

Semantics and Expressiveness Issues in Active Databases

(Extended Abstract)

Philippe Picouet

E.N.S.T. - Paris
picouet@inf.enst.fr

Victor Vianu*

U.C. San Diego
vianu@cs.ucsd.edu

Abstract

A formal framework is introduced for studying the semantics and expressiveness of active databases. The power of various abstract trigger languages is characterized and related to several major active database prototypes such as ARDL, HiPAC, Postgres, Starburst, and Sybase.

1 Introduction

In the past few years there has been tremendous interest in active database systems. Active databases provide “trigger systems” that execute actions in response to specified events. A wealth of active database models have been proposed and several major prototypes produced [CCCR⁺90, Coh86, Han89, MD89, SKdM92, SJGP90, WF90] (see also [WCD95]). However, foundational work in this area is still scarce (e.g., see [AWH92, BM91, HJ91a]). The aim of the present paper is to develop a formal framework for active databases and use it to investigate several basic issues relating to their semantics and expressiveness. We consider questions such as: how should one define the semantics of a trigger system? When are two trigger systems equivalent? Which programming constructs encountered in practical trigger systems are central to their expressiveness, and which are cosmetic? And, are there semantic properties that are desirable for trigger systems?

There is a wide range of possible events and actions. These can be used for various purposes, from constraint maintenance [CW90, CTF88, Mor83] or incremental view maintenance [CW91], to real-time monitoring of temperature in nuclear plants. In the model we introduce, we focus on the common situation where

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

PODS '95 San Jose CA USA

© 1995 ACM 0-89791-730-8/95/0005..\$3.50

the events and actions are updates to the database, as in practical systems such as ARDL [SKdM92], HiPAC [D⁺88, HLM88, MD89], Postgres [Sto86], Starburst [WF90, Wid91], Sybase [Syb87], etc. The basic scenario here 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 of the database according to some discipline. Active database semantics is typically 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 system.

To capture this scenario, we begin by introducing an abstract model for active databases based on “relational machines”. These are Turing Machines augmented with a relational store. The Turing component interacts with the store via first-order queries and updates. Relational machines were introduced in [AV91b, AV95] to capture computations in the spirit of C+SQL, where a database language is embedded in an arbitrarily powerful programming language. In our model, a relational machine represents a given trigger system. External programs are also represented by relational machines, with triggering states that transfer control to the trigger system. We take the semantics of a trigger system to be the transformation whose input is an external program and whose output is the database update resulting from the execution of the external program in conjunction with the trigger system. This definition of semantics induces a notion of equivalence of trigger systems: two trigger systems are equivalent if they have the same transformational semantics over all relational machines. If the external programs use a restricted language, a weaker notion of equivalence relative to a language becomes natural.

Note that the above semantics for triggers mixes apples and oranges: the input is a program and the output is an update. Unfortunately, this cannot be avoided: trigger systems are sensitive to syntactic variations in

input programs. This is disturbing for various reasons. Modifications of external programs which are sound in the sense that they do not change the effect of the program taken in isolation, become unsound in the presence of the trigger system. (Such modifications could be carried out by the programmer, compiler, or optimizer.) Trigger systems that produce equivalent results on equivalent input programs are called “semantically consistent”. We show that semantic consistency is undecidable, but holds in significant special cases. We also introduce a weaker notion of semantic consistency that takes into account trigger states and is satisfied by all trigger systems.

Using the notion of equivalence given by transformational semantics, we show that our active database model based on relational machines subsumes all of the active database prototypes considered here (subject to several simplifying assumptions spelled out in the paper). While most prototypes are much weaker than relational machines, HiPAC turns out to be precisely equivalent to relational machines. This suggests that the relational machine model provides a convenient unifying formalism for investigating a variety of active database questions. In particular, we investigate the expressive power of various languages for trigger systems, relative to various languages for external programs. (E.g., a commonly arising special case is that where the trigger system uses a rule-based language.) These results provide insight into the nature of programming constructs in various existing trigger systems.

Previous foundational work on active databases has mainly focused on proposing powerful models or programming constructs that generalize the main active database systems. Thus, [HJ91a] 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. [AWH92] studies an important problem in practical active database systems: the termination and confluence of production rules.

The paper is organized as follows. Section 2 reviews several update languages used in the paper, and relational machines. Section 3 introduces basic concepts relating to active databases, reviews several execution models encountered in practical systems, and discusses the connection between triggers and external programs. The model of active databases based on relational machines is presented in Section 4, and it is shown that relational machines subsume the trigger systems of ARDL, HiPAC, Postgres, Starburst, and Sybase. The expressive power of several major languages for triggers

relative to various languages for external programs is examined in Section 5. We also consider the expressiveness of the trigger languages of the main prototypes. Section 6 discusses semantic consistency of triggers, and Section 7 provides conclusions. An appendix provides brief descriptions of the prototypes.

2 Preliminaries

We briefly review several relational update languages, and basic notions relating to active databases. We assume familiarity with the terminology of relational databases, including relational calculus and algebra, as in [Ull88, Kan91, AHV95]. Relations are denoted here by P, Q, R, \dots , database schemas by $\mathbf{P}, \mathbf{Q}, \mathbf{R}, \dots$, and the set of instances over a schema \mathbf{R} by $Inst(\mathbf{R})$. The first-order queries (defined by the calculus and algebra) are denoted FO.

Updates and update languages

An *update* over a database schema \mathbf{R} is a mapping from $Inst(\mathbf{R})$ to $Inst(\mathbf{R})$ which is computable and C-generic for some finite set C of constants (i.e., commutes with isomorphisms of the domain that leave C fixed) [AV90, HY84]. The semantics of an update program in some language is the update it defines. The set C in C-genericity accounts for constants that may be explicitly mentioned by the update program. Update languages typically use basic constructs to talk about change to the database, such as insertions or deletions. Query languages can generally be turned into update languages by allowing assignment of answers to database relations. We next review several languages used in the paper, which are update language versions of well-known query languages.

FO*. This denotes programs obtained by composition of assignments statements of the form $R := \varphi$ where R is a relation and φ is an FO query. The semantics of assignment is destructive (R is assigned the value of φ).

While and Fixpoint. *While* extends FO with recursion. It provides relation variables, FO* statements of the form $R := \varphi$, and a looping construct *while* φ *do* where φ is an FO condition. *Fixpoint* is the same as *while* except the semantics of assignment is cumulative (i.e., an assignment $R := \varphi$ *adds* φ to the current content of R). This guarantees termination of *fixpoint* programs in PTIME, whereas *while* programs require PSPACE.

Datalog, Datalog⁻, and Datalog^{-∇}. A Datalog program is a set of rules of the form

$$A_0(t_0) \leftarrow A_1(t_1), \dots, A_n(t_n)$$

where each A_i is a relation, t_i are tuples of variables and constants of appropriate arities, and each variable in t_0 occurs in some t_i , $i > 0$. In a Datalog rule as above, $A_0(t_0)$ is called the *head* and $A_1(t_1), \dots, A_n(t_n)$

the *body* of the rule. A rule firing consists of finding a substitution of variables by constants such that the body of the rule is satisfied, and inserting the tuple t_0 into A_0 . A *stage* consists of one firing of all the rules in parallel. A Datalog program is evaluated by executing consecutive stages until no new tuples are inserted. Datalog⁻ extends Datalog by allowing negative literals in bodies of rules. We consider here (as usual in rule-based active databases) a highly procedural forward chaining semantics for Datalog⁻, in the style of production systems. A negative ground literal $\neg A_i(t_i)$ is satisfied at a given stage if t_i is not currently in A_i (t_i consists of constants from the active domain of the database). Datalog⁻ is evaluated just like Datalog, in consecutive stages of firings of rules. Finally, Datalog⁻⁺ is a further extension allowing for negative literals in heads of rules. The effect of inferring $\neg A_0(t_0)$ is to delete t_0 from A_0 . If a tuple is both inserted and deleted in a stage in the evaluation, the insertion has priority (this arbitrary choice has no significant consequence). It was shown in [AV91a] that Datalog⁻ is equivalent to *fixpoint* and Datalog⁻⁺ is equivalent to *while*.

Relational machines

A *relational machine* (RM) [AV91b, AV95] consists of a Turing machine (TM) augmented with a finite set of fixed-arity relations forming a *relational store*. The machine works as follows. The input and output are designated relations in the store. The tape of the machine is a work tape and is initially empty. In addition to changing its internal state, moving the head on the tape and writing on the tape, the machine can check whether the store satisfies some FO condition, and assign to a relation the result of an FO query on the store. An RM has *arity* k if all relations in the store have arity at most k and the maximum number of variables in its FO queries is at most k . The schema of the relational store of an RM M is denoted by $sch(M)$.

Thus, the relational machine provides arbitrary computation interacting with the database by FO queries and updates. This captures the spirit of external programs, which often use an FO language (say, SQL) embedded in a full programming language (say, C). The TM part of the machine models the use of C, whereas the FO manipulations of the relational store correspond to the SQL queries. Relational machines compute all queries, assuming input databases are ordered (i.e., they provide an order relation among all elements in the active domain).

RMs are equivalent to the following extension of the *while* language with integer arithmetic. Let $while_N$ [Cha81] be the *while* language extended with (i) integer variables i, j, \dots , initialized to zero; (ii) *increment*(i) and *decrement*(i) statements; and (iii) tests of the form $i = 0$ in termination conditions of loops. RMs without the

tape (where the external program is limited to a finite automaton) is precisely equivalent to *while* [AV91b].

An appealing feature of RMs is their robustness. As a useful example, consider an extension of RMs allowing addressable relations (the relational store has a variable, unbounded number of relations of bounded arity). The extended machine remains equivalent to RM [AV95].

3 Active Databases

Basic concepts

We begin with a brief review of active databases, following the development in [AHV95].

Active databases generally support the automatic triggering of updates in response to “events”. In a manner reminiscent of expert systems, forward-chaining of *rules* is generally used to accomplish the response. There are three distinguishing components in an active database: (i) a subsystem for monitoring events, (ii) a set of rules, often called a *rule base*, and (iii) a semantics for rule application, typically called an *execution model*.

Rules typically have the following so-called “ECA” form:

on $\langle event \rangle$ **if** $\langle condition \rangle$ **then** $\langle action \rangle$

Depending on the system and application, the *event* may range over external phenomena and/or over internal events (such as a method call, or inserting a tuple to a relation). Events may be atomic, or may be *composite* events built up from atomic events using, e.g., regular expressions or a process algebra. Events may be essentially boolean, or may “return” a set of tuples that indicate what triggered the event. *Conditions* typically involve parameters passed in by the events, and also the contents of the database. As will be described shortly, several systems permit 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 some systems events are not explicitly specified; essentially any change to the database makes the event true, and leads to testing of all rule conditions. 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. In this paper we focus exclusively on the relational model, although the basic concepts we introduce can clearly be used with other models. We do not consider concurrent access by multiple users.

A fundamental aspect of active databases concerns the choice of an execution model. We outline several possible ones. Suppose that a user transaction $t = c_1; \dots; c_n$ 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

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

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 *concurrent* firing, a separate 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 concurrent), which may be further refined by associating a coupling-mode between event and condition testing, and between condition testing and action execution.

We now examine the semantics of immediate and deferred firing in more detail. We assume for this discussion that the event of each rule is simply 'true'. To illustrate immediate firing, suppose that a rule r with action $d_1; \dots; d_m$ is triggered in state \mathbf{I}_1 of the above sequence of states. Then the sequence of database states might start with

$$\mathbf{I}_0, \mathbf{I}_1, \mathbf{I}'_1, \mathbf{I}'_2, \dots, \mathbf{I}'_m, \dots$$

where \mathbf{I}'_1 is the result of applying d_1 to \mathbf{I}_1 , and \mathbf{I}'_{j+1} is the result of applying d_{j+1} to \mathbf{I}'_j . After \mathbf{I}'_m , the command c_2 would be applied. The semantics of intermediate rule firing is in fact more complex, for two reasons. First, another rule might be triggered during the execution of the action of the first triggered rule. In general, this calls for a recursive style of rule application, where the command sequences of each triggered rule are placed onto a stack. Second, several rules might be triggered at the same time. It is common in this case to assume that the rules are ordered, and that rules triggered simultaneously are considered in that order.

In the case of deferred firing, the full user transaction is completed before any rules are fired, and also, each rule action is executed in its entirety before another rule action is initiated. This gives rise to a sequence of states having the form

$$\mathbf{I}^{orig}, \mathbf{I}^{user}, \mathbf{I}_2, \mathbf{I}_3, \dots, \mathbf{I}^{curr}$$

where now \mathbf{I}^{orig} is the original state, \mathbf{I}^{user} is the result of applying the user-requested transaction, and the states $\mathbf{I}_2, \mathbf{I}_3, \dots, \mathbf{I}^{curr}$ are the results of applying the actions of

fired rules. The sequence shown here might be extended, if additional rules are to be fired. As before, the order of rule firing must be considered, if multiple rules are triggered at a given state. Another issue concerns the database states that are accessed by a rule triggered by a transition from a former to a later state. What states should be used? It is natural to use \mathbf{I}^{curr} as the later state. With regards to the former state, some systems advocate using \mathbf{I}^{orig} , while other systems support the use of one of the intermediate states (where the choice may depend on a complex condition). Suppose now that two rules r and r' are triggered at some state $\mathbf{I}^{curr} = \mathbf{I}_i$, and that r is fired first to reach state \mathbf{I}_{i+1} . The event and/or condition of r' may no longer be true. This raises the question: should r' be fired? A consensus has not emerged in the literature.

As should be clear from the above discussion, there is a wide variety of choices for execution models. Several prototype active database systems have been implemented; As discussed in [HJ91b, HW92, Sto92], each uses a different execution model.

We will provide in this paper simple languages that can serve as the kernel of an active database. These languages do not use explicit events; restrict rule actions to insertions and deletions, and examine only the "current" state at each invocation. In some sense that will be made precise, these can simulate all common active databases. This elucidates issues of expressiveness in active databases. However, in practice richer rule languages and execution models may be desirable to make more manageable the task of developing complex rule bases that enforce a desired behavior.

Major prototypes

There are several major active database prototypes. In this paper we refer to the following: ARDL, HiPAC, Postgres, Starburst, and Sybase¹. Unfortunately, all prototypes are in a state of flux, and many versions of each are described in the literature. In order to be able to make precise statements about them, we use semi-formal descriptions of "snapshots" of these systems that are provided in [Pic], based on their presentation in a preliminary version of [WCD95]. These descriptions are quite lengthy and are omitted here; for convenience, we highlight some of the main features of each prototype in the appendix. In order to be able to compare the different prototypes we additionally make 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.

¹Sybase is actually a commercial product.

- Triggers have access to database relations, as well as to private relations used for bookkeeping (for example, to store *deltas*). 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 consist of insertions and deletions of sets of tuples into relations; these are definable in FO (SQL) using the database state(s) and private relations available to the trigger. In particular, inserted tuples are over the active domain of the database.
- 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 a standard execution order among tuples is chosen to make the semantics deterministic (thus we assume Postgres only operates on ordered databases). 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. Throughout the paper, any formal statement concerning the above prototypes refer to their “snapshots” as described in [Pic] based on [WCD95], and conforming to the above assumptions (see also Appendix).

Triggers vs. external programs

A major emphasis in this paper is the connection between external programs and triggers. In particular, we will consider the expressive power of trigger languages relative to external update languages. The coexistence of external programs and triggers raises fundamental issues of semantics. When an external program is run against an active database, its trigger system modifies the effect of the program. Furthermore, the transformation of the effect of the external program by the trigger system is generally sensitive to syntax, so equivalent external programs have different final effects under the trigger system. Clearly, this only makes sense if it is assumed that the external programmer works in cooperation with the trigger system; otherwise, the programmer loses all control over the semantics of the program

he/she writes. The natural consequence is that the language for external programs must provide, as a first-class citizen, some mechanism for passing the control to the trigger system. And, the semantics of the external program must take this mechanism into account. The syntactic cue for the transfer of control may coincide with the operation of composition of statements (marking, say, the end of an SQL statement), or may be distinct. In most active databases the transfer of control *de facto* occurs at boundaries of SQL statements, although some do allow for explicit transfer of control in the application program (e.g., see [MD89]).

Many of the abstract languages provide natural breaking points where transfer of control to the trigger system can occur. For example, *fixpoint* and *while* (as described here) provide composition. This is not the case for languages with declarative semantics, such as Datalog. It seems that a procedural semantics is needed in order for a language to coexist unambiguously with a trigger system. In the case of Datalog, one could for example transfer control to the trigger system at each evaluation stage, or at the end of the evaluation. To specify the discipline of control transfer for $\text{Datalog}^{(\neg)(\neg)}$ languages, we will use a variation of the *rule algebra* of [IN88]. The rule algebra $\text{RA}(\text{Datalog}^{(\neg)(\neg)})$ over the language $\text{Datalog}^{(\neg)(\neg)}$ is the smallest set of expressions containing the programs in $\text{Datalog}^{(\neg)(\neg)}$ such that

$$\text{if } e, f \in \text{RA}(\text{Datalog}^{(\neg)(\neg)}) \text{ then } (e)^* \in \text{RA}(\text{Datalog}^{(\neg)(\neg)}) \text{ and } (e; f) \in \text{RA}(\text{Datalog}^{(\neg)(\neg)}).$$

The semantics of a $\text{Datalog}^{(\neg)(\neg)}$ program P is *one firing* of the rules in P . Note that P may be empty, in which case its semantics is the identity. The $*$ operator indicates iteration up to a fixpoint. The “;” operator is composition. In the context of a trigger system, it signals transfer of control to the trigger system. Note that $\text{RA}(\text{Datalog})$ is more powerful than Datalog as an update language (despite the fact that Datalog is closed under composition as a query language). Indeed, [Don88] provides an example of Datalog programs P and Q such that $P^*; Q^*$ is not equivalent to any Datalog program: $P = \{R(x, z) \leftarrow R(x, y), R(y, z)\}$ and $Q = \{R(y, x) \leftarrow R(x, y)\}$.

The semantics and expressiveness of an update language in conjunction with a trigger system depends critically on the discipline of transfer provided by the language. In the model of relational machines presented next, the transfer of control is specified using “triggering states”.

4 An Abstract Model for Active Databases

We now present a model for active databases based on relational machines. Let \mathbf{R} be a database schema. A

trigger machine over \mathbf{R} is a relational machine M such that $\mathbf{R} \subseteq \text{sch}(M)$. An *external machine* over \mathbf{R} is a relational machine M augmented as follows: in addition to the start and final states, a set of *triggering states* is specified. External machines are meant to represent external programs. Indeed, their computations consist of periodically issuing SQL (FO) commands to the database, subject to some arbitrarily powerful external control mechanism. Occasionally, they transfer control to the trigger system (whenever they reach a triggering state). Trigger programs are represented by trigger machines, which are normal relational machines (without triggering states). Once again, this models programs that interact with the database using SQL commands, using potentially arbitrary computing power on the side. The semantics of a pair $\langle M_e, M_t \rangle$ consisting of an external machine M_e and a trigger machine M_t over database schema \mathbf{R} is the update $M_t(M_e)$ over $\text{Inst}(\mathbf{R})$ resulting from the following algorithm: (i) M_t is run until a final state is reached; (ii) M_e is run starting in its current state (initially the start state) until a final state or a triggering state is reached; if a final state is reached, M_t is run one last time and the computation stops. If a triggering state is reached, M_t is run starting in its start state until a final state is reached, then (ii) is repeated. The tapes of M_e and M_t , and the relations in $\text{sch}(M_e)$ and $\text{sch}(M_t)$ other than \mathbf{R} are initially empty. The tapes and relations of M_e and M_t are *not* reinitialized between turns in the computation, so are persistent throughout the computation.

We can now define the semantics of trigger machines. Two trigger machines M_t and M'_t over database schema \mathbf{R} are *equivalent*, denoted $M_t \equiv M'_t$, if $M_t(M_e) = M'_t(M_e)$ for all external machines M_e over \mathbf{R} .

The main justification for the model introduced in this section is that it subsumes the trigger system languages of the main active database prototypes. We now state one of the central results of the paper. Recall that formal statements about prototypes refer to their specifications as given in [Pic], based on the descriptions in [WCD95] (see also Appendix).

Theorem 4.1 For each ARDL, HiPAC, Postgres, Starburst, or Sybase program there exists a trigger machine equivalent to it.

Proof (sketch): The simulations of ARDL, Postgres, Starburst, and Sybase programs by trigger machines are straightforward. The necessary bookkeeping needed to simulate each of the execution models can be done using the (persistent) tape and private relations of the machine. The simulation of HiPAC is nontrivial because an unbounded number of triggered rules can await execution at each given moment, and each may have access to the database state at the time of triggering. Thus, an unbounded number of relations must be

remembered in this execution model. Nonetheless, this can still be simulated by a trigger machine with fixed number of relations. The argument is nontrivial, and involves an extension of a result in [AV95] showing that relational machines can be extended with an unbounded number of addressable relations of fixed arity without increase in power. This uses a normal form for relational machines of given arity. A computation in the normal form consists of two phases: (i) an *analysis* phase that extracts from the input all information necessary to carry out the remainder of the computation on the tape alone, and (ii) a *computation* phase involving only the tape, that uses the information provided in the analysis phase. Computation with an unbounded number of relations of fixed arity can be simulated in the second phase using the tape. The proof for trigger machines is more complicated because of the presence of external programs, since there is no uniform bound on the arity of relations across all external programs. Here the analysis phase must be performed again after each turn of the external program, and the information obtained from previous analysis phases must be refined accordingly. \square

Remark: External programs use the trigger system much like an oracle in the course of the computation. However, there are significant differences, such as the initialization of the computation by the trigger machine, and the persistence of the tape and private relations of the trigger machine between turns in the computation. These features are needed to carry out the simulations in Theorem 4.1.

Theorem 4.1 shows that the rather complex execution models of active database prototypes can be simulated by the very straightforward execution model of trigger machines. Since relational machines are equivalent to while_N (see Section 2), it follows that this can be used as a trigger specification language that can simulate the above prototypes (the integer variables, as well as temporary relations are persistent between computation turns). Thus, while_N can be used as a yardstick for comparing various active database prototypes. Indeed, each prototype corresponds to a fragment of while_N . As we shall see, most prototypes can in fact be simulated by *while* alone.

5 Expressive Power

The update languages considered earlier can be used to specify either external programs or trigger programs. It is easy to see that they can all be viewed as special classes of external machines, resp. trigger machines. Therefore, in the following we blur the distinction between programs written in these languages and the corresponding trigger machines. Now suppose that the active database works with a specific update language

for external programs. Then it makes sense to compare two trigger systems relative to the external machines corresponding to the given update language, rather than all possible external machines. This leads to the following. Two trigger machines M_t and M'_t are equivalent relative to an external language L , denoted $M_t \equiv_L M'_t$, if $M_t(M_e) = M'_t(M_e)$ for all $M_e \in L$.

Remark: There are subtle differences between equivalence of two programs M_t and M'_t as update programs and as trigger programs:

- M_t and M'_t may be equivalent as update programs without being equivalent as trigger programs. For example, let M_t be the $while_N$ program $R := R \cup \{t\}$ (insert t into R) and M'_t the program

$$\begin{array}{l} \text{if } i = 1 \text{ then } R := \emptyset \\ \text{else begin } R := R \cup \{t\}; i := 1 \text{ end} \end{array}$$

Clearly, M_t and M'_t are equivalent as update programs (since i is initially zero). They are not equivalent as trigger programs. Indeed, for the external program $M_e = \{R := R - \{t\}\}$, $M_t(M_e)$ inserts t into R whereas $M'_t(M_e)$ assigns \emptyset to R (because M_e causes M'_t to execute more than once, and i is persistent between these executions).

- M_t and M'_t may be equivalent as trigger programs without being equivalent as update programs. This uses the fact that trigger programs are executed at least twice according to our model: before and after the external program. As an example (similar to the above), let M_t be the $while_N$ program $R := \{t\}$ and M'_t the program

$$\text{if } i = 0 \text{ then } R := \emptyset \text{ else begin } R := \{t\}; i := 1 \text{ end}$$

Clearly, M_t and M'_t are not equivalent as update programs over R (recall that i is initially zero). They are however equivalent as trigger programs over R . Indeed, for any external program M_e over R , the final value of R is $\{t\}$ in both $M_t(M_e)$ and $M'_t(M_e)$.

Thus, one should be cautious about assuming that known equivalence results about query and update programs and languages extend to triggers, and conversely. \square

Similar remarks apply to external programs. The effect of an external program in the context of a trigger program is very sensitive to the placement of triggering states in the external program. In particular, two external programs equivalent as update programs may have radically different effects as external programs in an active database. As a very simple example, the program $R := R \cup \{t\}$ causes two executions of the trigger program (initial and final), whereas the program

$R := R - \{t\}; R := R \cup \{t\}$, equivalent as an update program to the first, causes three executions of the trigger program.

Going one step further, we are interested in comparing *languages* for trigger programs. A trigger language T *subsumes* another trigger language T' if for each $M'_t \in T'$ there exists $M_t \in T$ such that $M_t \equiv M'_t$. T and T' are *equivalent* if T subsumes T' and T' subsumes T . Clearly, one can relativize these definitions to given languages for external programs, as done earlier. This gives rise to the notions of subsumption and equivalence of trigger languages relative to a given external language.

Given a trigger program M_t and an external language E , we denote

$$M_t(E) = \{M_t(M_e) \mid M_e \in E\}.$$

Thus, $M_t(E)$ represents all updates expressible by external programs written in the language E in the context of the trigger program M_t . Of particular interest is the case when $M_t(E) \subseteq E$. Intuitively, this says that the semantics of any program in E can still be specified within E after it is transformed by the trigger M_t . In this case, we say that the language E is *closed* under M_t .

The above definitions can be extended from a single trigger M_t to a trigger language T . We denote

$$T(E) = \{M_t(M_e) \mid M_t \in T, M_e \in E\}.$$

The set $T(E)$ represents the combined expressive power of all external programs in E in conjunction with all trigger programs in T .

The following provides $T(E)$ for the main trigger languages T and external languages E . We consider the external languages FO^* , $RA(\text{Datalog})$, $RA(\text{Datalog}^\neg)$, $fixpoint$, $RA(\text{Datalog}^{\neg\neg})$, $while$, and the trigger languages FO^* , Datalog , $fixpoint$ (or equivalently Datalog^\neg) and $while$ (or equivalently $\text{Datalog}^{\neg\neg}$).

Theorem 5.1 .

- $FO^*(FO^*) = FO^*$;
 $FO^*(E) = while$ for $E \in \{RA(\text{Datalog}), RA(\text{Datalog}^\neg), fixpoint, RA(\text{Datalog}^{\neg\neg}), while\}$.
- $\text{Datalog}(FO^*) = \text{stratified-Datalog}^\neg$;
 $\text{Datalog}(RA(\text{Datalog})) = RA(\text{Datalog})$;
 $\text{Datalog}(E) = fixpoint$ for $E \in \{RA(\text{Datalog}^\neg), fixpoint\}$;
 $\text{Datalog}(E) = while$ for $E \in \{RA(\text{Datalog}^{\neg\neg}), while\}$.
- $fixpoint(E) = fixpoint$ for $E \in \{FO^*, RA(\text{Datalog}), RA(\text{Datalog}^\neg), fixpoint\}$;
 $fixpoint(E) = while$ for $E \in \{RA(\text{Datalog}^{\neg\neg}), while\}$.
- $while(E) = while$ for $E \in \{FO^*, RA(\text{Datalog}), RA(\text{Datalog}^\neg), fixpoint, RA(\text{Datalog}^{\neg\neg}), while\}$.

Proof (sketch): To illustrate the technique of the proof, we show that $\text{FO}^*(\text{fixpoint}) = \text{while}$. Clearly, $\text{FO}^*(\text{fixpoint}) \subseteq \text{while}$. For the converse, consider a *while* program w . Let trace be a binary relation and f the *fixpoint* program obtained from w by replacing each assignment $R := \varphi$ by $R := R; \text{trace} := \{\{R, \varphi\}\}$ (by slight abuse, R and φ are treated here as domain elements). Now let t be the FO^* program

$$\bigvee (\text{if } \text{trace} = \{\{R, \varphi\}\} \text{ then } R := \varphi; \text{trace} := \emptyset$$

(the disjunction is over all statements $R := \varphi$ occurring in w). Clearly, $t(f)$ is equivalent to w . Note that different f and t are required for each w . \square

Observe that, if two trigger languages T, T' are equivalent relative to an external language E , then $T(E) = T'(E)$. The converse is false. For example, by Theorem 5.1 $\text{FO}^*(\text{RA}(\text{Datalog})) = \text{while}(\text{RA}(\text{Datalog})) = \text{while}$. However, it is clear that $\text{FO}^* \not\equiv \text{RA}(\text{Datalog})$ *while*.

Like before, if $T(E) \subseteq E$ we say that E is closed under T . Intuitively, this means that the semantics of programs in E transformed by trigger programs in T can still be specified within E . This may have practical benefits, since the external program in E and trigger program in T can be compiled into another program in E and evaluated using resources developed for E .

For example, *while* is closed under *while*; $\text{RA}(\text{Datalog})$ is *not* closed under FO^* .

Prototypes vs. abstract languages

We next relate several prototypes to the languages discussed above. It turns out that the language *while* plays an important role. Indeed, we have:

Theorem 5.2 The trigger languages of ARDL, Postgres², Starburst, and Sybase are subsumed by *while*.

Recall that *while* is equivalent to $\text{Datalog}^{\neg, \neg}$, a rule-based language whose syntax is closer to that of the prototypes discussed above. Thus, the elaborate execution models of these prototypes can be simulated using this straightforward language. This shows that many of the constructs used in these prototypes have no impact on expressive power; such constructs may of course be needed to help manage the task of building complex rule systems. Theorem 5.2 provides information allowing to distinguish between the two situations.

Generally, *while* and $\text{Datalog}^{\neg, \neg}$ are more powerful than the prototypes they subsume. Precise equivalence is obtained with various modifications to the semantics of our trigger programs, described next. In all cases we need the assumption that a trigger program is

²Recall that the semantics of Postgres is only defined in our context on ordered databases, see Appendix.

run in a triggering state only if there has been a change in some database relation since the last turn. With this assumption, *while* is equivalent to ARDL. Supplementary assumptions are needed for Postgres, Starburst, and Sybase:

- Recall that Postgres and Sybase do not have deferred rules. If the final run of trigger programs at the end of external programs is omitted, *while* becomes equivalent to Sybase, and to Postgres on ordered databases. This uses the fact that Postgres allows recursive triggering of rules, and that Sybase, although not allowing recursive triggering, provides an iterative construct in actions of rules.
- Starburst allows *only* deferred rules. If the trigger program is only run, after the initial run, at the end of the external program (i.e., the external program has no intermediate triggering states), then *while* becomes equivalent to Starburst.

HiPAC is a case apart, because of its ability to trigger multiple occurrences of the same rule, and the ability of triggered rules to access previous database states via their delta relations. It is easily seen that HiPAC subsumes *while*. It turns out that HiPAC is strictly stronger than *while*. In fact, we show:

Theorem 5.3 HiPAC is equivalent to while_N .

Proof (sketch): The fact that HiPAC is subsumed by while_N follows from Theorem 4.1. The converse requires a simulation of while_N by HiPAC. The first-order assignments of while_N are easily simulated. The simulation of arithmetic is based on the fact that HiPAC allows an unbounded number of multiple occurrences of the same rule in the execution queue. These can be used as counters. More specifically, the simulation uses several rules triggered in deferred mode. Each triggered rule is associated with an integer variable i of the while_N program. This is done using a special unary relation VAR that holds, at the time of triggering, the name of the integer variable. The value of a variable i at any given point is given by the number of occurrences in the current execution queue of rules $r(i)$ for which VAR contained i at the time of triggering. Increments and decrements of i in the while_N program are simulated by firing or executing occurrences of $r(i)$. Tests $i > 0$ in the while_N program are done by determining if some $r(i)$ occurs in the queue. The increments, decrements, and tests require scanning the queue and reaching certain occurrences of rules. This is achieved by performing rotations of the execution queue; rules are executed and triggered again until the desired rotation is achieved. \square

Recall that while_N is equivalent to the full relational machine. Thus, HiPAC is also equivalent to it. In particular, $\text{HiPAC}(\text{FO}^*) = \text{while}_N(\text{FO}^*) = \text{while}_N$.

Since $while_N$ expresses all queries on ordered databases, so does $\text{HiPAC}(\text{FO}^*)$.

Prototypes vs. prototypes

The above results comparing prototypes to abstract languages show that each prototype corresponds to some fragment of $while_N$. This allows to use $while_N$ as a yardstick for comparing the expressive power of the prototypes. Indeed, we can show:

- Postgres and Sybase are equivalent (on ordered databases);
- Starburst is incomparable to Postgres and Sybase;
- ARDL strictly subsumes Postgres, Starburst, and Sybase;
- ARDL using just immediate firing is equivalent to Sybase, and to Postgres on ordered databases.
- ARDL using just deferred firing is equivalent to Starburst;
- HiPAC strictly subsumes ARDL.

Recall that these statements hold subject to the simplifying assumptions we have made about the prototypes, and to our overall framework for comparing trigger systems. They may be quite sensitive to changes in either. As an example, suppose that external machines can signal to the trigger program that they have reached a final state, via a particular database relation used as a flag. Then the deferred triggers such as those of Starburst can be simulated with immediate triggers such as those of Postgres and Sybase, and it follows that Postgres and Sybase subsume Starburst. One can also show that Sybase is equivalent to ARDL, and Postgres is equivalent to ARDL on ordered databases.

We next focus in more detail on HiPAC, which is of particular interest because it provides a wide diversity of features that interact in intricate ways. Also, HiPAC subsumes all the other prototypes we have considered. It is of interest to consider the impact of various features of HiPAC on its power as a trigger language. We briefly discuss a few variations next. The proof of the equivalence of HiPAC and $while_N$ (Theorem 5.3) uses the fact that in HiPAC the execution queue may contain multiple occurrences of the same rule. Note that the ability of triggers to access their triggering states is used in a very limited way in the proof. If boolean parameters could be passed to the rules at the time of triggering, then accessing the triggering state would become unnecessary (as long as a fixed set of private relations could be used for bookkeeping). This also becomes unnecessary if the execution model is slightly modified so that deferred rules are executed in order of triggering, regardless of priorities among triggers.

Indeed, in this case we can use in the proof different triggers associated with each integer variable of the $while_N$ program.

If multiple occurrences of the same rule are disallowed, then HiPAC is clearly subsumed by $while$, so there is a dramatic loss of power. As an intermediate possibility, suppose that rules can access their triggering states, but multiple occurrences of rules triggered by equal states are disallowed. Denote this variant of HiPAC by **HiPAC**. It turns out that **HiPAC** lies strictly between $while$ and $while_N$. Indeed, we have:

Theorem 5.4 **HiPAC** strictly subsumes $while$ and is strictly subsumed by $while_N$ (and is therefore strictly subsumed by HiPAC).

Proof (sketch): First note that on one hand, **HiPAC**(FO^*) can be evaluated in EXPSpace , because the execution queue cannot become more than exponentially long, and by our assumptions every single rule can be evaluated in FO^3 . On the other hand, $while_N(\text{FO}^*)$ has no complexity bound. This shows that **HiPAC** strictly subsumes $while$. To show that **HiPAC** strictly subsumes $while$ we prove that **HiPAC**(FO^*) can express EXPSpace -complete updates. This is sufficient, since $while(\text{FO}^*)$ is within PSPACE and $\text{PSPACE} \neq \text{EXPSpace}$ by the space hierarchy theorem. The proof that **HiPAC**(FO^*) expresses EXPSpace -complete updates involves a simulation of a Turing Machine using exponentially many tape cells. The tape is represented by associating to each cell a triggered rule that awaits execution. The past database state accessed by each triggered rule identifies the cell and provides its content. This uses a relation providing a counter that can count up to an exponential, thus identifying exponentially many cells. As a consequence of this result, one can also show that **HiPAC**(FO^*) expresses exactly the set of EXPSpace queries, assuming the input database is ordered. \square

6 Semantic Consistency of Triggers

When an external program is executed in conjunction with a trigger program, its effect is modified by the triggers. As mentioned in Section 3, the final effect is generally dependent on the syntax of the external program, not only its semantics. Consider for example the FO^* program

$$\begin{aligned} P & := Q \bowtie R \bowtie S; \\ V & := Q \bowtie S \bowtie T \end{aligned}$$

³This uses our assumption that actions are limited to tuple insertions and deletions. The full HiPAC allows arbitrary calls to external procedures, so is not included in EXPSpace .

This program is equivalent (as an update program) to the following:

$$\begin{aligned} P &:= Q \bowtie S; \\ V &:= P \bowtie T; \\ P &:= P \bowtie R; \end{aligned}$$

However, the final effects of the two programs under a given trigger program are generally different. This can be unpleasant, since there are natural situations where program transformations are performed by the user or compiler, for optimization or other purposes. In this particular case, the transformation of the first program to the second would be natural for an optimizer detecting common subexpressions. The property that the final semantics of an external program under a trigger program is well defined as a function of the original semantics is therefore a desirable one. This is formalized by the following notion (referring back to the machine model of Section 4). A trigger machine M_t is *semantically consistent* if $M_t(M_e) = M_t(M'_e)$ for all equivalent external machines M_e and M'_e . As one might expect, semantic consistency is undecidable.

Theorem 6.1 It is undecidable whether a given trigger machine M_t is semantically consistent.

Proof (sketch): We use reduction of validity of FO sentences to semantic consistency of a trigger program. Let R be a relation and $\varphi(R)$ an FO sentence over R . Then $\varphi(R)$ is valid (true for all R) iff the following FO* trigger program over R is semantically consistent:

$$R := \{t \mid (\neg flag \wedge R(t)) \vee (flag \wedge R(t) \wedge \varphi(R))\}; flag := \mathbf{true}$$

where $flag$ is boolean. \square

One can relativize the notion of semantic consistency to languages for external and trigger programs in the natural way. The proof of Theorem 6.1 shows that semantic consistency of FO* trigger programs with respect to FO* external programs is undecidable (so this remains undecidable for all stronger languages, including the trigger languages of the prototypes). Nonetheless, it is interesting to note that semantic consistency always holds for some classes of languages. For example, let $RA^*(\text{Datalog}) = \{e^* \mid e \in RA(\text{Datalog}) \text{ and } \text{' ; ' occurs in } e\}$.

Proposition 6.1 Datalog trigger programs are semantically consistent relative to $RA^*(\text{Datalog})$ external programs.

The proof uses the monotonicity of Datalog.

The fact that semantic consistency does not generally hold motivated our use of trigger states in our machine model for external programs, and our assumption that external languages working with triggers provide first-class constructs for passing control to the trigger system. When such constructs are present, it seems natural to

relax the requirement of semantic consistency by taking into account the trigger states. This is based on a stronger notion of equivalence of external machines. Intuitively, it requires that the machines behave the same way not just with respect to their start and final states, but also relative to their intermediate trigger states. More precisely, let M and M' be two external machines such that $sch(M) = sch(M') = \mathbf{R}$ and S, S' be their respective sets of start, triggering, and final states. For each pair of states $x, y \in S$, let $M_{\langle x, y \rangle}$ be the machine obtained from M by having the start state be x and the final state be y (and similarly for M'). M and M' are *trigger equivalent* if there exists a bijection f from S to S' such that:

1. s is the start state of M iff $f(s)$ is the start state of M' ;
2. h is a final state of M iff $f(h)$ is a final state of M' ;
3. t is a triggering state of M iff $f(t)$ is a triggering state of M' ; and,
4. for every pair of states $x, y \in S$, $M_{\langle x, y \rangle}$ is uniformly equivalent to $M'_{\langle f(x), f(y) \rangle}$, that is: for \mathbf{I} over \mathbf{R} , $M_{\langle x, y \rangle}$ halts in state y with output \mathbf{J} over \mathbf{R} iff $M'_{\langle f(x), f(y) \rangle}$ halts in state $f(y)$ with output \mathbf{J} over \mathbf{R} .

Clearly, trigger equivalence of external machines implies equivalence. Next, a trigger machine M_t is *weakly semantically consistent* if $M_t(M_e) = M_t(M'_e)$ for every M_e and M'_e that are trigger equivalent. The following is shown directly from the definition of the semantics of trigger machines:

Fact: Every trigger machine is weakly semantically consistent.

Intuitively, weak semantic consistency indicates that external programs can be transformed according to usual transformation rules as long as the affected portions of code do not go across trigger invocation boundaries.

7 Conclusions

We introduced the relational machine model for active databases and used it as a framework to formally articulate some basic concepts relating to the semantics and expressiveness of active databases. The semantics emphasizes the interaction of external programs with the trigger system, and highlights the importance of two factors: the execution model of the trigger system and the discipline for transfer of control from the external program to the trigger system. For the expressiveness results, we considered both abstract languages and simplified versions of several active database prototypes. Two abstract languages emerge as central: *while* (with its rule-based variant $\text{Datalog}^{\neg\neg}$) subsumes most active

database prototypes. And, $while_N$ (or the relational machine) subsumes all prototypes we considered and is equivalent to HiPAC. Thus, abstract languages such as $while$ and $while_N$ can serve as a yardstick for comparing different active database systems and provide a tool for understanding the impact of various active database features on expressiveness.

The present paper is one step towards a formal study of active databases. Many issues remain unexplored, and some important aspects of active databases (such as syntax-based events, or nondeterministic semantics) are not covered by our framework. In relation to this paper, the impact on expressiveness of variations to the framework developed here and to the assumptions made on the prototypes needs to be better understood. Our results suggest that expressiveness is insensitive to many variations in execution models, but very sensitive to other aspects. As one example, expressiveness is sensitive to the way private, persistent relations are used by triggers. A modification allowing persistence across user transactions would have significant impact. In particular, it may allow incremental evaluation of queries using auxiliary information, as in [DT92], and the implementation of structural recursion constructs based on insertion, as in [BBN92].

Acknowledgement: We thank Richard Hull for discussions on Heraclitus, and Eric Simon for discussions on ARDL and other active database systems. Many thanks to Richard Hull, Eric Simon, and Jennifer Widom for very thorough and insightful comments on a draft of this paper.

References

- [AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [AV90] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41:181–229, 1990.
- [AV91a] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43:62–124, 1991.
- [AV91b] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 209–219, 1991.
- [AV95] S. Abiteboul and V. Vianu. Computing with first-order logic. *Journal of Computer and System Sciences*, 1995. To appear.
- [AWH92] A. Aiken, J. Widom and J.M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *Proc. ACM-SIGMOD Int'l. Conf. on Management of Data*, pages 59–68, 1992.
- [BM91] C. Beeri and T. Milo. A model for active object oriented databases. In *Proc. of Int'l. Conf. on Very Large Data Bases*, pages 337–349, 1991.
- [BBN92] V. Breazu-Tannen, P. Buneman and S. Naqvi. Structural Recursion as a Query Language. In *Proc. of Int'l. Workshop on Database Programming Languages*, pages 9–19, Morgan Kaufmann, 1992.
- [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 Symp. 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.
- [Coh86] D. Cohen. Programming by specification and annotation. In *Proc. of AAAI*, 1986.
- [CTF88] M. A. Casanova, L. Turcherman, and A. L. Furtado. Enforcing inclusion dependencies and referential integrity. In *Proc. of Int'l. Conf. on Very Large Data Bases*, pages 38–49, 1988.
- [CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. of Int'l. Conf. on Very Large Data Bases*, 1990.
- [CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of Int'l. Conf. on Very Large Data Bases*, pages 577–589, 1991.
- [D+88] U. Dayal et al. The HiPac project: Combining active databases and timing constraints. In *ACM SIGMOD Record*, 1988.
- [Don88] G. Dong. On the composition and decomposition of datalog program mappings. In *Proc. of Int'l. Conf. on Database Theory*, pages 87–101, 1988.

- [DT92] G. Dong and R. Topor. Incremental evaluation of datalog queries. In *Proc. of Int'l. Conf. on Database Theory*, pages 282–296, 1992.
- [Han89] E.H. Hanson. An initial report on the design of ariel: a dbms with an integrated production rule system. In *Sigmod Record*, 18(3), pages 12–19, 1989.
- [HW92] E. N. Hanson and J. Widom. An overview of production rules in database systems. Technical Report RJ 9023 (80483), IBM Almaden Research, October 1992.
- [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 Int'l. 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.
- [HY84] R. Hull and C. K. Yap. The Format model: A theory of database organization. *Journal of the ACM*, 31(3):518–537, 1984.
- [IN88] T. Imielinski and S. Naqvi. Explicit control of logic programs through rule algebra. In *Proc. ACM Symp. on Principles of Database Systems*, pages 103–116, 1988.
- [Kan91] P. C. Kanellakis. Elements of relational database theory. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1074–1156. Elsevier, 1991.
- [MD89] D. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 215–224, 1989.
- [Mor83] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Proc. of Int'l. Conf. on Very Large Data Bases*, pages 34–42, 1983.
- [Pic] P. Picouet. *Puissance d'expression et Consistance sémantique de systèmes de triggers*. PhD thesis, Ecole Nationale Supérieure de Télécommunications, Paris. In preparation.
- [SKdM92] E. Simon, J. Kiernan, and C. de Maindreville. Implementing high level active rules on top of a relational dbms. In *Proc. of Int'l. 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.
- [SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 281–290, 1990.
- [Sto92] M. Stonebraker. The integration of rule systems and database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4:415–423, 1992.
- [Syb87] Sybase, Inc. Transact-sql user's guide. Technical report.
- [Ull88] J.D. Ullman. *Principles of Database and Knowledge Base Systems, Volume I*. Computer Science Press, 1988.
- [WF90] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proc. ACM SIGMOD Symp. 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.
- [WCD95] J. Widom, S. Ceri and U. Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, Inc., San Francisco, California. To appear, 1995.
- [WCL91] J. Widom, R.J. Cochrane and B.G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In *Proc. Conf. on Very Large Data Bases*, Barcelona, 1991.

8 Appendix

We briefly highlight below some of the main features of the active database prototypes considered in this paper. These reflect our simplifying assumptions described in Section 3. Further descriptions can be found in [Pic], based on a preliminary version of [WCD95].

- **ARDL**: rules are triggered after each external SQL update on the database. Conditions are first-order tests involving the current database states and delta-relations providing the changes from the start of the external transaction to the current time. Actions are tuple insertions and deletions. The coupling mode between triggering event and rule execution may be immediate or deferred. The execution order of several rules awaiting execution can be specified using expressions in a rule algebra.
- **HiPAC**: HiPAC is perhaps the richest of the active database prototypes in terms of variety of features and flexibility. We consider here a relational version of HiPAC (as opposed to its object-oriented version). The execution model is based on a nested transaction model. Each atomic step in the external program is a subtransaction. Rules are triggered at the end of each subtransaction. Events associated with each trigger consist of updates on a particular *signal* relation. Composite events on the signal relation can be specified by regular expressions. Conditions are first-order tests involving the signal relation of the trigger as well as delta relations providing the changes to the signal relation from the *time of triggering* to the current state. An action is a procedure call, which we assume here to be a first-order update defined using the current state and the delta relations. Coupling modes between triggering event and condition checking and between condition checking and execution of the action can be specified as immediate, deferred, or decoupled. A rule triggered in immediate mode is executed as a subtransaction right after the triggering subtransaction. A rule triggered in deferred mode is executed as a new child of the root of the triggering transaction. Decoupled triggers are in principle executed concurrently; however, we assume here a serial execution based on a fixed predefined priority among the rules. HiPAC allows several occurrences of the same rule in the current execution queue (each occurrence uses its own delta relations). Rules are ordered in the execution queue by a criterion taking into account both priority and the time of triggering.
- **Postgres**: Postgres provides a rule system PRS2 that comes in two variants: a semantic-oriented one (Tuple Level System, TLS) and a syntactic-oriented one (Query Rewrite System, QRS). We

only consider here the semantic variant TLS. In TLS, events are tuple-oriented: they consist of insertions, deletions, or updates of tuples from a table (we consider only insertions and deletions here). Applicable rules are triggered immediately after a tuple is accessed, even if the user command is set-oriented. This may lead to nondeterminism. We use here a deterministic variant by assuming a fixed order between tuples, and considering the semantics resulting from accessing tuples in that order. Consequently, we assume Postgres only operates on ordered databases. Our framework restricts Postgres to only react to tuple insertions and deletions when control is passed to it in a triggering state of the external program. Conditions are comparisons of attribute values from the current tuple. Because of its tuple oriented-semantics, Postgres provides a tuple oriented delta relation using tuple variables *new* and *old*. Actions consist of SQL updates. There is no bound on nested triggering of rules.

- **Starburst**: rules are first triggered at the end of the external transaction. For each trigger, the triggering event consists of an update of a single database relation. Rule executions can trigger other rules. The coupling mode between triggering event and rule execution is deferred. Conditions and actions are first-order queries involving the current database state and delta relations. The delta relations provide the cumulative changes since the beginning of the transition that last triggered the rule. The execution queue does not contain multiple occurrences of the same rule. Rules are totally ordered in the execution queue by priorities that are either explicitly defined or the result of a default tie-breaking algorithm.
- **Sybase**: rules are triggered by external SQL commands. Events associated to each rule consists of updates on a single relation. We make the important assumption that events are semantic (otherwise, Sybase is incomparable to the other prototypes considered here). The coupling mode between triggering event and rule execution is always immediate. Actions consist of SQL updates involving the current database state and using the delta relations providing the changes to the triggering relation by the triggering SQL update. The SQL updates may be iterated using a *while* construct. The action of a trigger can itself trigger another rule; however there is a fixed bound on the number of rules consecutively triggered. In particular, a trigger cannot trigger itself.