cup AxIW \cup AxI \cup AxS \cup AxE \cup AxEx$. Note, however, that the $*holdsAt$ predicate can be used to define others, each of which is referred to as $*holdsAt$-*derived* (HAD). For instance, the HAD predicate $*holdsNow(Atom)$ could be used to point implicitly to the current state:

*holdsNow(Atom) ←
    clock(Now),
    *holdsAt(DatabaseAtom, Now)

under the assumption that Now can be instantiated with the time point $\tau \in TimeLine$ that corresponds to the system clock at invocation time. It is also possible to define $*holdsAt$-derived predicates that define states relative to an original state. In this case, one sets a time point as the origin 0 of an integer line $m, \ldots, -1, 0, +1, \ldots, n$ and states can be defined relative to the origin by an offset. For example, if 0 is now, then -1 is the state between now and the time of the last event occurrence, -2 is the state that held before the last event, and so on. Let HAD denote the set consisting of the $*holdsAt$ predicate and those derived from it, and let AxP^+ denote the set of clauses whose heads have a predicate symbol in HAD.

A **reducible event calculus** $\mathbf{EC}^+$ can now be defined as $\mathbf{EC}^+ = AxP^+ \cup AxTp \cup AxBP \cup AxIW \cup AxI \cup AxS \cup AxE \cup AxEx$. A DDB $\mathbf{S}^\star$ can be defined as $\mathbf{S}^\star = \mathbf{EC} \cup \mathbf{H}$.

Since there is no direct use for any predicate in $\mathbf{EC}^+$ other than those in HAD, and since all other predicates in HAD can be rewritten in terms of $*holdsAt$, the convention is adopted that $IPred(\mathbf{S}^\star) = \{*holdsAt\}$. Further, having shown how a reducible event calculus can be formulated and reduced, the remainder of the paper uses either of the calculi as best suits the context. In the light of the above, the set of all logical consequences of $\mathbf{S}^\star$ consists of facts of the form $*holdsAt(\alpha, \tau)$, each of which denotes that the relationship named by the atom α holds at time point τ.

Precisely which atoms are derived from $\mathbf{S}^\star$ depends on the domain-specific definition of $(AxS \cup AxE \cup AxEx) \subset \mathbf{EC}$. Given $cons(\mathbf{S}^\star)$, $\mathbf{DB}^\star = \{\alpha : *holdsAt(\alpha, \tau) \in cons(\mathbf{S}^\star)\}$ denotes the set of ground atoms naming all relationships that hold now or have held at some time point τ. One can then define, for each $\tau \in TimeLine$, a set $\mathbf{DB}^\star(\tau) = \{\alpha : *holdsAt(\alpha, \tau') \in \mathbf{DB}^\star, \tau = \tau'\}$, called the *extensional database at* τ. By isomorphism with the naturals, this shows how $TimeLine$ induces a sequence of *extensional databases*, each term of which is a subset of $IHB(\mathbf{S}^\star)$, i.e., the logical consequences of $\mathbf{S}^\star = \mathbf{EC} \cup \mathbf{H}$. Subsection 6.2 shows how DRs can be defined over the extensional databases, thereby giving rise to a condition specification language.

This section has formally defined two event calculi and shown how they give rise to a time-stamped sequence of database states. In Section 6 the three specification languages required for ECA rules are given a precise characterization relying on the contents of this and the preceding sections.

6 Specification Languages for ECA-Rule Languages

This section provides detail on how the specification languages used in ECA rules can be defined in the context of the history-centred ADB architecture described in Section 3. It uses the formal definitions of a history and the event calculi given, respectively, in Sections 4 and 5. The three specification languages are brought together in Section 7 to define a language for ECA rules.

Note that clausal form is uniformly used throughout the paper as this is essential for the argument it presents. Since the semantics of all languages described in the remainder of the paper is very well known, there are clear paths for adding syntactic sugar to the languages without compromising any semantic property. It is not in any way part of the approach described that the languages should *look*, as opposed to *be interpreted*, in the way they are in the sections that follow.

On a related point, it is important to stress that the approach contributed by the paper is also not dependent on the data model that underlies the database insofar as its main intent is to use a logical formalism to describe several aspects of active functionality. However, from the viewpoint of an implementor of the contributed approach, there is a significant advantage in selecting a database that already implements a deductive engine, since this engine may then be reused to perform event detection, condition verification and the effects of action execution.

A detailed description of this reuse strategy is beyond the scope of this paper, but the interested reader should not find it difficult to verify the feasibility of the task from the contents of this section, Section 3 and a detailed account of a DDB system, such as e.g., [21, 43, 49] for the relational case or [4, 3] for the object-oriented one.

6.1 An Event Specification Language

This subsection describes how an ESL $\mathcal{E}$ can be characterized as the query language of a DDB $\mathbf{E}^\star$ whose EDB is a history $\mathbf{H}$ and whose IDB $\mathbf{ED}$ is a set of DRs, called *event derivation-rules.*

The set of *known event occurrences* defined by event derivation-rules naturally includes all the event occurrences explicitly recorded in the history. More importantly, it includes event occurrences that are inferred from explicitly asserted ones.

Among known event occurrences some correspond to so-called *composite events.* A typical composite event is the one usually denoted by $(E1, E2)$, with the intended interpretation that events $E1$ and $E2$ are known to have occurred, not necessarily in chronological succession. In this paper, when it is ascertained that a relationship between event occurrences holds, the corresponding composite event is said to have been *detected.* Because such relationships are derived, in the sense that they are verified when the corresponding rule body (or query) is verified, the event occurrence they denote could also be said to be *derived*, and is *composite* in the sense that for its detection to be ascertained it is necessary that the detection (or, as it happens, the lack thereof) of other event occurrences is ascertained.

The event-specification language $\mathcal{E}$ is the set of all queries (as defined in Section 2) over the DDB $\mathbf{E}^\star = \mathbf{H} \cup \mathbf{ED}$, whose extensional part $\mathbf{H}$ is a history (as defined in Section 4) and whose intensional part $\mathbf{ED}$ is defined as follows.

First, information explicitly asserted in the extensional part $\mathbf{H}$ is made available in the intensional part $\mathbf{ED}$ by including in the latter the following rules:

```
event(EventId) ←
      *event(EventId)
at(EventId, TimePoint) ←
      *at(EventId, TimePoint)
happened(ReferringEventId, Role, Referred) ←
      *happened(ReferringEventId, Role, Referred)
```

This introduction of new predicates is standard in DDBs and makes the set of intensionally-defined predicates disjoint from the set of extensionally-defined ones. This has the purpose of making a clear-cut distinction between the set of asserted facts and the set of derived facts, thereby making it simpler to understand many concepts and techniques in the theory of DDBs.

Then, one extends the set of derived events by including rules in $\mathbf{ED}$ that use in their bodies:

- predicates in the set $\{event/1, at/2, happened/3\}$, and
- as expressive a set of auxiliary predicates as one chooses in the light of the trade-offs in semantic complexity that they imply. Typical auxiliary predicates are negation (e.g., $\sim /1$ interpreted as finite failure to prove), aggregates (e.g., *setof*, *count*, *sum*, *max*, *min*, *average*, all binary), arithmetic (e.g., $+, -, /, *, mod$, all ternary) and comparison (e.g., $<, >, \leq, \geq, =$, all binary)

predicates. Access to the system clock through a predicate, e.g., *clock*/1, is also of obvious interest.

With these atoms, one can write rules that relate event occurrences and thus model composite events. To exemplify the approach, consider deriving the occurrence of two events having occurred irrespective of their chronological succession:

conjunction(E1, E2) ←
  event(E1),
  event(E2)

Disjunction and sequence can be similarly defined:

disjunction(E1, E2) ←
  event(E1)
disjunction(E1, E2) ←
  event(E2)
sequence(E1,E2) ←
  at(E1, TimePoint1),
  at(E2, TimePoint2),
  TimePoint2 > TimePoint1

The definition of more complex ways of composing events, such as those defined in [16, 29, 31], proceeds in the same predicate-based style, as opposed to the operator-based style of the above. This is essentially a process of adding new predicates to the intensional part of $\mathbf{E}^\star$, and hence enlarging the set of logical consequences of $\mathbf{E}^\star$. An event specification is then a query over $\mathbf{E}^\star$, i.e., an ESL can be formally defined as

$$\mathcal{E} = \{\leftarrow E_1, \ldots, E_n : \forall 1 \leq i \leq n(F \sqsubseteq E_i \wedge F \in IHB(\mathbf{E}^\star))$$

where $L \sqsubseteq L'$ denotes that L is a substitution-instance of L' as in logic programming in general [12]. When an event specification holds against $\mathbf{E}^\star$ the corresponding event occurrence is said to have been *detected*. Let δ describe a known event occurrence that has been detected, then the known event occurrence can be represented as $\langle \upsilon, \tau, \delta \rangle$ where υ is system supplied, τ is the latest time point occurring in δ, and δ is the specification of the detected event occurrence. For instance, when $sequence(ev435, ev437)$ and $at(ev437, 94.05.12.15.54.19.25)$ hold, the following event occurrence has been detected `[ev453, 94.05.12.15.54.19.25, sequence(ev435, ev437)]`, which, in clausal form, is appended to the history as:

*event(ev453) ←
*at(ev453, 94.05.12.15.54.19.25) ←
*happened(ev453, event-kind, sequence) ←
*happened(ev453, sequence-first, ev435) ←
*happened(ev453, sequence-second, ev437) ←

This allows new known events to be verified taking previous known events into account. For instance, an event specification such as

disjunction(E3, E2),
happened(E2, event-kind, sequence)

now holds, and hence the corresponding event occurrence can be detected and appended to the history as described above.

The process of extending the set of intensional predicates of $\mathbf{E}^\star$ is limited only by the built-in expressiveness of the logic language one decides to employ in **ED**. For relational DDBs, this expressiveness can range from pure Datalog (in which complex recurring patterns that require collecting, counting, summing, and so on, are not expressible) to a language such as $\mathcal{LDL}$ [40] in which special language extensions (e.g., grouping, union, partition, choice) are available that can express aggregates on solutions.

The implications of choosing between pure Datalog and $\mathcal{LDL}$-like languages can be gauged from the thorough discussion in [12] of the added semantic complexity of $\mathcal{LDL}$ in comparison to pure Datalog. As an orthogonal alternative to this spectrum, one can drop relational DDBs and represent $\mathbf{E}^\star$ as a DOOD. In the particular case of ROCK & ROLL [4, 25, 26] this would make available a very expressive language in which to express known events with fine-grained control over the added complexity in the semantics of queries over $\mathbf{E}^\star$, as discussed in the references given above.

6.2 A Condition Specification Language

This subsection describes how a CSL $\mathcal{C}$ can be characterized as the query language over a DDB $\mathbf{C}^\star = \mathbf{DB}^\star \cup \mathbf{DR}^\star$, whose EDB is the set of logical consequences $cons(\mathbf{S}^\star) = \mathbf{DB}^\star$ of $\mathbf{S}^\star$ and whose IDB is a set of DRs, called *data derivation-rules*.

The set of *derived data*, defined by data derivation-rules naturally includes every piece of data of every state that can be inferred by the event calculus $\mathbf{EC}^+$ from the event occurrences recorded in the history $\mathbf{H}$ and the data inferred from it, in turn. It is this property that makes $\mathbf{C}^\star$ a genuine DDB, in a way that is orthogonal to the active capabilities of the system as whole.

Let $IHB(D)$ and $EHB(D)$ denote those subsets of the Herbrand base associated with a database D that correspond, respectively, to $IDB(D)$ and $EDB(D)$. The aspects that distinguish $\mathbf{C}^\star$ from traditional DDBs, as defined in [12], are:

- The EDB of $\mathbf{C}^\star$ is obtained by computing the logical consequences of another DDB, viz., $\mathbf{S}^\star$. This means that to ascertain that a ground atom F is a fact in $EHB(\mathbf{C}^\star)$ one has to ascertain that F is in $IHB(\mathbf{S}^\star)$. This is so because one wants the states that are used for condition verification to reflect the event occurrences recorded in the history, and the event calculus, construed as an IDB, intervenes to determine how this reflection takes place.
- It follows from this and from the discussion in Section 5 that the EDB of $\mathbf{C}^\star$ can be construed as a sequence of time-stamped EDBs, so that there is an EDB at each time point τ for which an event occurrence is recorded in the history $\mathbf{H}$.

With these two points in mind one can characterize the intensional part $\mathbf{DR}^\star$ of $\mathbf{C}^\star$ as follows. First, information inferred from $\mathbf{S}^\star$ is made available in $\mathbf{DR}^\star$ by including in the latter the following rule:

holdsAt(Atom, TimePoint) ←
    *holdsAt(Atom, TimePoint)

and similar ones for each $*holdsAt$-derived predicate, e.g.,

holdsNow(Atom) ←
    *holdsNow(Atom)

The set of specifiable events can be extended by including rules in **ED** that use in their bodies:

- the *holdsAt* predicate (as well as the predicates corresponding to any $*holdsAt$-derived predicates defined for the particular reducible event calculus $\mathbf{EC}^+$ one is using), and
- as expressive a set of auxiliary predicates as one chooses, with the same caveat as in the case of event-derivation rules.

A condition specification is then a query over $\mathbf{C}^*$, i.e., a CSL can be formally defined as

$$\mathcal{C} = \{\leftarrow C_1, \ldots, C_n : \forall 1 \leq i \leq n(F \sqsubseteq C_i \wedge F \in IHB(\mathbf{C}^\star))$$

The data that is derived in this approach is inherently time-stamped. In the typical case, this presupposes that the time point indicating when the atom derived in the head starts to hold is the same as that of the atoms used in the body of the corresponding rule. This characterization has to be done explicitly but a default to that effect can easily be imposed, as follows.

Given a rule of the form

holdsAt(Derived, TimePointD) $\leftarrow$
  holdsAt(Asserted$_1$, TimePointA$_1$),
  ...
  holdsAt(Asserted$_n$, TimePointA$_n$)

such a default would amount to adding to the body above the subgoals

  TimePointD = TimePointA$_1$,
  ...
  TimePointA$_{n-1}$ = TimePointA$_n$

This suffices to enforce that data is derived for a particular time point only from data that holds at *that* time point. If one further assumes, that

  clock(Now),
  TimePointD = Now

is also present in the body, the default is strengthened to derivation in the current state *only*. If both the above defaults are adopted, the original rule can be written as

holdsAt(Derived) $\leftarrow$
  holdsAt(Asserted$_1$),
  ...
  holdsAt(Asserted$_n$)

with the implicit time point being that of the system clock. In this case, however, the *holdsAt* predicate becomes vacuous, i.e., one can take a further step and write

Derived $\leftarrow$
  Asserted$_1$,
  ...,
  Asserted$_n$

which is just what one would write if multiple states were not involved.

If $*$*holdsAt*-derived predicates are defined that provide means to refer to states by indices instead of absolute time points, it becomes easier to write data-derivation rules that involve, e.g., two consecutive states, and this means that some dynamic constraints over sequences of states can be expressed on top of the usual static ones.

6.3 An Action Specification Language

To complete the set of specification languages, an ASL $\mathcal{A}$ can be minimally defined in terms of monotonic appendage of event occurrences in clausal form to the history. On the one hand, this suffices to change the set of known events by changing the extensional base on which the logical consequences of $\mathbf{E}^\star$ are computed; on the other hand, by virtue of the event calculus, this suffices to induce a new state in $\mathbf{DB}^\star$ and hence change the logical consequences of $\mathbf{C}^\star$ that can be computed using $\mathbf{DR}^\star$. Note that, in this approach, when applications make changes to the database they too need only append event occurrences in clausal form to the history.

This means that an ASL $\mathcal{A}$ can be formally defined as the set of all expressions of the form *append*(δ), where δ is an event description in surface form that is mappable into an internal form, denoted by *internal(δ)*. With the assignment of a unique event identifier, the internal form becomes mappable into its corresponding Deliyanni-Kowalski form. This is a set of clauses denoted by *deliyanni-kowalski(internal(δ))*. Finally, with a time stamp, the Deliyanni-Kowalski form gives rise to an event occurrence in clausal form that can be appended to the history. This is a set of clauses denoted by *clausal(deliyanni-kowalski(internal(δ)))*. (All the above mappings, i.e., *internal*, *deliyanni-kowalski* and *clausal*, are as described in Section 4).)

Unlike ESL and CSL expressions, whose semantics is that of DDB queries (as described in [12]), the semantics of ASL is given by a simple set-theoretic expression. Given a history $\mathbf{H}$ and an event description δ in surface form, the result of evaluating an ASL expression *append*(δ) is the set $\mathbf{H}'$ of clauses defined by

$$\mathbf{H}' = \mathbf{H} \cup \mathit{clausal(deliyanni\text{-}kowalski(internal(}\delta\mathit{)))}$$

From a history $\mathbf{H}'$, one can derive anew both

- the set of known events $cons(\mathbf{E}^\star)'$ by the intervention of the event derivation-rules $\mathbf{ED}$ over $\mathbf{H}'$, and
- the set of known facts $cons(\mathbf{C}^\star)'$ by the intervention of the data derivation-rules $\mathbf{DR}^\star$ over the new base facts that are derivable using the event calculus $\mathbf{EC}^+$ over $\mathbf{H}'$.

Thus, the three specification languages are related by their definability in terms of the history.

Note that to facilitate a more concise expression of ECA rule-sets, it may be useful to provide composition mechanisms over the atomic action *append*(δ). These mechanisms might range from simply sequencing of two atomic actions to making both $\mathcal{E} \subset \mathcal{A}$ and $\mathcal{C} \subset \mathcal{A}$, thereby opening the way for composite actions that are dependent on events and data. This can be realized by admitting statement forms such as *if-then* and *while-do* in the action part of ECA rules.

This section demonstrated how the specification languages used in ECA rules can be defined using the formal definitions of a history and an event calculus given, respectively, in Sections 4 and 5. Section 7 uses the three specification languages to define a language for ECA rules.

7 A Language for ECA Rules

This section uses $\mathcal{E}$, $\mathcal{C}$, and $\mathcal{A}$, viz. the ESL, CSL and ASL, respectively, defined in Section 6 to show how ECA rules can be characterized syntactically and semantically from them.

In Section 2 an ECA rule $E : C \rightsquigarrow A$ was defined as being the specification part σ of an AR ρ. Given the characterizations of $\mathcal{E}$, $\mathcal{C}$, and $\mathcal{A}$ in the three preceding subsections it is now possible to strengthen the definition of the specification part of an AR as follows.

Given an AR $\rho = \langle \sigma, \kappa \rangle$, the specification part σ of ρ is an expression of the form $E : C \rightsquigarrow A$ where $E \in \mathcal{E}$, $C \in \mathcal{C}$, and $A \in \mathcal{A}$, and $Var(\mathcal{E}) = Var(\mathcal{C}) = Var(\mathcal{A})$. Note that, under this definition, information can flow from/to event detection, condition verification and action execution by first-order unification.

Consider an ECA rule that detects the event of a change in the salary of an employee, verifies whether the new salary exceeds that of the employee's manager, and, if so, executes the action of raising the latter's salary to that of the employee. This can be written as:

happened(EventId, referencing-act, put),
happened(EventId, referenced-property, salary),
happened(EventId, reference-source, Employee),
happened(EventId, reference-target, EmployeeSalary)
:
  holdsNow(property(Employee, manager, Manager)),
  holdsNow(property(Manager, salary, ManagerSalary)),
  ManagerSalary < EmployeeSalary
$\rightsquigarrow$
    append(put_salary(EmployeeSalary)@Manager)

It is not difficult to devise friendlier syntax to express the same semantics, e.g.

> happened(put_salary(EmployeeSalary)@Employee)
> :
>> get_manager@Employee == Manager,
>> get_salary@Manager == ManagerSalary,
>> ManagerSalary < EmployeeSalary
>
> $\leadsto$
>>> append(put_salary(EmployeeSalary)@Manager)

in which the event specification uses surface form rather than clausal form, and the condition specification uses the syntax of ROCK & ROLL [4] (but could equivalently have used the syntax of Datalog, for example) and the default that the current clock time gives the time point of interest for condition verification.

The reading associated with $E : C \leadsto A$ is thus *when* $(E$ *and* $C)$ *do* A. Given that E and C are query expressions over $\mathbf{E}^\star$ and $\mathbf{C}^\star$, respectively, and that $A = \mathit{append}(\delta)$ is a set-union operation over two sets of clauses, viz. the history $\mathbf{H}$ and the clause set denoted by *clausal(deliyanni-kowalski(internal(*δ*)))*, the operational semantics of the specification part of an individual AR is intuitively defined as: if evaluating E over $\mathbf{E}^\star$ succeeds and evaluating C over $\mathbf{C}^\star$ succeeds, then append to the history the event occurrence defined in A; in any other case, do nothing. At each stage of the evaluation bindings flow bidirectionally by virtue of the set of variable symbols of $\mathcal{E}$, $\mathcal{C}$, and $\mathcal{A}$ being shared between them.

Recall the *evaluateECAset* procedure depicted in Figures 1-3. In *evaluateECAset*, *bind* becomes the unification procedure over terms in $\mathcal{E}$, $\mathcal{C}$ and $\mathcal{A}$. The binding-generation procedures *detect* and *verify* and the transition-effecting procedure *execute* can be more precisely characterized as follows:

- *detect* becomes the query evaluation procedure associated with the DDB $\mathbf{E}^\star$: it takes an event specification E and a DDB $\mathbf{E}^\star$ and returns the corresponding answer-substitutions, i.e. pairs of the form $Var \times Val$ such that, for some substitution-instance F of E, it is the case that $F \in \mathit{cons}(\mathbf{E}^\star)$. In line with the general signature given in Section 2, the ECA-rule interpreter, as it reacts (or is made to react) to new events, logs which events it has driven, thereby making it possible to partition the EDB of $\mathbf{E}^\star$ accordingly.
- Similarly, *verify* becomes the query evaluation procedure associated with the DDB $\mathbf{C}^\star$: it takes a condition specification C and a DDB $\mathbf{C}^\star$ and returns the corresponding answer-substitutions, i.e. pairs of the form $Var \times Val$ such that, for some substitution-instance F of C, it is the case that $F \in \mathit{cons}(\mathbf{C}^\star)$. Note that, whereas the general signature given in Section 2 considers only one database state, $\mathbf{C}^\star$ is capable of making all known states available for condition verification.
- Finally, *execute*, instead of effecting transitions by directly mapping a database state into another while generating new events in the process (as indicated by the general signature given in Section 2), can achieve the same effect simply by adding event occurrences to the history. These include any composite events detected in the process of evaluating event specifications.

Now, as remarked in Section 2, *bind*, *detect*, *verify*, and *execute* are the crucial steps in *evaluateECASet*. This section has defined a language for ECA rules and shown it allows for these four procedures to be so defined that the trade-offs in complexity and termination properties stand out very clearly, and can be kept under control without renouncing expressiveness.

In summary, because E, C and A can be given a logical interpretation, *bind* reduces to first-order unification. In the case of *detect* and *verify*, their complexity and termination properties will be those of the query evaluation procedures associated with $\mathbf{E}^\star$ and $\mathbf{C}^\star$, and these in turn will depend on the expressiveness of the DRs that comprise their intensional parts. Finally, *execute* amounts to no more than a set-union operation. In Section 8 a brief comparison of related work on the incorporation of active capabilities into DDBs is given.

8 Related Work

In this section some of the more recent work on integrating ADBs and DDBs is briefly compared with the approach described in this paper. This latter approach derives from the perception that, in all but the most trivial cases, ARs and DRs are more different in semantic terms than their syntactic similarities would suggest. To substantiate this, consider the following major contrasts between DRs and ARs, and the impact these have on the operational semantics of the two kinds of rule-sets.

A DR-set is an axiomatic logical theory [28, 45]. In particular, syntactically speaking, a DR inherits the (inverted) form of a kind of first-order logic formula known as *material implication*. This means that, evaluating a DR yields a value of type **Boolean**, since an expression of the form $H \leftarrow B$ is semantically equivalent to $H \vee \neg B$.[1] The crucial notion upon which the operational semantics of DDBs is constructed is that of classic logical consequence, and this presupposes a single fixed state for the evaluation of expressions.

The above properties of DRs stand in contrast to comparable properties of ARs as follows. An AR-set cannot be taken to be a logical theory of the classic kind. The form of an AR is that of an imperative command that is meant, in principle, to effect a state transition, as the operational semantics given in Section 2 shows. This is the first major contrast with the deductive case: an AR expresses a state transition, i.e., a relation *between* consecutive states, whereas a DR expresses a relation that holds *within* any state.

Each AR $E : C \rightsquigarrow A$, considered in isolation, is equivalent in some sense to the program fragment **if** $(E \wedge C)$ **then** **do** A **od** **fi** with the consequence that the type of E and C is **Boolean**, the type of A is **statement**, and so is the type of $E : C \rightsquigarrow A$. This is not to claim, however, that the operational semantics of an AR *set* $\{\rho_1, \ldots, \rho_n\}$ is not significantly different from the one that would be obtained from simply wrapping all ARs in the set to yield a block such as **begin** $\rho_1; \ \ldots; \ \rho_n$ **end**. In fact, even assuming a fixed predefined order, some aspects of the flow of control in the execution of AR-sets are implicit and may not be gauged entirely from the textual representation alone, as they depend on the sequence of event occurrences and this is not determinate. Further, the explicit part of the control stretches beyond (e.g., when it interacts with the transaction model) the normal flow of execution of simple condition statements. The is the second major contrast with the deductive case: both DRs and ARs assume a general operational semantics but only the latter assumes, in addition, an implicit (and indeterminate) flow of control. The third major contrast is that only the operational semantics of AR-sets has to encompass control parameters that govern the interaction of the AR evaluation process with other system components, e.g., the transaction manager.

The contrasts above seem to justify reconsidering the view that the operational semantics of DR- and AR-sets intuitively lie in a single, simple, continuous spectrum, and can therefore be unified by a simple extension, or restriction as the case may be, of one point in the spectrum into the other.

Two recent proposals typify this view at opposing ends. In [51] a very general operational semantics of ADBs is presented informally and at a very high level of abstraction. It admits a simple restriction that, ultimately, should allow DRs to be processed by the same engine that processes ARs. In [52] a new operational semantics for DDBs, based on *X-Y stratification*, is formalized (see also [53]). It admits a simple extension that, ultimately, should allow ARs to be processed by the same engine that processes DRs. The next paragraphs discuss and compare each of these proposals with the one presented in this paper in terms of the contrasts highlighted in the beginning of this section.

The central argument in [51] is that there is a basic computational engine for rule processing that, by being subject to slight modifications at each point in a spectrum, leads from purely-deductive Datalog rules to very-active Postgres rules. However, the very starting point of the spectrum takes Datalog to be no more than a view definition language, clearly ignoring the fact that DRs rules can uniformly express other forms of invariant relations between elements of the application domain (e.g., business rules, data-model invariants, integrity constraints). Unless one is prepared to accept a more

[1] More precisely, the evaluation of a DR yields bindings for the variables in it that characterize which instances of the rule are true in the current state.

restricted perspective of DDB technology than that already provided by existing implementations, the initial assumption in [51] is somewhat unsatisfactory. Also, it is arguable that some of the required modifications from one point of the spectrum to the next are not as slight as one would wish. Being a position paper, [51] does not provide a detailed formal semantics, and this makes it impossible to gauge to what extent the three major contrasts highlighted in the beginning of this section are reconciled by its approach.

The thrust of [52] is that there is a basic computational engine that can process pure DRs and yet simulate the effect of active behaviour. The contrast between single and multiple states is addressed in a way that is similar to the event calculus in that it defines multiple states by distinguishing them in logical, and not physical, terms and differs from the event calculus in that it incorporates the information that is necessary to distinguish logical states by rewriting the existing rules and introducing names for the strata [52] that model logical states, whereas the event calculus achieves a similar effect by simply adjoining additional rules (those given in Section 5) to constrain the logical consequences of the existing ones. The problem of control is not addressed in [52]. Since the argument stresses that the new semantics is fixpoint-based and hence essentially declarative, it seems that incorporating explicit control has the potential to disrupt the foundations of that approach. Overall, [52] seems to take a restrictive view of active behaviour in the light of current ADB technology. Its undeniable elegance is achieved by avoiding issues such as the means for specifying very complex composite events and to interleave rule execution with the transaction model orthogonally supported by the database system.

The approach in this paper differs from that of both [51] and [52] in that it does not aim at a single computational engine that can process both DRs and ARs. The two kinds of rules, as argued above, are sufficiently different in semantic terms that attempts to unify, as opposed to integrate, them are probably bound to produce a distorted picture of either or both. In contrast with [51], the full power of DDB technology is preserved, and in contrast with [52], there is no barrier to reach the level of current ADB technology in complex event specification and transaction-dependent control. While both [51] and [52] see the task as one of *unification*, this paper argues for *integration* on the grounds that semantic issues block the way to unrestricted unification. Nevertheless, a logical basis for the integration need not be renounced. In fact, the paper has shown that using a logical conception of a history it is possible to construct a simple but very expressive characterization of ECA rules.

The discussion so far has concentrated on the two proposals that represent extreme positions emerging from the two paradigms of concern. Other work addressing the integration of active and deductive behaviour includes [6, 10, 34], which, for lack of space, can only briefly be compared to the approach presented in this paper.

As in [52], the proposal put forward in [34] extends the operational semantics of DDBs to capture some form of active behaviour. However, in this case, the only events that can be specified are structural operations over data, viz., updates characterized by difference relations, and, as in [52], there is no indication that the formalism is capable of incorporating interactions with a transaction manager.

The position expressed in [6] is that of wrapping pure DDBs with a full-fledged model of active behaviour. As a result, it does not provide any logical framework for integration, i.e., it is loosely rather than tightly coupled. With similar objectives, Chimera [10] is built on top of Algres [11]. The proposal has similarities to the one described in this paper, and it builds on the strengths of Algres in terms of expressing dynamic behaviour. Its capabilities for specifying events are limited to structural operations only and, to the best of our knowledge, a unified logic-based account of active behaviour has not been made public. In this case, as in that of [6] the integration is loose when contrasted with that achieved in this paper.

9 Conclusions and Future Directions

This paper has demonstrated how a logic-based approach can be used to formalize a large set of syntactic and semantic aspects of ADBs. The paper has shown that this approach makes it possible to define the operational semantics of event detection, condition verification and action execution in terms of a well-defined, well-studied semantics which can be computed by tractable algorithms, viz., that of DR programs and deductive queries. This means that the analysis and optimization of nontrivial AR programs can benefit extensively from well-known methods and techniques used to optimize DR programs and queries.

Although the paper's main goal was not to propose either a specific ADB architecture or a specific AR language, it did sketch examples of both an ADB architecture and an AR language to indicate more concretely how the logic-based approach might spawn results that lie closer to the implementation level.

Since the main goal is to provide a logical view of active functionality for the formalization of both syntactic and semantic aspects of ADBs, a comprehensive discussion of implementation issues is clearly beyond the scope of the paper. However, some of its features lead to a number of expectations as to how easily ADBs could be implemented using the contributed approach and how efficiently they could perform. Among those expectations, the following are noteworthy:

- Not only does this paper's approach preserve the standard logical semantics of classical CSLs (which subsumes that of RDB query languages as well), but it also shows how ESLs and ASLs are amenable to being given comparable logical semantics.

  It follows that the implementation of the CSL and ASL can exploit standard techniques to implement deductive query processing and database updates, since no novel features are introduced in either. Furthermore, the approach precisely specifies the loci where direct kernel extensions are needed and in doing so facilitates their implementation. At the level of specification languages, the novelty lies in the ESL, but, here again, the approach reduces the semantics of event detection to that of deductive querying, for which, as pointed out above for the CSL, standard implementation techniques are available.

- The advantages of the paper's approach are twofold. Firstly, since event detection is equated to querying, it follows that the event part of an ECA rule can not only be evaluated but also be analyzed and optimized similarly to the condition part of the rule. This is in contrast with languages such as SAMOS and ODE which use order-dependent constructs to assign a semantics to event detection (Petri nets and finite-state automata, respectively) and would, therefore, find it harder to develop techniques to analyze and optimize event specifications. Secondly, if event detection, condition verification and action execution have the semantics of function-free programs, it follows that each of these stages of AR processing can be executed by a terminating procedure with polynomial time complexity (see [1] for details). At the very least, this ensures that three of the most important stages of AR processing may be made to have desirable termination and efficiency properties, whereas in most existing ADBs such properties cannot, in general, be ascertained in advance.

  Of course, an optimization capability per se does not imply that AR processing will be faster for the optimized rule set than for an equivalent, non-declarative, maybe differently optimized, rule set.

- As pointed out above, the approach presented here makes it viable to ensure the tractability of event-, condition- and action-processing stages. However, because the approach is based on a tight integration (i.e., an underlying DDB engine is either presumed or strongly recommended), the overall performance of the ADB is dependent on the performance of the underlying DB engine. (Several performance figures for representative DDB prototypes are reported in [21, 43, 49]. Note that an RDB is a special case of a DDB and the most successful commercial systems have very acceptable performance.)

Although this dependence is sometimes true of other approaches in the case of condition verification, in the approach described in this paper this dependence is further extended to event detection. A trade-off can therefore be characterized between capabilities for AR analysis and optimization (and hence, acceptable average performance) and hand crafted event and condition specification (and hence, performance which may be better or worse depending on the skills of programmers).

- While the approach leads to expectations of performance in the event-, condition- and action-processing stages, it is neutral with respect to several other aspects of the AR execution model (e.g., enforcement of coupling modes, conflict resolution, etc.). That is to say, our approach neither facilitates nor hinders the pursuit of efficient ways of dealing with these stages of AR processing.
- For certain issues, e.g., composite event detection or visibility of previous database states, the logic-based approach, in the form presented here, is probably not restrictive enough. Specifically, an AR programmer can legitimately refer to all past events and all past states. Clearly, more restrictive visibility windows on event histories (such as those specifiable in Sentinel) and on database states (such as that specifiable in Starburst) are particularizations of our approach and, if adopted, should yield significant speed-ups in event detection and condition verification.
- Still with respect to event detection, if restricted visibility is adopted it may be possible to both cache the visible history in main memory and adopt an eager evaluation mode for the derivation of database states (as Kowalski suggests in [36]).

The inspiration for the work described is the work on ROCK & ROLL [4, 3], a deductive object-oriented database system that seamlessly integrates a powerful persistent, object-oriented imperative language with a logic-based query language operating over a shared object-oriented model. The experience of designing and implementing ROCK & ROLL has demonstrated that a logic-based formalization can bring together seemingly incomparable concepts. ROCK & ROLL acknowledges the limits of this task by drawing a clear boundary beyond which other, not necessarily logic-based, formal techniques must be brought into play. Thus, in ROCK & ROLL, the declarative and the imperative components are formally distinct but so tightly integrated that users are not disadvantaged by this underlying fact.

The approach described here follows the same methodological principle, in that it uses classic, well-understood logical constructs to describe, analyze and express the concept set that underlies ADBs, and instead of aiming at achieving a result that unifies two components but yields lower synergy, concentrates on an integrated whole with a higher synergy that justifies embarking on a paradigm-merging enterprise.

The approach to the formalization of ADB functionality described in this paper has been used to specify the semantics of an AR component [23] that will extend the functionality of ROCK & ROLL, which is now being implemented. This should result in an advanced database system that is scheduled to serve as a platform for experimenting with an AR-toolkit which will embody the framework described in [8] extended with tools for rule analysis and rule optimization.

Acknowledgements The work that resulted in this paperhas been funded by the UK Engineering and Physical Sciences Research Council (Grant GR/H4387) and their support is duly acknowledged. Many improvements and corrections were suggested by our colleagues Jack Campin and Andrew Dinn, for which we are sincerely thankful. We also thank the referees for their constructive criticism, which, we believe, has helped to improve the presentation in valuable ways.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley Publishing Company, Reading, MA, USA., 1995.

[2] R. Agrawal and N.H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *[17]*, pages 36–45, 1989. Also available as AT&T Technical Report att-db-tr-89-2.

[3] Maria L. Barja, Alvaro A.A. Fernandes, Norman W. Paton, M. Howard Williams, Andrew Dinn, and Alia I. Abdelmoty. Design and Implementation of ROCK & ROLL: A Deductive Object-Oriented Database System. *Information Systems*, 20(3):185–211, 1995.

[4] Maria L. Barja, Norman W. Paton, Alvaro A.A. Fernandes, M. Howard Williams, and Andrew Dinn. An Effective Deductive Object-Oriented Database Through Language Integration. In Jorge Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 463–474, Santiago, Chile, September 1994. Morgan Kaufmann Publishers, Inc.

[5] Petra Bayer. State-of-the-Art Report on Reactive Processing in Databases and Artificial Intelligence. *The Knowledge Engineering Review*, 8(2):145–171, 1993.

[6] Petra Bayer and Willem Jonker. A Framework for Supporting Triggers in Deductive Databases. In *[42]*, pages 316–329, 1994.

[7] H. Branding, A. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an Open System: The REACH Rule System. In *[42]*, pages 111–126, 1994.

[8] Jack Campin, Norman W. Paton, and M. Howard Williams. Specifying Active Database Systems in an Object-Oriented Framework, 1994. Submitted for publication.

[9] Jack Campin, Norman W. Paton, and M. Howard Williams. A Structured Specification of an Active Database System. *Information and Software Technology*, 37(1):47–61, 1995.

[10] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Active Rule Management in Chimera, 1994. Manuscript.

[11] Stefano Ceri, Stefano Crespi-Reghizzi, Roberto Zicari, Gianfranco Lamperti, and Luigi A. Lavazza. Algres: An Advanced Database System for Complex Applications. *IEEE Software*, 7(4):68–78, July 1990.

[12] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases.* Surveys in Computer Science. Springer-Verlag, Berlin, Germany., 1990. ISBN 3-540-51728-6, 0-387-51728-6.

[13] Stefano Ceri, Katsumi Tanaka, and Shalom Tsur, editors. *Deductive and Object-Oriented Databases (Third International Conference DOOD'93, Phoenix, Arizona, USA)*, LNCS 760, Berlin, Germany., 1993. Springer-Verlag. ISBN 3-540-57530-8, ISBN 0-387-57530-8.

[14] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In Jorge Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 606–617, Santiago, Chile, September 1994. Morgan Kaufmann Publishers, Inc.

[15] Sharma Chakravarthy. A Comparative Evaluation of Active Relational Databases. Technical Report UF-CIS-TR-93-002, Department of Computer and Information Science, University of Florida, Gainesville, FL, USA., January 1993.

[16] Sharma Chakravarthy and Deepak Mishra. Snoop: An Expressive Event Specification Language for Active Databases. Technical Report UF-CIS-TR-93-007, Department of Computer and Information Science, University of Florida, Gainesville, FL, USA., March 1993. Revised and Extended Version of UF-CIS-TR-91-23, September 1991.

[17] James Clifford, Bruce Lindsay, and David Maier, editors. *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, OR, USA., June 1989. ACM Press. SIGMOD Record, 18(2), June 1989.

[18] Subrata Kumar Das. *Deductive Databases and Logic Programming*. Addison-Wesley Publ. Co., Wokingham, England, UK., 1992. ISBN 0-201-56897-7.

[19] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M.J Carey, M. Livny, and R. Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM SIGMOD Record*, 17(1):51–70, March 1988.

[20] Amaryllis Deliyanni and Robert A. Kowalski. Logic and Semantic Networks. *Communications of the ACM*, 22(3):184–192, March 1979.

[21] Marcia A. Derr, Shinichi Morishita, and Geoffrey Phipps. The Glue-Nail Deductive Database System: Design, Implementation and Evaluation. *The VLDB Journal*, 3:123–160, 1994.

[22] Oscar Díaz, Norman W. Paton, and Peter M.D. Gray. Rule Management in Object-Oriented Databases: A Uniform Approach. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 317–326, Barcelona, Spain., June 1991. Morgan Kaufmann Publishers.

[23] Andrew Dinn, Norman W. Paton, M. Howard Williams, and Alvaro A.A. Fernandes. An Active Rule Language for ROCK & ROLL, 1996. Submitted for publication.

[24] Roger Duke, Paul King, Gordon Rose, and Graeme Smith. The Object-Z Specification Language. Technical Report TR 91-1, Software Verification Research Centre, Department of Computer Science, The University of Queensland, Queensland 4072, Australia, May 1991. (Version 1).

[25] Alvaro A.A. Fernandes. *An Axiomatic Approach to Deductive Object-Oriented Databases*. PhD thesis, Department of Computing and Electrical Emgineering, Heriot-Watt University, Edinburgh, Scotland, UK., September 1995. Available via URL `http://www.cee.hw.ac.uk/Databases/dood.html`.

[26] Alvaro A.A. Fernandes, M. Howard Williams, and Norman W. Paton. Deductive Object-Oriented Databases as Horn Logic-Programs, 1994. Submitted for publication.

[27] C.L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[28] Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and Databases: A Deductive Approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.

[29] Stella Gatziu and Klaus R. Dittrich. Events in an Active Object-Oriented Database System. In *[42]*, pages 23–39, 1994.

[30] Stella Gatziu, Andreas Geppert, and Klaus R. Dittrich. Integrating Active Concepts into an Object-Oriented Database Systems. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types & Persistent Data*, pages 399–415, Nafplion, Greece., August 1991.

[31] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Event Specification in an Active Object-Oriented Database. In *[47]*, pages 81–90, 1992. Also available as AT&T Technical Report att-db-tr-92-6.

[32] Eric N. Hanson. An Initial Report on the Design of Ariel: A DBMS with an Integrated Production Rule System. *ACM SIGMOD Record*, 18(3):12–19, September 1989.

[33] Eric N. Hanson and Jennifer Widom. An Overview of Production Rules in Database Systems. *The Knowledge Engineering Review*, 8(2):121–143, 1993.

[34] John V. Harrison and Suzanne W. Dietrich. Integrating Active and Deductive Rules. In *[42]*, pages 289–305, 1994.

[35] Matthias Jarke, Stefan Eherer, Rainer Gallersdoerfer, Manfred A. Jeusfeld, and Martin Staudt. ConceptBase - A Deductive Object Base Manager. Technical Report 93-14, Aachener Informatik-Berichte/RWTH Aachen, Aachen, Germany., October 1993.

[36] Robert Kowalski. Database Updates in the Event Calculus. *The Journal of Logic Programming*, 1992(1–2):121–146, 1992.

[37] Robert Kowalski and Marek Sergot. A Logic-Based Calculus of Events. *New Generation Computing*, 4:67–95, 1986.

[38] Guy M. Lohman, Bruce Lindsay, Hamid Pirahesh, and K. Bernhard Schiefer. Extensions to Starburst: Objects, Types, Functions and Rules. *Communications of the ACM*, 34(10):94–109, October 1991.

[39] D.P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems. In *Proceedings of AAAI*, pages 42–47, 1987.

[40] Shamim A. Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, Rockville, MD, USA., 1989. ISBN 0-7167-8200-6.

[41] Norman W. Paton, Oscar Díaz, M. Howard Williams, Jack Campin, Andrew Dinn, and Arturo Jaime. Dimensions of Active Behaviour. In *[42]*, pages 40–57, 1994.

[42] Norman W. Paton and M. Howard Williams, editors. *Rules in Database Systems: Proceedings of the 1st International Workshop, Edinburgh, Scotland (30 August–1 September 1993)*, Workshops in Computing Series, London, England, UK., 1994. Springer-Verlag. ISBN 3-540-19846-6, 0-387-19846-6.

[43] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL Deductive System. *The VLDB Journal*, 3:161–210, 1994.

[44] Raghu Ramakrishnan and Jeffrey D. Ullman. A Survey of Research on Deductive Database Systems. Technical report, Department of Computer Science, Stanford University, November 1993.

[45] Raymond Reiter. Towards a Logical Reconstruction of Relational Database Theory. In Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Topics in Information System Series, pages 191–233. Springer-Verlag, New York, NY, USA., 1984. ISBN 0-387-90842-0, 3-540-90842-0.

[46] S.M. Sripada. A Logical Framework for Temporal Deductive Databases. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 171–182, Los Angeles, CA, USA., 1988. Morgan Kaufmann Publishers.

[47] Michael Stonebraker, editor. *Proceedings of the 1992 ACM SIGMOD International Conference on the Management of Data*, San Diego, CA, USA., June 1992. ACM Press. SIGMOD Record 21(2).

[48] Michael Stonebraker and Greg Kemnitz. The POSTGRES Next-generation Database Management System. *Communications of the ACM*, 34(10):78–92, October 1991.

[49] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask, and James Harland. The ADITI Deductive Database System. *The VLDB Journal*, 3:245–288, 1994.

[50] M.H. van der Voort and M.L. Kersten. Facets of Database Triggers. Technical Report CWI-CS-R9122, CWI, Amsterdam, The Netherlands., April 1991. Revised, April 1993.

[51] Jennifer Widom. Deductive and Active Databases: Two Paradigms or Ends of a Spectrum? In *[42]*, pages 306–315, 1994.

[52] Carlo Zaniolo. A Unified Semantics for Active and Deductive Databases. In *[42]*, pages 271–287, 1994.

[53] Carlo Zaniolo, Natraj Arni, and Kayliang Ong. Negation and Aggregates in Recursive Rules: The $LDL++$ Approach. In *[13]*, pages 204–221, 1993.