Expressiveness and Complexity of Active Databases

Philippe Picouet[1] and Victor Vianu[2*]

[1] E.N.S.T., 46 rue Barrault, 75013 Paris, France, picouet@inf.enst.fr
[2] U.C. San Diego, CSE 0114, La Jolla, CA 92093-0114, vianu@cs.ucsd.edu

Abstract. The expressiveness and complexity of several active database prototypes are formally studied. First, a generic framework for the specification of active databases is developed. This factors out the common aspects of the prototypes considered, and allows studying various active database features independently of any specific prototype. Furthermore, each of the prototypes can be specified by specializing certain parameters of the framework. The prototypes considered are ARDL, HiPAC, Postgres, Starburst, and Sybase. Using their formal specifications, the prototypes are compared to each other with respect to expressive power. The results provide insight into the programming paradigm of active databases, the interplay of various features, and their impact on expressiveness and complexity.

1 Introduction

The ability of a database to react to specified events is an increasingly common requirement in advanced database systems. This has led to the emergence of active databases, which provide a qualitatively new paradigm of interaction between the database and the outside world. Numerous models for active databases have been proposed and several major prototypes produced [CCCR+90, MD89, SKdM92, Sto86, WF90] (see also [WC95]). However, many basic aspects of active databases remain little understood, and foundational work in the area is still scarce (e.g., see [AHW95, BM91, HJ91b, FT95, PV95]). Active databases are notoriously complex and hard to deal with. In evaluating existing prototypes and designing future ones, one would benefit from clear answers to questions such as: Which features of current execution models are cosmetic, and which are central to their functionality? Can execution models be simplified? When are two execution models equivalent? This paper addresses such basic questions. Its objective is to understand the computational paradigm introduced by several representative active database prototypes and systems, and particularly the impact of various active database features on their relative expressive power and complexity.

* Work performed in part while this author was visiting E.N.S.T.; supported in part by the National Science Foundation under grant IRI-9221268.

We consider the following prototypes and systems: ARDL [SKdM92], HiPAC [D$^+$88, HLM88, CBB$^+$89, MD89], Postgres [Sto86], Starburst [WF90, Wid91], and Sybase [Syb87]. These are quite diverse, and generally incomparable due to various idiosyncrasies. In order to meaningfully compare the computational paradigms they provide, we make certain simplifying assumptions, spelled out in the paper. For example, we assume the model is relational, events are semantic rather than syntactic, and the execution model is deterministic.

The basic scenario in all prototypes is the following. External programs issue updates to the database. The active database monitors these updates and periodically performs actions in response to specified update events. The actions result in further updates. The control is passed back and forth between the external program and the trigger system, typically at the boundaries of SQL statements. Active database semantics is usually specified in highly procedural terms by the "execution model" of the system. The final update of the database results from the combined effect of the external program and the trigger program. Consequently, we define the semantics of a trigger program as the mapping associating to each external program the final update performed on the database. The external programs we consider are essentially embedded SQL programs, in the style of C+SQL. Thus, equivalent trigger programs must generate the same final database update for each C+SQL external program.

In order to be able to make formal statements about the prototypes, we provide precise procedural semantics for each of them, subject to our unifying assumptions. To do this effectively, we first describe a generic active database framework which factors out the common aspects of all prototypes considered. Then each prototype description is obtained as a specialization of the generic framework by specifying certain parameters. The parameters include: the type of delta relations used, the coupling modes, the scheduling discipline, etc. We believe that the articulation of the generic framework is an important contribution of this paper. First, this provides a skeleton that allows precise, unambiguous specification of various active databases. Second, the generic framework provides a convenient abstraction that allows discussing and comparing various active database features independently of specific prototypes. Indeed, our first group of results does just that. We examine immediate and deferred triggering within the general framework, and the impact of various types of delta relations and scheduling disciplines within each coupling mode. For example, we show that immediate triggering generates computations of complexity limited to EXPTIME (and lower with some types of delta relations) whereas deferred triggering can generate arbitrary computations (but stays within PSPACE if no multiple rule occurrences are allowed in deferred queues). Such complexity results allow to understand the computational characteristics of various combinations of active database features.

The next group of results looks at the specific prototypes and compares them with respect to expressive power. We obtain a complete classification of the five prototypes, summarized in Figure 3. Results on the complexity of the prototypes are summarized in Figure 2.

Previous foundational work on active databases has mainly focused on proposing powerful models or programming constructs that generalize the main active database systems. Thus, [FT95] provides a model that subsumes most active database prototypes. [HJ91b] introduces a programming language for manipulating "deltas", that can be used to uniformly specify a variety of computations encountered in active databases. In [BM91], an object-oriented model for active databases is introduced. The model uses a very flexible trigger mechanism based on nested transactions. It is shown that the model can simulate the main features of active database systems in a uniform fashion. [AHW95] studies an important problem in practical active database systems: the termination and confluence of production rules.

The present paper builds upon our work in [PV95]. We introduced there a simple, abstract framework for active databases, capturing the interaction of external programs with trigger systems and based on *relational machines* (first introduced in [AV91a, AV95]). These are Turing machines augmented with a relational store, modeling computation in the style of C+SQL. Although very useful for formalizing the basic paradigm of active databases, the model based on relational machines does not make the fine distinctions needed to understand the relative expressiveness of the prototypes and the impact of various features. The general framework developed in the present paper lies much closer to the actual prototypes and fulfills this role.

The paper is organized as follows. The Preliminaries review informally some basic concepts of active databases. The generic framework for active databases is developed in Section 3. The impact of various active database features on expressiveness and complexity is investigated in Section 4 within the generic framework. Section 5 contains the results on the prototypes, and brief conclusions are provided in Section 6.

This paper is based on portions of the thesis [Pic95] and the related journal article [PV] (which subsumes the present paper as well as [PV95]). All proofs can be found there.

2 Preliminaries

Active databases support the automatic triggering of updates in response to "events". These responses are typically specified by so called "ECA" rules of the form:

$$\textbf{on}\ \langle\ \textit{event}\ \rangle\ \textbf{if}\ \langle\ \textit{condition}\ \rangle\ \textbf{then}\ \langle\ \textit{action}\ \rangle$$

Although events may range over various external and internal phenomena, most prototypes restrict events to database updates. Conditions typically involve the current database and some information about the event. Some systems allow conditions to look at more than one version of the database state, e.g., corresponding to the state before the event and the state after the event. Accessing past states is usually done by keeping incremental information in so called *delta* relations. Deltas are relations private to the trigger system and persistent between calls to the trigger system within the same user transaction. In principle, the *action* may be a call to an arbitrary routine. In many cases in relational systems, the action will involve a sequence of insertions, deletions and modifications, and in object-oriented systems it will involve one or more method calls. Note that this may in turn trigger other rules.

A fundamental aspect of active databases concerns the choice of an execution model. We outline several possible ones. Suppose that a user transaction $t = c_1; \ldots; c_n$ is issued, where each of the c_i's is an atomic command. In the absence of active database rules, application of t will yield a sequence

$$\mathbf{I}_0,\ \mathbf{I}_1,\ \ldots,\ \mathbf{I}_n$$

of database states, starting with the original state $\mathbf{I}_0$, and where each state $\mathbf{I}_{i+1}$ is the result of applying c_{i+1} to state $\mathbf{I}_i$. If rules are present, then a different sequence of states might arise. Under *immediate* firing, a rule is essentially fired as soon as its event and condition becomes true; under *deferred* firing, rule application is delayed until after the state $\mathbf{I}_n$ is reached; and under *separate* firing, a process is spawned for the rule action, and executed concurrently with other processes. In the most general execution models, each rule is assigned its own "coupling-mode" (i.e., immediate, deferred, or separate), which may be further refined by associating a coupling-mode between event and condition testing, and between condition testing and action execution.

There is a wide variety of choices for execution models. The prototypes we examine illustrate some of the main ones. In order to meaningfully compare the prototypes, we make in this paper the following unifying assumptions:

- The database model is relational.
  For prototypes specified in other models (such as object-oriented) this requires recasting their models into a relational framework.
- Triggers have access to database relations, as well as to private relations used for bookkeeping. The private relations are persistent between invocations of the trigger system within the same user transaction.
- Events consist of insertions and deletions of tuples into relations (we do not consider modifications). We only consider here semantic events, although some active databases react to syntactic insertions and deletions (such systems are not covered by our framework). Actions are programs causing insertions and deletions of sets of tuples into relations; these use the database state(s) and private relations available to the trigger. Composite events are not considered but can be simulated (e.g., composite events specified by regular expressions can be detected by finite automata maintained in private relations).
- The semantics is deterministic.
  If several rules are triggered simultaneously, a preset priority among them is assumed to ensure determinism. For systems with nondeterministic tuple-at-a-time semantics (e.g., Postgres), we assume the data is ordered and events

consist of insertions and deletions of single tuples (thus we assume Postgres only operates on ordered databases and in conjunction with external programs operating one tuple at a time). If subtransactions are executed concurrently, we ignore the nondeterminism that might arise from the concurrency control, and assume instead a serial execution in order of priority.

The above-listed assumptions result in ignoring or slightly modifying certain features of the prototypes. We aimed at retaining the essential aspects of the data manipulation and execution models of each prototype.

What is the semantics of a trigger program? In the basic active database scenario, the input to a trigger program t is an external program e. The output is the database update resulting from the combined effect of the external program and the trigger program, denoted $t[e]$. Thus, (following the definition provided in [PV95]), we take the semantics of a trigger program t to be the mapping associating to each external program e the aforementioned database update, $t[e]$. This induces a notion of equivalence of trigger programs: two trigger programs t and t' are equivalent if for all external programs e, $t[e] = t'[e]$. Based on this definition, we can next compare trigger languages. We say that trigger language T is *subsumed* by trigger language T' if for each trigger program t in T there exists a trigger program t' in T' such that t and t' are equivalent. And two languages T and T' are *equivalent* if T subsumes T' and T' subsumes T. To make this formal, we need to precisely define (i) what an "external program" is, and (ii) the execution model of trigger programs in each language, in conjunction with the external programs.

Part (ii) is the subject of the next section. For (i), we use a very powerful language modeling SQL embedded in a complete programming language (e.g., C+SQL). We use as a convenient abstraction the language $while_N$ (first defined in [Cha81]). The language provides relation variables $P, Q, R, ...$ and integer variables $i, j, ...$. The basic instructions are $R := \varphi$ where R is a relational variable and φ is a first-order (FO) query (this assigns the answer of φ to R); *increment(i)*, *decrement(i)* where i is an integer variable. We also assume that each program begins with a special instruction *start* and ends with a special instruction *halt*. Additionally, there are two looping constructs: *while* φ *do ...* where φ is a FO condition on the database, and *while* $i > 0$ *do ...* where i is an integer variable. Note that this provides computation in the spirit of C+SQL, where a computationally complete language interacts with the database by FO queries/updates. In particular, $while_N$ expresses all queries and updates over ordered databases (i.e. databases providing a total order relation on domain elements) [Cha81].

Let $\mathbf{D}$ be a (public) database schema. External programs generally use private relations in addition to those in $\mathbf{D}$. As we shall see, trigger programs are set off by updates to database relations. We will call an instruction $R := \varphi$ where $R \in \mathbf{D}$, a *database update instruction* of the external program.

Occasionally, we will need to consider restricted external languages. We mention two. The language *while* is $while_N$ without integer variables. The language FO^* consists of line programs whose instructions are assignments $R := \varphi$, where R is a relation variable and φ is an FO query. Finally, the simplest language we consider is *setflag*, whose programs consist of a single instruction setting some boolean flag (propositional variable) to true.

If T is a trigger language and E an external language, $T[E]$ denotes the set of all updates resulting from the joint effect of external programs in E and trigger programs in T.

3 Generic Framework for Active Databases

The generic framework we present in this section extracts a common skeleton for the specific prototypes considered later. Once this is available, each prototype can be concisely specified by specializing the framework, in particular by providing certain parameters. We believe that such a framework is of interest in its own right, as a common vehicle for specifying in a precise manner the execution models of various trigger systems.

We begin by specifying the syntax of programs in the generic framework, then elaborate on delta relations and queues, and finally provide the semantics of trigger programs.

Syntax. A trigger program t is a 7-tuple $< \mathbf{D}, \mathbf{R}, rules, cpl, ev, pri, \Delta\text{-}type >$ where

- $\mathbf{D}$ is the (public) database schema.
- $\mathbf{R}$ is the schema of t, denoted $sch(t)$, and $\mathbf{D} \subseteq \mathbf{R}$. The relations in $\mathbf{R} - \mathbf{D}$ are the *private relations* of t.
- $rules$ is a set of rules over $sch(t)$. A rule is an expression of the form

  $$condition \rightarrow action$$

  where *condition* is an FO sentence and *action* is an external program. Recall that the most general external programs we consider are $while_N$ programs. Further information on the relations accessible by conditions and actions is given below.
- cpl is a mapping from $rules$ to $\{imm, def\}$ providing the *coupling mode* of each rule (immediate or deferred).
- ev is a multi-valued mapping from $rules$ to the set $\{R^+, R^- \mid R \in sch(t)\}$ called the *event* mapping of t (R^+ represents insertions into R and R^- deletions from R).
- pri is a mapping from $rules$ into $\{1, \ldots, |rules|\}$, called the *priority* mapping of t.
- $\Delta\text{-}type$ is a mapping from $rules$ to $\{global, local\text{-}fixed, local\text{-}fluid\}$.

The meaning of the mappings *cpl, ev* and *pri* is quite intuitive. We elaborate on the delta relations and the Δ-*type* mapping next.

Delta relations. Let t be a trigger program as above. For each $R \in \mathbf{R}$, t uses so called *delta relations* Δ_R^+ and Δ_R^- of the same arity as R. Let $sch(t) = \mathbf{R}$, and $sch_\Delta(t) = \{\Delta_R^+, \Delta_R^- \mid R \in \mathbf{R}\}$. The delta relations in $sch_\Delta(t)$ contain incremental information among two database instances. The particular instances involved are determined in different ways in various contexts and systems.

A given rule r accesses only the delta relations associated to its triggering event(s). More precisely, let $sch(ev(r)) = \{\Delta_R^x \mid R^x \in ev(r)\}$. Then the condition of the rule is an FO sentence over $sch(t) \cup sch(ev(r))$ and *action* is an external program over the same schema, which can access but not modify delta relations.

There are three main ways in which delta relations are used in trigger programs:

- Delta relations can record incremental information between the initial database instance and the current instance: Δ_R^+ keeps the tuples inserted into R and Δ_R^- keeps the tuples deleted from R since the beginning of the computation. These relations are updated *automatically* every time a database update instruction is executed (by the external program or some rule action). These delta relations can be accessed by rules as global variables, unless this is overridden by a local variable declaration as below. We refer to this type of delta relation as *global*.
- Delta relations may record incremental information between the database instance at the time of triggering of some rule, and the current instance. The delta relation in question is then treated by the triggered rule as a local variable. As before, its value evolves automatically. We refer to this mode of utilization of delta relations as *local-fluid*.
- Finally, delta relations may be accessed as local variables as above, except that they record incremental information between two *fixed* database instances; their values do not evolve. This mode is referred to as *local-fixed*.

When rules are triggered, they are placed in a queue of rule occurrences – immediate or deferred, according to the coupling mode specified by the *cpl* mapping. An occurrence of a rule may or may not be parameterized by a delta relation. More specifically, if $\Delta\text{-}type(r) = global$ then there is no parameter. When the rule is executed, any reference to delta relations concerns the current values of the global delta relations. If $\Delta\text{-}type(r) \in \{local\text{-}fixed, local\text{-}fluid\}$ the occurrence of the rule is parameterized by its delta relation(s) in $sch(ev(r))$. This is denoted by $r(\Delta_R^+)$, $r(\Delta_R^-)$, $r(\Delta_R^+, \Delta_R^-)$, etc. Here again there are two possibilities. If $\Delta\text{-}type(r) = local\text{-}fixed$, the value of the delta relation(s) passed as parameter does not change since the time the rule is triggered. If $\Delta\text{-}type(r) = local\text{-}fluid$, the value is local but evolves automatically by keeping track of the updates to the relevant relation. This is described in more detail below.

We will use the following convenient notation for the evolution of delta relations. A *transition* over $sch(t)$ is a pair of database instances $\tau = (\mathbf{I}, \mathbf{J})$ over $sch(t)$. For $R \in sch(t)$, the delta relations associated with τ, denoted $\Delta_R^x(\tau)$ for $x \in \{+, -\}$, are defined by $\Delta_R^+(\tau) = \mathbf{J}(R) - \mathbf{I}(R)$ and $\Delta_R^-(\tau) = \mathbf{I}(R) - \mathbf{J}(R)$.

Suppose we have current delta relations Δ_R^x, $x \in \{+, -\}$, and a new transition τ occurs. The delta relations are updated as follows. The new Δ_R^+ is denoted by $\tau^+(\Delta_R^+, \Delta_R^-)$ and is defined as $(\Delta_R^+ - \Delta_R^-(\tau)) \cup (\Delta_R^+(\tau) - \Delta_R^-)$. Similarly, the new Δ_R^- is denoted by $\tau^-(\Delta_R^+, \Delta_R^-)$ and is defined as $(\Delta_R^- - \Delta_R^+(\tau)) \cup (\Delta_R^-(\tau) - \Delta_R^+)$.

This in effect accumulates the delta relations Δ_R^x and $\Delta_R^x(\tau)$, in a manner similar to the *weak merge* operator of [HJ91a].

Let $\tau_1 = (\mathbf{I}, \mathbf{J})$, $\tau_2 = (\mathbf{J}, \mathbf{K})$ and $\tau_2 \circ \tau_1 = (\mathbf{I}, \mathbf{K})$ be transitions over $sch(t)$. It is easy to verify that for each $R \in sch(t)$ and $x \in \{+, -\}$, $\Delta_R^x(\tau_2 \circ \tau_1) = \tau_2^x(\Delta_R^+(\tau_1), \Delta_R^-(\tau_1))$.

Queue Manipulations. A rule queue is a sequence of rule occurrences, possibly parameterized (the empty queue is denoted by ϵ). The above notation is extended to queues as follows. Let q be a queue $r_1 \ldots r_n$ and τ be a transition. Then $\tau(q)$ denotes the queue $\tau(r_1) \ldots \tau(r_n)$, where $\tau(r_i) = r_i$ if $\Delta\text{-}type(r_i) \in \{global, local\text{-}fixed\}$, and $\tau(r_i) = r(\tau(sch(ev(r)))$ if $r_i = r(sch(ev(r)))$ and $\Delta\text{-}type(r) = local\text{-}fluid$.

There are two kinds of queues: q_{imm} holds rules whose coupling mode is immediate, while q_{def} holds rules whose coupling mode is deferred. At various points in the execution of the program, new rule occurrences are added to the queues. Also, two queues q and q' (which can be immediate or deferred) may be combined to yield new immediate and deferred queues. The way this is done is system dependent. We therefore do not specify in the generic framework a particular discipline for these operations. Instead, we parameterize the semantics by the following three functions:

- $add(r, q)$ returns the queue resulting from adding rule r to the queue q;
- $merge_{imm}(q, q')$ returns the immediate queue resulting from the merging of q and q'.
- $merge_{def}(q, q')$ returns the deferred queue resulting from the merging of q and q'.

Furthermore, we assume that the three functions perform queue manipulations of low complexity. More precisely, each mapping is computable in polynomial time (in the number of rules in the input), where comparison and insertion/deletion of rule occurrences are counted as single operations. This captures very general scheduling disciplines and covers all prototypes.

Semantics. We next describe the semantics of trigger programs. Recall that the semantics of a trigger program t is a mapping from external programs e in $while_N$ to updates $t[e]$ over $\mathbf{D}$. Given t and e, the update $t[e]$ is defined as follows. The definition comprises two main phases:

1. e and t take turns taking control of the computation; e starts out, then t takes control following each execution of an instruction $R := \varphi$ of e, where $R \in \mathbf{D}$ (recall that these are called *database update instructions* of e). When this happens, events are detected and new rules are triggered, both immediate and deferred. The immediate queue is executed until empty, generating a new database instance and deferred queue. Then control is passed back to e, and so on until e halts. At this point the first phase ends; its output is a new database instance and a queue of deferred rules.
2. The deferred rules are executed, producing a final database instance.

We next elaborate on (1) and (2) above. We use the following mutually recursive procedures:

- *exec-program* takes as input a 4-tuple $(f, s, \mathbf{I}, q)$ where f is an external program, s is the *start* instruction or a database update instruction of f, $\mathbf{I}$ a database instance, and q a queue of rules; its output is a pair $(\mathbf{I}', q')$ where $\mathbf{I}'$ is a new database instance, and q' a new queue of rules. The inputs of *exec-program* represent the current (start or database update) instruction in the execution of f, the current database instance, and the current q_{def}. The outputs represent the database instance and the queue of deferred rules after f has halted and all immediate queues have been executed.
- *exec-imm* takes as input a triple $(\mathbf{I}, q_{imm}, q_{def})$ where $\mathbf{I}$ is a database instance, q_{imm} is an immediate queue, and q_{def} a deferred queue. It outputs a pair $(\mathbf{I}', q'_{def})$, where $\mathbf{I}'$ is a new database instance and q'_{def} a new deferred queue, both resulting from the execution of the immediate queue of rules q_{imm}.

Once *exec-program* and *exec-imm* are defined, the update $t[e]$ is defined as follows. Suppose the input database instance is $\mathbf{I}$. The global values of the delta relations are initialized to empty.

1. run *exec-program*$(e, start, \mathbf{I}, \epsilon)$. This either does not end – in which case the result is undefined – or it ends with output $(\mathbf{J}, q_{def})$.
2. execute the following pseudo-code program, where $(\mathbf{J}, q_{def})$ is the output of the first phase:

$q_{imm} := q_{def}$; $q_{def} := \epsilon$; **db** $:=$ **J**
while $q_{imm} \neq \epsilon$ **do**
    begin
    $(\mathbf{db}, q_{def}) :=$ *exec-imm*$(\mathbf{db}, q_{imm}, \epsilon)$;
    $q_{imm} := q_{def}$;
    end;
return db

It remains to describe the programs *exec-program* and *exec-imm*. We begin with *exec-program*. On input $(f, s, \mathbf{I}, q_{def})$, *exec-program* does the following. First, run f on $\mathbf{I}$ starting from instruction s, until a new database update instruction u of f is executed, or the *halt* instruction is reached. In the latter case, *exec-program* stops and outputs $(\mathbf{I}, q_{def})$. Otherwise, let the new database instance at that point be $\mathbf{I}'$ and τ denote the transition $(\mathbf{I}, \mathbf{I}')$. The transition τ affects the global delta relations: each global value of Δ_R^x is replaced by $\tau(\Delta_R^x)$, $x \in \{+, -\}$. The deferred queue q_{def} evolves as well: it is replaced by $\tau(q_{def})$. Next, immediate and deferred rules are triggered as follows. A rule r is triggered if for some $R^x \in ev(r)$, $\Delta_R^x(\tau) \neq \emptyset$ $(x \in \{+, -\})$.

When a rule r is triggered, r is placed in the immediate queue if $cpl(r) = imm$ or in the deferred queue if $cpl(r) = def$. If Δ-$type(r) \in \{local\text{-}fixed$, $local\text{-}fluid\}$, the rule r is parameterized with values for the local delta relations $\Delta_R^x \in sch(ev(r))$. Specifically, in both cases $\Delta_R^x = \Delta_R^x(\tau)$, $x \in \{+, -\}$. The immediate queue q_{imm} consists of the triggered rules, possibly parameterized, for which $cpl(r) = imm$, in increasing order of their priorities as specified by the *pri* mapping. Next, the triggered rules r for which $cpl(r) = def$ are added to q_{def} by repeated application of the *add* function, in arbitrary order (it turns out that the order is irrelevant in the various definitions of *add* as they occur in systems).

We now have an immediate queue q_{imm} which needs to be executed. This is done using procedure $exec\text{-}imm(\mathbf{I}', q_{imm}, q_{def})$, which returns (if it terminates) a new database instance $\mathbf{J}$ and a new deferred queue q'_{def}. It is now time to resume the execution of f, starting from the database update instruction u last executed. Thus, $exec\text{-}program(f, u, \mathbf{J}, q'_{def})$ is executed. This concludes the recursive definition of $exec\text{-}program$.

We now proceed with the description of $exec\text{-}imm$. On input $(\mathbf{I}, q_{imm}, q_{def})$, $exec\text{-}imm$ does the following. If $q_{imm} = \epsilon$ then $exec\text{-}imm$ stops and outputs $(\mathbf{I}, q_{def})$. Otherwise, let r be the first rule occurrence in q_{imm}; r is of the form *condition* $\rightarrow$ *action*, where *condition* is an FO sentence over $sch(t) \cup sch(ev(r))$ and *action* is an external program over the same schema, which does not modify delta relations. If $\Delta\text{-}type(r) = global$ then the global values of the delta relations are used, otherwise the local values (fixed or fluid) provided as parameters are used instead. If *condition* is true, the procedure $exec\text{-}program(action, start, \mathbf{I}, \epsilon)$ is run. If it terminates, an output $(\mathbf{J}, q''_{def})$ is produced. Let τ denote the transition $(\mathbf{I}, \mathbf{J})$. Let q'_{imm} be q_{imm} from which the first rule in the queue is removed and to which τ is applied, and $q'_{def} = \tau(q_{def})$. New immediate and deferred queues are obtained according to the mappings $merge_{imm}$ and $merge_{def}$. Thus, $exec\text{-}imm$ is executed on new input $(\mathbf{J}, merge_{imm}(q'_{imm}, q''_{def}), merge_{def}(q'_{def}, q''_{def}))$.

This concludes the mutually recursive definition of $exec\text{-}imm$ and $exec\text{-}program$, and also the description of the semantics of t.

We shall refer to the generic framework described in this section as *Generic*.

Specializations

Descriptions of the prototypes considered in this paper can be obtained by specializing the generic framework presented above. These descriptions, omitted here due to space limitations, can be found in [PV]. They are based on "snapshots" of the prototypes, as presented in [WC95]. Additionally, recall that we are making certain unifying assumptions, outlined in Section 2.

4 Impact of active database features on expressive power

We next examine in more detail the impact on expressive power of various active database features, using *Generic* as a vehicle to unify the discussion. We organize the discussion around coupling modes: immediate triggering, deferred triggering, and their interaction.

We begin by developing some notation for several specializations of the generic framework. The prototypes correspond to some of these specializations.

Restrictions to the generic framework are denoted by specifying restrictions on *action* (rule actions), *cpl,* *Δ-type*, and the deferred or immediate queues. The restrictions on the queues involve specifying the $add, merge_{imm}$ and $merge_{def}$ mappings, or stipulating some properties that they must satisfy. For example, q_{def}: *bounded* might require that the mappings be defined so that the length of the deferred queues is statically bounded. Specializations of *Generic* are then denoted in the following style:

Generic(*action: while, cpl: def, Δ-type: global, q_{def}: bounded*)

This means that rule actions are restricted to *while* programs, all rules are deferred and use global delta relations, and $add, merge_{imm}$ and $merge_{def}$ are defined such that the deferred queues have statically bounded length. The notation will be self-explanatory, or spelled out when needed.

In the remainder of the paper, we will present two types of expressiveness results. The first type compares trigger languages with respect to subsumption and equivalence. Such are the results of Figure 3, providing the subsumption relationships for the prototypes. The second type of result looks at the power of a trigger language T to express updates in conjunction with some external language E, i.e. characterizes $T[E]$. In order for this to tell us something meaningful about T, the power of E should not overwhelm that of T. In fact, it is useful to minimize the power of E by restricting it to *setflag*, whose programs are limited to setting off the trigger program by turning on a boolean flag.

The second type of expressiveness result is related to the first. If $T[setflag]$ and $T'[setflag]$ are not equal then T and T' cannot be equivalent. This allows us to prove nonequivalence results about trigger languages.

4.1 Immediate triggering

We begin by examining the computational characteristics of immediate triggering. Recall that in the generic framework, immediate queues can generally be nested. Among the active database prototypes with immediate triggering, there are two main approaches to the nesting of immediate queues: (i) allow unbounded nesting (ii) ensure that the nesting is statically bounded. We examine both approaches here.

What is the computational power of immediate triggering with unbounded nesting? It turns out that this essentially provides EXPTIME computation. Intuitively, to perform the bookkeeping involved in immediate triggering with unbounded nesting, one needs a pushdown store. The remainder of the computation is in PSPACE. To arrive at the EXPTIME characterization, we use a result by Cook relating Turing machines with an auxiliary deterministic pushdown automaton (dpda) to time-bounded Turing machines. In particular, it is shown in [C71] that the PSPACE Turing machines with an auxiliary dpda express precisely EXPTIME. Thus, we are able to show the following:

Theorem 1. (i) *All updates expressed by* Generic(action: while, cpl: imm)[setflag] *are in* EXPTIME.

(ii) Generic(action: while, cpl: imm)[setflag] *expresses precisely the* EXPTIME *updates on ordered databases.*

Let us now consider bounded nesting. More precisely, let *Generic*(*action: while, cpl: imm,* q_{imm}*: bounded*) denote the common framework restricted so that rule actions are *while* programs, the coupling mode is immediate, and the depth of nesting of immediate queues is statically bounded. One would expect that the boundedness restriction on nesting of immediate queues would drastically reduce expressiveness. Although likely, this is far from obvious. Indeed, we can show that the complexity of updates expressed with bounded nesting (with *setflag* external programs) is PSPACE. However, it is open whether PSPACE $\neq$ EXPTIME. We are then able to show:

Theorem 2. *If* PSPACE $\neq$ EXPTIME *then* Generic(action: while, cpl: imm, q_{imm}: bounded) *is strictly subsumed by* Generic(action: while, cpl: imm).

Theorems 1 and 2 continue to hold if rule actions are restricted to FO* programs.

4.2 Deferred triggering

We next consider *Generic* restricted to deferred triggering. Recall that, when only deferred rules are present, there is just one deferred queue at any given time (as opposed to immediate triggering, where there may be nested immediate queues). However, unlike the immediate queues, the deferred queue is not generally statically bounded.

Also recall from the description of the generic framework that the deferred queue is treated as an immediate queue when executed.

The computational power of unrestricted deferred triggering is, in some sense, complete. More precisely, there is no complexity bound on the computations that may be generated, and all updates can be expressed if the database is ordered. In the following, the restriction "*queues*: HiPAC" means that the queue manipulation mappings are defined as in HiPAC.

Theorem 3. Generic(action: while, cpl: def, Δ-type: local-fluid, queues: HiPAC) [setflag] *expresses the same updates as* while$_N$*, and all updates on ordered databases.*

Some prototypes limit the power and complexity of deferred triggering by placing restrictions upon the deferred queue. We consider two kinds of restrictions. The first disallows multiple occurrences of rules in the deferred queue. The second allows multiple occurrences of a rule, but only if the occurrences are parameterized by distinct delta relations. For the first restriction, we denote the fact that the mapping *add* is defined so that there are no multiple occurrences of a rule in queues by *queues: no-multiple-rules*. We can show the following:

Theorem 4. (i) *All updates expressible in* Generic(action: while, cpl: def, queues: no-multiple-rules)[setflag] *are in* PSPACE.
(ii) Generic(action: while, cpl: def, queues: no-multiple-rules)[setflag] *expresses precisely the* PSPACE *updates on ordered databases.*

Consider now the second type of restriction: suppose that multiple occurrences of rules are allowed, but only with distinct delta relations (this allows unbounded queues). Denote this restriction by *queues: no-multiple-Δ*. We then obtain a trigger language lying strictly between deferred triggering with bounded queues and deferred triggering with unrestricted multiple occurrences of rules. Before showing this, we need the following complexity characterization:

Theorem 5. (i) *All updates expressible in* Generic(action: while, cpl: def, queues: no-multiple-Δ)[setflag] *are in* EXPSPACE.
(ii) Generic(action: while, cpl: def, queues: no-multiple-Δ)[setflag] *expresses precisely the* EXPSPACE *updates on ordered databases.*

We also note that Theorem 5 continues to hold if rule actions are limited to FO* programs instead of *while*. Indeed, *while* actions can be simulated using cascading rules with FO* actions.

We can now show:

Theorem 6. Generic(action: while, cpl: def, queues: no-multiple-rules) *is strictly subsumed by* Generic(action: while, cpl: def, queues: no-multiple-Δ) *which is strictly subsumed by* Generic(action: while, cpl: def, queues: HiPAC).

Immediate vs. deferred triggering A natural question comes up at this point: how does the power of immediate triggering compare to that of deferred triggering? In general, the two are incomparable. First, one cannot expect deferred triggering to simulate immediate triggering. Indeed, immediately triggered rules can change the public database and thus affect the execution of the external program. This cannot be done by deferred rules. On the other hand, immediate triggering cannot generally simulate deferred triggering because the former cannot detect when the execution of the external program has ended. Things are further complicated by the fact that deferred triggering is generally more powerful *computationally* than immediate triggering. Thus, it is generally not the case that immediate triggering subsumes deferred triggering even if immediate triggering can explicitly test for the end of the execution of the external program.

Suppose we wish to design an active database with immediate and deferred triggering, where the computational discrepancy between the two coupling modes is eliminated. It is clear that one has to use a triggering discipline where deferred rules recursively generated in the course of the execution of immediate rules can be reintegrated into the immediate queue, thus providing immediate triggering with the computational power of deferred rules. While this cannot be done in any of the prototypes, it can be achieved by a hybrid restriction which mixes elements of the queuing disciplines of Starburst and HiPAC. Such a hybrid system is described in [PV].

5 Expressiveness and complexity of the prototypes

We summarize in this section our results on the expressiveness and complexity of the five prototypes. For prototypes P allowing both immediate and deferred rules (i.e. ARDL and HiPAC), we denote by $P(imm)$ and $P(def)$ the restrictions of P allowing only immediate, respectively deferred rules. Many results follow from the study in the previous section of active database features within the general framework.

The first results on the prototypes establish the expressiveness of ARDL, ARDL(*imm*), ARDL(*def*), Starburst, and Sybase in conjunction with the external language *setflag*.

Theorem 7. *ARDL*[setflag]*, ARDL*(imm)[setflag]*, ARDL*(def)[setflag]*, Starburst*[setflag]*, and Sybase*[setflag] *express precisely the* PSPACE *updates on ordered databases.*

We next consider Postgres and HiPAC, which are no longer within PSPACE. Our formalization of Postgres is somewhat of a special case, since it works only on ordered databases and with tuple-at-a-time external programs in *while*, so all statements assume these restrictions.

Theorem 8. (i) *Postgres*[while] *expresses precisely the* EXPTIME *updates on ordered databases.*
(ii) *HiPAC*(imm)[setflag] *expresses precisely the* EXPTIME *updates on ordered databases, and HiPAC*(def)[setflag] *expresses all computable updates on ordered databases.*

We now establish one of the main results of the paper. This provides a complete classification of the prototypes considered, with respect to subsumption and equivalence.

Theorem 9. *The relationships in Figure 3 hold among the prototypes (and their* immediate *and* deferred *restrictions, where appropriate).*

Remark: Further equivalences among prototypes hold if we assume that immediate rules can test explicitly whether the execution of the external program has ended. Specifically:
(i) ARDL(*imm*), ARDL, and Sybase become equivalent, and
(ii) ARDL(*def*) and Starburst are both subsumed by the three prototypes above.

6 Conclusions

This paper makes two main contributions. First, it provides a generic framework for active databases that allows to concisely and unambiguously specify various execution models. We illustrated this for five representative prototypes: ARDL, HiPAC, Postgres, Starburst, and Sybase, subject to certain unifying assumptions. Second, the paper uses the formal specifications of the generic framework and prototypes to investigate the computational paradigm of active databases with respect to expressiveness and complexity.

The generic framework was used to study various active database features independently of any specific prototype. We organized the investigation around coupling modes, immediate and deferred, and considered within each the impact of features such as the types of delta relations used, and queuing disciplines. We obtained results on the complexity and expressive power of various combinations of features. For example:

- The complexity of immediate triggering is essentially EXPTIME, even without delta relations. If the depth of nesting of immediate queues is bounded (as is the case in several prototypes), the complexity goes down to PSPACE.
- Deferred triggering is computationally more powerful than immediate triggering, since there is no complexity bound in this case. Complexity bounds are obtained under various restrictions on the queuing disciplines. If no multiple occurrences of rules are allowed in queues, then the complexity becomes PSPACE. If multiple occurrences of rules are allowed but only with distinct delta relations as parameters, then the complexity is EXPSPACE. The complexity results induce results on the relative expressive power of the various restrictions.

The complexity results are summarized in Figure 1.

Using some of the results obtained within the general framework, we studied the five prototypes and classified them completely with respect to subsumption and equivalence (Figure 3). We also characterized their complexity (Figure 2).

We believe that the results obtained in this paper provide insight into the programming paradigm of active databases, the interplay of various features, and their impact on expressiveness and complexity. In particular, they indicate which features are cosmetic, and which are central to expressiveness. They also provide useful information for the design of new trigger systems with desirable properties (as exemplified by the hybrid system mixing elements of Starburst and HiPAC).

One should keep in mind that the results described here are dependent upon the unifying assumptions made about the prototypes. Many aspects of active databases, such as syntax-based events, nondeterministic semantics, object-oriented features, real-time aspects, etc, were left out in the interest of simplicity. Such aspects are nonetheless important, and deserve separate investigation.

References

[AV91a] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 209–219, 1991.

[AV91b] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1), pages 62–124, 1991.

[AV95] S. Abiteboul and V. Vianu. Computing with first-order logic. *Journal of Computer and System Sciences*, 50(2), pages 309–335, 1995.

[AHW95] A. Aiken, J. Widom and J.M. Hellerstein. State analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems* 20(1), pages 3–41, 1995.

[BM91] C. Beeri and T. Milo. A model for active object oriented databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 337–349, 1991.

[CCCR+90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proc. ACM SIGMOD Int'l. Conf. on the Management of Data*, pages 225–236, 1990.

[Cha81] A. K. Chandra. Programming primitives for database languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 50–62, 1981.

[CBB+89] S. Chakravarthy, et. al. Hipac: a research project in active time-constrained databases management. Technical report, Xerox Advanced Information Technology, July 1989.

[C71] S.A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *J. of the ACM*, 18(1), pages 4–18, 1971.

[D+88] U. Dayal et al. The HiPac project: Combining active databases and timing constraints. In *ACM SIGMOD Record*, 1988.

[FT95] P. Fraternali and L. Tanca. A structured approach for the definition of the semantics of active databases. *ACM Transactions on Database Systems* 20(4), pages 414–471, 1995.

[HLM88] M. Hsu, R. Ladin and D.R. McCarthy. An execution model for active data base management systems. In *Proc. Int'l. Conf. on Data and Knowledge Bases*, pages 171–179, Jerusalem, 1988.

[HJ91a] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 455–468, 1991.

[HJ91b] R. Hull and D. Jacobs. On the semantics of rules in database programming languages. In J. Schmidt and A. Stogny, editors, *Next Generation Information System Technology: Proc. of the First International East/West Database Workshop, Kiev, USSR, October 1990*, pages 59–85. Springer-Verlag LNCS, Volume 504, 1991.

[MD89] D. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. ACM SIGMOD Int'l. Conf. on the Management of Data*, pages 215–224, 1989.

[Pic95] P. Picouet. *Puissance d'expression et Consistance sémantique de bases de données actives* (Expressive Power and Semantic Consistency of Active Databases.) PhD thesis, Ecole Nationale Supérieure de Télécommunications, Paris, 1995.

[PV95] P. Picouet and V. Vianu. Semantics and expressiveness issues in active databases. In *Proc. ACM Symp. on Principles of Database Systems*, 1995.

[PV] P. Picouet and V. Vianu. Semantics and expressiveness issues in active databases. Invited to special issue of *JCSS*, to appear.

[SKdM92] E. Simon, J. Kiernan, and C. de Maindreville. Implementing high level active rules on top of a relational dbms. In *Proc. of Intl. Conf. on Very

Large Data Bases, pages 315–326, 1992.

[Sto86] M. Stonebraker et.al. A rule manager for relational database systems. Technical Report, *The Postgres Papers*, Electronics Research Lab, UCB/ERL M86/85, U. of California, Berkeley, 1986.

[Syb87] Sybase, Inc. Transact-sql user's guide. Technical report.

[WF90] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proc. ACM SIGMOD Int'l. Conf. on the Management of Data*, pages 259–264, 1990.

[Wid91] J. Widom. Deduction in the Starburst production rule system. Technical report, IBM Almaden Research, 1991.

[WC95] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan-Kaufmann, Inc., San Francisco, California, 1995.

Restrictions of *Generic*		T[*setflag*]
coupling mode	queue management	ordered DB
immediate coupling	bounded	PSPACE
	unbounded	EXPTIME
deferred coupling	no-multiple-rules	PSPACE
	no-multiple-Δ	EXPSPACE
	no restriction	all

Fig. 1. Expressive power and complexity of main restrictions of the Generic framework

Coupling mode	Prototypes	T[*setflag*] ordered DB
Immediate coupling	Sybase	PSPACE
	ARDL(*imm*)	PSPACE
	HiPAC(*imm*)	EXPTIME
	Postgres ordered DB tuple at a time semantics	EXPTIME
deferred coupling	ARDL (*def*)	PSPACE
	Starburst	PSPACE
	HiPAC(*def*)	all
mixed coupling	ARDL	PSPACE
	HiPAC	all

Fig. 2. Expressive power and complexity of prototypes

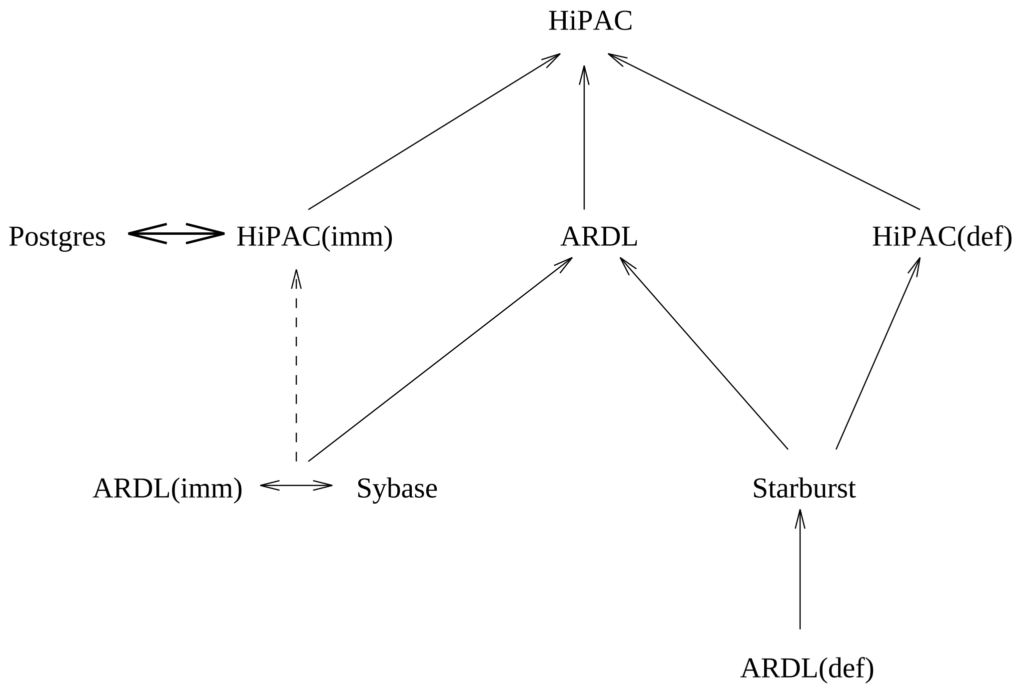

Fig. 3. Relative expressiveness of prototypes. Solid single arrows indicate strict subsumption. The broken single arrow indicates subsumption, and strict subsumption assuming that PSPACE $\neq$ EXPTIME. The double solid arrow indicates equivalence. The double boldfaced arrow indicates equivalence on ordered databases and tuple-at-a-time external programs.

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