Semantics and Expressiveness Issues in Active Databases

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. This allows to formally compare the expressiveness of the prototypes. 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

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 [WC95]). However, foundational work in this area is still scarce (e.g., see [AWH92, BM91, HJ91b]). 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

*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.

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 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 [AV91a, 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). This suggests that the relational machine model provides a convenient unifying formalism for investigating a variety of active database questions. In particular, it can be used as a yardstick against which various trigger systems can be measured.

Although very useful for formalizing the basic paradigm of active databases, the abstract 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. Therefore, we introduce a formal model which closely captures the programming style of the prototypes. Using this as a common framework, we specify formally the prototypes considered here, subject to certain simplifications and unifying assumptions (e.g.,, the model is relational, the semantics is deterministic, etc.). We next go on to establish results on the complexity and on the relative expressiveness of the prototypes (see Figure 1). 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, [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. [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 reviews basic concepts relating to active databases, discusses the connection between trig-

gers and external programs, and introduces the model of active databases based on relational machines. Expressiveness results concerning the abstract model are presented in Section 4, including characterizations of the power of several major languages for triggers relative to various languages for external programs. The formal specifications of the prototypes is provided in Section 5. Section 6 establishes the results on the expressiveness and complexity of the prototypes. First, it is shown that relational machines subsume the trigger systems of ARDL, HiPAC, Postgres, Starburst, and Sybase. Next, the power of various features of the prototypes is investigated. The discussion is centered around immediate triggering, deferred triggering, and their interaction. Finally, the relative expressiveness and complexity of the prototypes is established. Section 7 briefly discusses semantic consistency of triggers, and Section 8 provides conclusions.

2 Preliminaries

We briefly review several relational update languages, and an abstract model of database computation called *relational machine*. We assume familiarity with the terminology of relational databases, including relational calculus and algebra, as in [Ull88, Kan91, AHV95]. Relations are denoted here by $P, Q, R, \ldots$, database schemas by **P,Q,R,...**, and the set of instances over a schema **R** by *Inst*(**R**). The first-order queries (defined by the calculus and algebra) are denoted FO.

Updates and update languages

An *update* over a database schema **R** is a mapping from Inst(**R**) to Inst(**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* and **FO**$^+$. **FO*** 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). Some languages use assignment statements with cumulative semantics, denoted $R \mathrel{+}= \varphi$. The semantics is that the result of φ as *added* to the content of R. The set of programs obtained by compositions of cumulative assignment statements is denoted $\mathbf{FO}^+$.

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 that assignment statements are in FO^+. This guarantees termination of *fixpoint* programs in PTIME, whereas *while* programs require PSPACE.

Datalog, Datalog$^\neg$, and Datalog$^{\neg\neg}$. A Datalog program is a set of rules of the form

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

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 $A_1(t_1), \ldots, A_n(t_n)$ 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$^\neg$ 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$^\neg$, 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$^\neg$ is evaluated just like Datalog, in consecutive stages of firings of rules. Finally, Datalog$^{\neg\neg}$ 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 [AV91b] that Datalog$^\neg$ is equivalent to *fixpoint* and Datalog$^{\neg\neg}$ is equivalent to *while*.

Semi-positive Datalog$^\neg$ programs are Datalog$^\neg$ programs that only use negation on relations not occurring in heads of rules. The semantics is the obvious one: $\neg R(\mathbf{x})$ is true if $\mathbf{x}$ is not in the input relation R. Stratified-Datalog$^\neg$ extends semi-positive Datalog as follows. Each stratified-Datalog$^\neg$ program is a composition $P_1, \ldots, P_n$ of semi-positive programs P_i (called *strata*) such that for each relation R all rules where R occurs in the head

belong to one stratum, and each R occurring negatively in P_i is defined in some previous stratum $P_j, j < i$. It is known that stratified-Datalog$^\neg$ is strictly less expressive than *fixpoint* (and therefore Datalog$^\neg$) [K91].

Relational machines

The *fixpoint* and *while* queries can be viewed as programming languages with simple control interacting with the databases via a fixed set of FO queries. A natural extension of this paradigm is to consider a computationally complete computing device interacting with the database via given FO queries. This captures the spirit of external programs, which often use an FO language (say, SQL) embedded in a full programming language (say, C). It is formally captured by a computational model called "relational machine".

A *relational machine* (RM) is a Turing machine augmented with a *relational store*. The relational store consists of a set of relations of certain arities. Some of these relations are designated as input relations and some are designated as output relations. 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 a relation in the store is empty and assign to a relation in the store the result of an FO query. For example, the machine can have instructions such as:

> "If the machine is in state s_3, the head is reading the symbol 1, and relation R_1 is empty, then change the state to s_4, replace the symbol 1 by 0, move the head to the right, and replace R_2 by $\{x, y \mid \exists z(R_2(x, z) \wedge R_3(z, y))\}$".

If the machine halts, the output can be found in designated relations in the store.

It is easy to see that this generalizes most query languages, including *fixpoint* and *while*.

Note that a relational machine uses a fixed set of queries; the *arity* of a relational machine is the maximum number of variables used in its FO queries. The schema of the relational store of an RM M is denoted by $sch(M)$.

Relational machines express all computable queries, assuming the database is ordered. On arbitrary databases, relational machines are *not* complete; indeed, they collapse to *FO* on unary relation inputs and so cannot express "simple" queries such as checking evenness of the cardinality of the input. As discussed in [AV91a, AV95], this behavior is due to the fact that relational

machines using some constant k number of variables have limited discerning power: given an input, different tuples may not be distinguishable using k variables. Thus, the machine really manipulates classes of the equivalence relation on k- tuples: $u \equiv_k v$ on input $\mathbf{I}$ iff u and v cannot be distinguished by any relational machine with k variables running on input $\mathbf{I}$. As implicit in [IL90, AV91a] and shown in [AV95, KV92], $\equiv_k$ can be computed by a *fixpoint* query for each k. Moreover, the *fixpoint* query outputs the classes of $\equiv_k$ in some order. This allows reducing computations over unordered inputs to ordered inputs, and yields the following normal form for relational machines, which provides a key technical tool: For each relational machine M, there exists an equivalent machine which works in two phases. The first phase computes $\equiv_k$ and a set of *summary tables* describing the action of first-order queries with k variables on the classes of $\equiv_k$. For an input $\mathbf{I}$, let $summary_k(\mathbf{I})$ be the set of summary tables obtained for $\mathbf{I}$. To conclude the first phase, the summary tables are coded on the tape using integers to represent the equivalence classes. (The entire first phase is done in PTIME.) Next, the computation is carried out exclusively on the tape, using the information provided by $summary_k(\mathbf{I})$. The content of a relation is represented at all times by a sequence of integers representing the equivalence classes it contains.

For Turing machines, the most natural measure of complexity is in terms of the size of the input. This is no longer so for relational machines, since such machines cannot measure the size of their input. Instead, relational machines are sensitive to the number of classes of $\equiv_k$ on input $\mathbf{I}$, called the *k-size* of $\mathbf{I}$. To see the difference between k-size and regular size, note that the k-size of sets is constant, that of complete binary trees is poly-logarith mic in their size, and for ordered structures it is polynomial in the size. This explains the fact that relational machines collapse to first-order on sets but are complete on ordered inputs. This also suggests that the right measure of the input for a relational machine with k variables is the k-size of the input. The use of k-size as a measure of the input for a relational machine gives rise to a new notion of computational complexity, called *relational complexity*, resulting in classes such as PTIME$_r$ (relational polynomial time), and PSPACE$_r$. It is shown in [AV91a] that PTIME$_r$ = *fixpoint* and PSPACE$_r$ = *while*, generalizing the results of [Im86, Var82] that PTIME = *fixpoint* and PSPACE = *while*, *on ordered inputs.*

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 $i, j, ...$, initialized to zero; (ii) *increment(i)* and *decre-*

ment(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* [AV91a].

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

In this section we provide a brief, high-level review of active databases and an abstract model for active databases.

3.1 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:

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

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 $t = c_1; \ldots; c_n$ 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 *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 $d_1; \ldots; d_m$ is triggered in state $\mathbf{I}_1$ of the above sequence of states. Then the sequence of databases states might start with

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

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

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

where now $\mathbf{I}^{orig}$ is the original state, $\mathbf{I}^{user}$ is the result of applying the user-requested transaction, and the states $\mathbf{I}_2, \mathbf{I}_3, \ldots, \mathbf{I}^{curr}$ 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 [HJ91a, HW92, Sto92], each uses a different execution model.

We will provide in Section 5 formal descriptions of the execution models of several trigger systems. To allow meaningful comparisons among the various trigger systems, we will make certain unifying assumptions, outlined next.

3.2 Unifying assumptions

There are several major active database prototypes. In this paper we refer to the following: ARDL, HiPAC, Postgres, Starburst, and Sybase[1]. Unfortunately, all prototypes are in a state of flux, and many versions of each are

[1] Sybase is actually a commercial product.

described in the literature. In order to be able to make precise statements about them, we will provide in Section 5 formal descriptions of these systems, based mostly on their presentation in [WC95]. In order to be able to compare the different prototypes, we 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.

- Triggers have access to database relations, as well as to private relations used for bookkeeping (such as storing deltas relations). 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 are programs causing insertions and deletions of sets of tuples into relations; these are defined by FO (SQL) queries 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 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.

3.3 An abstract model for active databases

We now present a model for active databases based on relational machines. It is a simple model that provides a very basic formal framework for active databases. As we shall see in Section 6, its main justification is that the model subsumes all prototypes considered.

A major characteristic of the active database paradigm is that triggers work in conjunction with external programs, and the semantics is the result of their interaction. The abstract model we propose is meant to capture the essential aspects of this mechanism. We begin with some remarks on the issues involved.

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 looses 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]). This is captured in the abstract model by "triggering states" present in external programs.

Specifically, let $\mathbf{R}$ be a database schema. An *external machine* over $\mathbf{R}$ is a relational machine M, $\mathbf{R} \subseteq sch(M)$, 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 a trigger program (whenever they reach a triggering state).

Trigger programs are represented by "trigger machines", which are usual

relational machines (without triggering states). More precisely, a *trigger machine* over $\mathbf{R}$ is a relational machine M such that $\mathbf{R} \subseteq sch(M)$ where $sch(M)$ contains a distinguished 0-ary relation *external-done*. The distinguished relation *external-done* is used to signal to the trigger machine that the external machine has halted; it is set to *true* when a halting state is reached. This provides the only means of communication between M_e and M_t apart from the relations in $\mathbf{R}$. Like external machines, trigger machines model programs that interact with the database using SQL commands, with potentially arbitrary computing power on the side. The semantics of a pair $< M_e, M_t >$ 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 Inst($\mathbf{R}$) resulting from the following algorithm: (i) M_t is first 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 starting in its start state until a final state is reached, at which time 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 $sch(M_e)$ and $sch(M_t)$ 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 equivalence 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 above provides a very general model for active databases, focused on the interaction of external and trigger programs. In particular, relational machines provide reasonable upper bounds for the power of external or trigger programs ever likely to be used. This is confirmed by the fact that he model subsumes all prototypes considered in this paper, as shown in Section 6. However, it is often of interest to consider external and trigger languages which are less powerful than the full relational machines. Note that an external language can provide not only a computational restriction but also a specific discipline of transfer of control to the trigger program, determined by restrictions on which states can be triggering states. For example, one could define external languages where triggering states follow each FO interaction with the database, or where triggering states follow only nontrivial updates to the database, etc. We consider next several restricted languages for external programs and for trigger programs, and the power of their interaction.

4 Expressiveness in the Abstract

We wish to investigate the expressiveness of various trigger languages in conjunction with various external languages. The update languages discussed in Section 2 can essentially be used as both trigger and external languages, with a technical twist having to do with the transfer of control between external programs and trigger programs. As argued in Section 3.3, external programs must provide explicit transfer points to the trigger system if their semantics is to make any sense; this is modeled in the abstract model by triggering states of external machines. 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.

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 Datalog$^{(\neg)(\neg)}$ languages, we will use a variation of the *rule algebra* of [IN88]. The rule algebra RA(Datalog$^{(\neg)(\neg)}$) over the language Datalog$^{(\neg)(\neg)}$ is the smallest set of expressions containing the programs in Datalog$^{(\neg)(\neg)}$ such that

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

The semantics of a 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 RA(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 $Q = \{R(y, x) \leftarrow R(x, y)\}$.

Since the formal update languages 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 external or 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

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

  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. □

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* (or FO$^+$ when appropriate), RA(Datalog), RA(Datalog$^\neg$), *fixpoint*, RA(Datalog$^{\neg\neg}$), *while*, and the trigger languages FO*, Datalog, *fixpoint* (or equivalently Datalog$^\neg$) and *while* (or equivalently Datalog$^{\neg\neg}$).

In comparing two classes $\mathcal{C}, \mathcal{C}'$ of updates, we allow some flexibility in the schemas involved. Update programs defining an update over schema $\mathbf{R}$ are customarily allowed to use an extended schema including "temporary"

relations, initially empty. Similarly, an external and trigger program are sometimes allowed to use a common schema extending $\mathbf{R}$ (the additional relations are, again, initially empty).

Theorem 4.1 .

(i) $\text{FO}^*[\text{FO}^*] = \text{FO}^*$;
$\text{FO}^*[E] =$ *while* for $E \in \{\text{RA(Datalog)}, \text{RA(Datalog}^\neg), \textit{fixpoint}, \text{RA(Datalog}^{\neg\neg}), \textit{while}\}$.

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

(iii) $\textit{fixpoint}[E] =$ *fixpoint* for $E \in \{\text{FO}^+, \text{RA(Datalog)}, \text{RA(Datalog}^\neg), \textit{fixpoint}\}$;
$\textit{fixpoint}[E] =$ *while* for $E \in \{\text{RA(Datalog}^{\neg\neg}), \textit{while}\}$.

(iv) $\textit{while}[E] =$ *while* for $E \in \{\text{FO}^*, \text{RA(Datalog)}, \text{RA(Datalog}^\neg), \textit{fixpoint}, \text{RA(Datalog}^{\neg\neg}), \textit{while}\}$.

Proof: (i) $\text{FO}^*[\text{FO}^*] = \text{FO}^*$ is obvious. It is straightforward to see that $\text{FO}^*[E] \subseteq$ *while* for $E \in \{\text{RA(Datalog)}, \text{RA(Datalog}^\neg), \textit{fixpoint}, \text{RA(Datalog}^{\neg\neg}), \textit{while}\}$. For the converse, it is sufficient to show that $\textit{while} \subseteq \text{FO}^*[\text{RA(Datalog)}]$, since clearly $\text{FO}^*[\text{RA(Datalog)}] \subseteq \text{FO}^*[E]$ for $E \in \{\textit{RA(Datalog}^\neg), \textit{fixpoint}, \text{RA(Datalog}^{\neg\neg}), \textit{while}\}$. Consider a *while* update over a schema $\mathbf{R}$. By a well-known normal form for *while* (e.g., see [AHV95]), for each $R \in \mathbf{R}$ there exists a *while* program defining the result of the update for R, of the form:

$S_R := \emptyset$;
while change do
    begin
    $S_R := \varphi_R(S_R)$;
    end
$R := \pi(\sigma(S_R))$.

where σ is a relational algebra selection and π is a projection. Let P be the RA(Datalog) program

$$(\{S_R \leftarrow S_R \mid R \in \mathbf{R}\};)^*$$

and f be the FO* program

$$(\Pi_{R\in\mathbf{R}}\ S_R := \varphi_R(S_R)); (\Pi_{R\in\mathbf{R}} R := \text{ if } \textit{external-done} \text{ then } \pi(\sigma(S_R)) \text{ else } R).$$

where Π denotes composition. It is easy to verify that $P[f]$ defines an update whose restriction to $\mathbf{R}$ is w. Note that different P *and* f are required for each w.

(ii) The inclusion Datalog[FO$^+$] $\subseteq$ *fixpoint* is easy: each update in Datalog[FO$^+$] is the composition of FO$^+$ assigments $R += \varphi$ and of Datalog updates, each of which is also a *fixpoint* update. Consider the inclusion stratified-Datalog$^\neg$ $\subseteq$ Datalog[FO$^+$]. Let P be a stratified-Datalog$^\neg$ program with stratification $P_1; \ldots; P_n$, defining an update over schema $\mathbf{R}$. For each P_i, let D_i be the Datalog program obtained from P_i by replacing each negative literal $\neg R(\mathbf{x})$ by $\overline{R}(\mathbf{x})$ where $\overline{R}$ has the same arity as R. Let D be the union of the D_i, $1 \leq i \leq n$. Let e be the FO$^+$ program $e(P_1); \ldots; e(P_n)$ where $e(P_i)$ is the concatenation in arbitrary order of assignments $\overline{R}+ = \neg R$ where R is defined in stratum P_i. Let $\mathbf{R}' = \mathbf{R} \cup \textit{sch}(P)$. It is easily verified that $D[e]$ defines an update over $\mathbf{R}'$ whose restriction to $\mathbf{R}$ (when $\mathbf{R}'-\mathbf{R}$ is initialized to $\emptyset$) is the update defined by P.

The remaining statements in (ii) are immediate.

(iii). The inclusions *fixpoint*$[E] \subseteq$ *fixpoint* for $E \in \{$FO$^+$, RA(Datalog), RA(Datalog$^\neg$), *fixpoint*$\}$ are obvious. They follow from the fact that E is in all cases included in *fixpoint*, so any update in $fixpoint[E]$ can be written as a composition of *fixpoint* updates – itself a *fixpoint* update. For the converse, consider for instance the case when E is FO$^+$. To show that *fixpoint* $\subseteq$ *fixpoint*[FO$^+$], let f be an update in *fixpoint*. To simulate f, take a trivial FO$^+$ external program $e: \ R+ = R$ and a *fixpoint* trigger program $\overline{f}$ obtained by adding to f a flag (private relation) which inhibits f after its first execution:

$$\overline{f} = (\text{if } \neg flag \text{ then f}); flag += \text{true}.$$

Clearly, $\overline{f}[e]$ defines the same update as f.

The remainder of (iii) is obvious; (iv) is shown along the same lines as (iii). □

Remark: Several subtleties related to the difference between update and query languages arise in Theorem 4.1 and its proof:

- Consider the gap between Datalog[FO$^+$] and stratified-Datalog$^\neg$ in (ii). The stems from the fact that the language stratified-Datalog$^\neg$ is

treated here as a "pure" update language, which must explicitly define the updated relations. This prevents it from expressing FO^+ updates of the form $R \mathrel{+}= \varphi(R)$, because of the "recursive" definition of R (which may use negation). This problem would not arise if stratified-Datalog$^\neg$ were treated as a "query language turned update language", which may use a different name, say $newR$ to define implicitly the updated value of R. In this case equality would hold.

- For the same reason as above, all inflationary update languages can only define updates which are *increasing*; this is not the case if different names can be used for old and new versions of relations. □

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 4.1 $\mathrm{FO}^*[\mathrm{RA(Datalog)}] = \mathit{while}[\mathrm{RA(Datalog)}] = \mathit{while}$. However, it is clear that $\mathrm{FO}^* \not\equiv_{\mathrm{RA(Datalog)}} \mathit{while}$.

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*; RA(Datalog) is *not* closed under FO^*.

5 Prototype Snapshots

The abstract model of active databases based on relational machines presented in Section 3 provides a very high-level and simplified view of the computation carried out in active databases. We wish to relate this to the actual prototypes and then make formal statements about the expressive power and other aspects of the prototypes. To this end, we now embark upon relatively formal descriptions of the prototypes, providing "snapshots" based on their descriptions in [WC95], and subject to the unifying assumptions stated in Section 3.2. Formal statements made later in the paper about the prototypes are based upon these descriptions.

To facilitate the task, we first present a common framework which subsumes all the prototypes we consider. We then describe how the common framework is specialized to each of the prototypes.

5.1 Common framework

The 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.

Let $\mathbf{R}$ be a relational schema. We describe a trigger program P over $\mathbf{R}$. The program uses the relations in $\mathbf{R}$, and for each $R \in \mathbf{R}$, so called *delta relations* Δ_R^+ and Δ_R^- of the same arity as R. Among relations in $\mathbf{R}$, a subset $\mathbf{D}$ is distinguished as the (public) database schema. Relations in $\mathbf{R} - \mathbf{D}$ are sometime referred to as the "private" relations of P. Let $sch(P) = \mathbf{R}$, and

$$sch_\Delta(P) = \{\Delta_R^+, \Delta_R^- \mid R \in \mathbf{R}\}.$$

The delta relations in $sch_\Delta(P)$ contain incremental information among two database instances. The particular instances involved are determined in different ways in various contexts and systems.

A rule over $sch(P)$ is an expression of the form:

$$condition \rightarrow action$$

where *condition* is an FO sentence and *action* is an external program. Further restrictions are mentioned below.

A trigger program over $sch(P)$ is a 5-tuple $< Rules, cpl, ev, pri, \Delta\text{-}type >$ where

- $Rules$ is a set of rules over $sch(P)$.
- 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 $\{R^+, R^- \mid R \in sch(P)\}$ called the *event* mapping of P (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 P.
- Δ-*type* is a mapping from $Rules$ to $\{global, local\text{-}fixed, local\text{-}fluid\}$.

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 $sch(P) \cup sch(ev(r))$ and *action* is an external program over the same schema, which can access but not modify delta relations.

The meaning of the mappings *cpl, ev* and *pri* is quite intuitive. We elaborate briefly on the Δ-*type* mapping. 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 triggering state is reached (by the external program or some rule action). These delta relations can be accessed by rules as global variables, unless this is overriden by a local variable declaration as below. This type of delta relation is referred to 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. This mode of utilisation of delta relations is referred to 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 Δ-$type(r) = global$ 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 Δ-$type(r) \in \{local\text{-}fixed, local\text{-}fluid\}$ 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 Δ-$type(r) = local\text{-}fixed$, the value of the delta relation(s) passed as parameter does not change since the time the rule is triggered. If Δ-$type(r) = local\text{-}fluid$, 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 $sch(P)$ is a pair of database instances $\tau = (\mathbf{I}, \mathbf{J})$ over $sch(P)$. For $R \in sch(P)$, the delta relations associated with τ, denoted $\Delta_R^x(\tau)$ for $x \in \{+, -\}$, are defined as follows:

- $\Delta_R^+(\tau) = \mathbf{J}(R) - \mathbf{I}(R)$, 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].

It is easy to verify the following:

Fact 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 $sch(P)$. For each $R \in sch(P)$ and $x \in \{+, -\}$,

$$\Delta_R^x(\tau_2 \circ \tau_1) = \tau_2^x(\Delta_R^+(\tau_1), \Delta_R^-(\tau_1)).$$

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 common 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'.

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

1. E and P take turns as described in the abstract model; essentially, P takes control each time E reaches a triggering state. 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 a state of F, $\mathbf{I}$ a database instance, and q a queue of rules; its output is a triple $(s', \mathbf{I}', q')$ where s' is a new state of F, $\mathbf{I}'$ a new database instance, and q' a new queue of rules. The inputs of *exec-program* represent the current state in the execution of F, the current database instance, and the current q_{def}. The outputs represent the next trigger state in the execution of F on instance $\mathbf{I}$, the new database instance, and the new queue of deferred rules after the immediate queue has 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 $P[E]$ 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 $\textit{exec-program}(E, \textit{start}, \mathbf{I}, \epsilon)$. This either does not end – in which case the result is undefined – or it ends with output $(h, \mathbf{J}, q_{def})$, where h is a halting state of F. The output of the first phase is $(\mathbf{J}, q_{def})$.

2. execute the following pseudo-code program, where $(\mathbf{J}, q_{def})$ is the output of the first phase:

$q_{imm} := q_{def}$; $q_{def} := \epsilon$; $\mathbf{db} := \mathbf{J}$
while $q_{imm} \neq \epsilon$ **do**
begin
$(\mathbf{db}, q_{def}) := \textit{exec-imm}(\mathbf{db}, q_{imm}, \epsilon)$;
$q_{imm} := q_{def}$;
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 in state s, until a new triggering state t of F is reached. 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 $\Delta\text{-}type(r) \in \{local\text{-}fixed, local\text{-}fluid\}$, 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 $\textit{exec-imm}(\mathbf{I}', q_{imm}, q_{def})$, 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, from the current state t. If t is a halting state then the computation stops and *exec-program* returns $(t, \mathbf{J}, q'_{def})$. Otherwise, $exec\text{-}program(F, t, \mathbf{J}, q'_{def})$ is executed. This concludes the recursive definition of *exec-program*.

We now proceed with the description of *exec-imm*. On input $(\mathbf{I}, q_{imm}, q_{def})$, *exec-imm* does the following. If $q_{imm} = \epsilon$ then *exec-imm* stops and outputs $(\mathbf{I}, q_{def})$. Otherwise, let r be the first rule occurrence in q_{imm}; r is of the form

$$condition \quad \rightarrow \quad action$$

where *condition* is an FO sentence over $sch(P) \cup sch(ev(r))$ 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 $(h, \mathbf{J}, q''_{def})$ is produced, where h is a halting state of *action*. 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 apllied, and $q'_{def} = \tau(q_{def})$. New immediate and deferred queues are obtained according to the mappings $merge_{imm}$ and $merge_{def}$. Thus, *exec-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-imm* and *exec-program*, and also the description of the semantics of P.

5.2 Specializations

We describe here how the common framework is specialized to each of the prototypes. We begin with Starburst (which only allows deferred triggering), proceed with Postgres and Sybase (which only have immediate triggering), and end with HiPAC and ARDL (which allow both coupling modes). In each case we first summarize informally the characteristics of the prototype and then specify the corresponding restrictions of the common framework. This includes the specifications of the mappings *add*, $merge_{imm}$ and $merge_{def}$.

Recall that we use in all cases the unifying assumptions listed in Section 3.2.

Starburst

The main distinguishing feature of Starburst is that rules are always executed in deferred mode. Thus, a rule program P makes no modification to relations in $sch(P)$ until the external program has halted. However, the deferred queue evolves during the execution of the external program.

Each rule is triggered by an insertion or deletion to one relation. The queue contains only one occurrence of each rule; if an occurrence is already present, the new occurrence of the same rule is ignored. The delta relations provide the cumulative changes since the triggering of the rule (thus, all delta relations are of type *local-fluid*). Actions are restricted to a sequence of first-order updates involving the current database state and delta relations.

In terms of the common framework, the following restrictions apply:

- *Rules*: *action* is an FO* program.
- *cpl*: for each $r \in$ *Rules*, $cpl(r) =$ *def*.
- *ev*: for each $r \in$ *Rules*, $ev(r) \subseteq \{R^+, R^-\}$ for some R.
- Δ-*type*: for each $r \in$ *Rules*, Δ-*type*$(r) =$ *local-fluid*.
- $add(r, q)$ is defined as follows. If q contains some occurrence of r, then $add(r, q) = q$. Otherwise, $add(r, q)$ inserts r in q following the last occurrence of a rule r' such that $pri(r') < pri(r)$ (or at the head of q if no such rule exists).
- $merge_{imm}(q, q')$ returns the queue obtained by repeated applications of $add(r, q)$ for all occurrences r in q' (in any order). On the other hand, $merge_{def}(q, q') = \epsilon$. (This is because, during rule execution, new deferred rules are immediately added to the current queue of deferred rules – which is treated as an immediate queue in the second execution phase.)

Postgres

Postgres provides a rule system 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 (also implemented in Ingres).

In TLS, events are tuple-oriented: they consist of insertions, or deletions of tuples from a table. Applicable rules are triggered tuple-by-tuple, i.e.

immediately after a tuple is inserted/deleted, even if the user command appears to be set-oriented. We formalize this fact by assuming that Postgres operates only in conjunction with external programs that insert/delete tuples one at a time between triggering states. This is generally possible if the database is ordered (otherwise genericity is a problem). Indeed, it is easy to see that for any external machine M_e there exists an external machine M'_e defining the same update as M_e on ordered databases and which satisfies the above restriction. Consequently, we additionally assume Postgres only operates on ordered databases.

Aside from the tuple-at-a-time triggering described above, the main distinguishing feature of Postgres is that it only provides immediate triggering.

A rule action consists of a *while* external program satisfying the above restriction (i.e., operating one tuple at a time).

The common framework is specialized to Postgres as follows :

- *Rules*: rule actions consist of *while* external programs operating one tuple at a time.
- *cpl*: for each $r \in Rules$, $cpl(r) = \ imm$
- *ev*: *ev* is single-valued.
- Δ-*type*: for each $r \in Rules$, $\Delta\text{-}type(r) = local\text{-}fixed$.
- since there are no deferred rules, the q_{def} are always empty, so *add* is never used and $merge_{imm}(q, q')$ and $merge_{def}(q, q')$ are only used in these trivial cases: $merge_{imm}(q, \epsilon) = q$ and $merge_{def}(\epsilon, \epsilon) = \epsilon$.

Sybase

Like Postgres, Sybase provides only immediate triggering. Rules are simplified by the absence of conditions. The action is a *while* program with The delta relations of each triggered rule contain incremental information about the transition in which the rule was triggered.

In Sybase, rules are triggered by *syntactic* insertions and deletions to a relation. However, in accordance to our unifying assumptions, we ignore this aspect and suppose triggering is semantic in Sybase. Furthermore, we assume priorities among rules are given, to avoid the nondeterminism which might arise from several rules being triggered simultaneously. In fact, priorities are not used in Sybase. Note that our model is consistent with

Sybase for external programs that modify at most one relation between two triggering states.

A restriction peculiar to Sybase is that triggers can cascade but cannot recursively fire; that is, a rule cannot be triggered recursively at runtime by the action of the same rule previously triggered. This semantics cannot be directly modeled within the common framework. Indeed, it would require keeping a runtime history of the rules currently executed. However, it can be shown that each program as above (runtime nonrecursive) is equivalent to one which is syntactically nonrecursive, in the following sense. Consider a graph whose vertices are the rules of the program. There is an edge from p to q if the action of p contains an assignment to a relation R such that $R^x \in ev(r)$ for some $x \in \{+, -\}$. A trigger program is syntactically nonrecursive if the above graph is acyclic. This is a restriction which can be made within our framework.

These and further restrictions are summarized by the following specialization of the common framework:

- the trigger program is syntactically nonrecursive.
- *Rules*: there is no condition (i.e., *condition* = *true*) and *action* is a *while* program.
- *cpl*: for each $r \in$ *Rules*, $cpl(r) =$ *imm*.
- *ev*: for each r, $ev(r) \subseteq \{R^+, R^-\}$ for some R. Additionally, $ev(r) \cap ev(r') = \emptyset$ for $r \neq r'$.
- Δ-*type*: for each $r \in$ *Rules*, Δ-*type*(r) is *local-fixed*.
- $add(r, q)$, $merge_{imm}(q, q')$ and $merge_{def}(q, q')$: as for Postgres, these mappings are only trivially used, since there are no deferred rules in Sybase.

HiPAC

HiPAC is perhaps the richest of the active database prototypes in terms of variety of features and flexibility. As such, it also comes closest to the common framework. We first describe some aspects of the system, then specify how they are mapped to our snapshot within the common framework.

The snapshot is based on a relational version of HiPAC (as is done in [CBB+89], as opposed to its object-oriented version presented in [?]).

Events associated with each trigger consist of semantic updates of specified relations. Composite events on the signal relation can be specified by regular expressions (we do not include them explicitly in our snapshot, although they can be simulated). All types of delta relations are allowed (as suggested in [CBB+89]). Coupling modes between triggering event and condition checking and between condition checking and execution of the action can be specified as immediate or deferred[2]. The execution model in the system is based on a nested transaction model. Each atomic step (triggering state) in the external program is a subtransaction. Rules are triggered at the end of each subtransaction. Furthermore:

- 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.

HiPAC has a cycle-like semantics for executing the deferred rules, just as in the common framework: the initial cycle (called cycle 0) is composed of the user transaction and the immediate rules that have been triggered during this cycle. The first cycle (cycle 1) contains the deferred rules that have been triggered during cycle 0, and the immediate rules that have been triggered during cycle 1. Cycle $n (n \geq 1)$ is composed from deferred rules triggered in cycle $n-1$ and immediate rules triggered during cycle n. This discipline is implemented in the snapshot by appropriately specifying the *add*, $merge_{imm}$, and $merge_{def}$ mappings.

HiPAC allows several occurrences of the same rule in the current execution queues (each occurrence uses its own delta relations). Rules are ordered in the execution queue by a criterion taking into account both priority (in a specific form called priority sets) and the time of triggering.

Our snapshot of HiPAC consists of the common framework restricted as follows:

- *Rules*: actions are *while* programs[3].
- Δ-*type*: no restriction.

[2] HiPAC also allows a *decoupled* coupling mode, which we do not consider. Neither do we consider coupling modes between condition checking and action execution.

[3] In actuality, HiPAC allows arbitrary procedures, so $while_N$ programs as actions; however, such powerful actions overwhelm the effect of other features and render them irrelevant to a large extent.

- *add*: $add(r, q)$ is obtained by inserting r into q following the last occurrence of a rule r' such that $pri(r') \leq pri(r)$ (or at the head of q if no such rule exists).
- $merge_{imm}$: $merge_{imm}(q, q') = q$.
- $merge_{def}$: $merge_{def}(q, q')$ is obtained by repeated applications of the *add* operator to q and each of the rule occurrences in q' (in the order in which they occur in q').

ARDL

ARDL provides both immediate and deferred triggering. Rules have no explicit conditions or events (they are triggered by any update). The immediate and deferred rules are executed up to a fixpoint as specified by control structures similar to RA(Datalog$^{\neg\neg}$) programs, with no recursive triggering. Since Datalog$^{\neg\neg}$ and RA(Datalog$^{\neg\neg}$) are equivalent as update languages we can therefore suppose without loss of expressiveness that there is just one immediate rule and one deferred rule whose actions are Datalog$^{\neg\neg}$ programs executed up to a fixpoint. These programs can use global delta relations providing incremental information from the instance at the beginning of the execution of the external program to the current instance. When an action $(P)^*$ is executed, the delta relations are updated at each firing of P, although there is no triggering state until the fixpoint is reached. Since delta relations can be used in rule bodies, these updates affect the next round of firing. This is not captured in our common framework, where delta relations evolve only at trigger states. However, the computation can be easily simulated using additional private relations to keep track of the intermediate values of the delta relations.

The ARDL snapshot is obtained as the following specialization of the common framework:

- $Rules = \{r_{imm}, r_{def}\}$ where for each of the two rules *condition=true* and *action* is an RA(Datalog$^{\neg\neg}$) program of the form $(P)^*$, where P is a Datalog$^{\neg\neg}$ program.
- *cpl*: $cpl(r_{imm}) = imm$ and $cpl(r_{def}) = def$.
- ev: for each $r \in Rules$, $ev(r) = \{R^+, R^- \mid R \in sch(Rules)\}$ (thus, $ev(r)$ includes *all* relations).

- Δ-*type*: all delta relations are of type *global*.
- *add*: since there is only one deferred rule, *add* is trivial: $add(r_{def}, \epsilon) = r_{def}$.
- $merge_{imm}$: $merge_{imm}(q, q') = \epsilon$.
- $merge_{def}$: $merge_{def}(q, q') = q$.

6 Expressiveness of the Prototypes

We compare the prototypes to some of the major abstract trigger languages we have considered (such as *while* and $while_N$), as well as to each other. To streamline the discussion, we begin by examining the impact of various features on expressive power, within the common framework for the prototypes, but somewhat detached from the specific prototyopes. Specifically, we look at immediate triggering, deferred triggering, and their interaction. Within each coupling mode, we examine relevant special cases, such as the impact of using different delta relation types, or single vs. multiple occurrences of rules in execution queues. We then look at the prototypes themselves; they correspond to various combinations of the features, supplemented by occasional idiosyncrasies.

We begin by developing some notation for several specializations of the common framework. The prototypes correspond to some of these specializations. We denote by *Common* the trigger language skeleton provided by the common framework, where the mappings *add*, $merge_{imm}$ and $merge_{def}$ are computable by a relational machine in time polynomial in the length of the queues (where the measure used for the delta relations is their relational size). Note that allows for a wide variety of queuing disciplines and includes all the mappings used in the prototypes. Restrictions to the common 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 *Common* are then denoted in the following style:

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

The notation will be self-explanatory, or spelled out when needed.

The main justification of the abstract model of active databases based on relational machines is that it subsumes the trigger languages of the main active database prototypes. We now state one of the central results of the paper.

Theorem 6.1 For each *Common* program there exists a $while_N$ program (and a trigger machine) equivalent to it. In particular, for each ARDL, HiPAC, Postgres, Starburst, or Sybase program there exists a $while_N$ program (and a trigger machine) equivalent to it.

Proof: The simulation by trigger machines of the execution model provided by the common framework (and therefore of ARDL, HiPAC, Postgres, Starburst, and Sybase programs) is mostly straightforward: the persistent tape and private relations can be used for bokkeeping. The relation *external-done* is used to detect when the external program has halted and simulation of the second phase begins. There is, however, one subtlety. When local delta relations are used, it is generally necessary to remember an unbounded number of relations, corresponding to the local delta relations of rules in the immediate or deferred queues. However, trigger machines have a fixed number of relations in their relational store. Nonetheless, this can still be simulated by a trigger machine. The argument is nontrivial, and involves an extension of a result in [AV95] showing that relational machines can be augmented with an unbounded number of addressable relations of fixed arity, without increase in power. We briefly recall the argument proving that result. It uses the normal form for relational machines of given arity, outlined in Section 2. Recall that a computation in the normal form consists of two phases:

1. *Analysis phase:* this extracts from the input all information necessary to carry out the remainder of the computation on the tape alone. More specifically, for machines of arity k, the analysis phase computes $\equiv_k$ *relative to the given input*, and tables $summary_k$ which provide certain information about the action of queries with k variables on equivalence classes of $\equiv_k$. The quotient structure with respect to $\equiv_k$ is an ordered structure which is encoded on the tape (each k-equivalence class of tuples is represented by an integer).

2. *Computation phase:* this phase involves only the tape; queries and updates are simulated using the information provided by $summary_k$, available on the tape.

Note that once $summary_k$ is available, any number of relations of arity k can be encoded on the tape, as long as they are obtained from the original input by queries using no more than k variables. This proves the result of [AV95]. 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. More precisely, every time the trigger machine gains control from the external program, $\equiv_k$ and $summary_k$ are recomputed starting from the old version of $\equiv_k$ and from the database instance returned by the external program. □

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 6.1. □

Theorem 6.1 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, some of the prototypes can in fact be simulated by *while* alone.

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

6.1 Immediate triggering

We consider first the power of trigger programs with *immediate* triggering only. Consider the common framework restricted so that for each rule r, $cpl(r) = imm$. Recall the way immediate triggering works. Immediate rules can be triggered recursively. However, at each level of recursion, the

length of the immediate queue is statically bounded by the number of rules of the program. Some systems make sure the depth of recursion (i.e. the number of immediate queues awaiting execution at different levels of recursion) is also statically bounded. Let *Common(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 recursion of immediate triggering is statically bounded. Essentially, this is equivalent to *while*. More precisely, let *while*(*imm*) be the trigger language *while* restricted as follows:

- the initial run of the program does not modify the public database;
- the program's effect is null (on all relations) when run after an external program transition causing no change to the public database; and,
- the relation *external-done* does not occur in the program.

We then have:

Theorem 6.2 *Common(action: while, cpl: imm, q_{imm}: bounded)* is equivalent to *while*(*imm*).

Proof: To see that *Common(action: while, cpl: imm, q_{imm}: bounded)* is subsumed by *while*(*imm*), it is enough to note that each turn of a *Common(action: while, cpl: imm, q_{imm}: bounded)* program can be simulated by a persistent PSPACE$_r$ trigger machine. Next, recall that *while* expresses PSPACE$_r$ (Section 2). For the converse, each run of a *while*(*imm*) program w is simulated by the immediate firing of a rule whose action is w, triggered by any update to a public database relation. There is one special case, involving the simulation of the first firing of w on the initial instance $\mathbf{I}$. Since rules are only triggered after some event occurs, the first run of w must be simulated when the first nontrivial transition caused by the external program occurs. This is straightforward. □

It is easily seen that the result continues to hold if in *Common*, rule actions are restricted to FO* programs. The *while* actions are simulated by cascading rules. Note that the boundedness condition on immediate queues restricts the number of queues awaiting execution at different levels of recursion, but does not preclude cascading of rules.

If the boundedness condition is removed, one would expect the power of immediate triggering to increase. However, this is far from obvious. We

will show that the complexity of updates expressed goes up from PSPACE to EXPTIME. However, it is open whether PSPACE $\neq$ EXPTIME. So, it remains open whether any power is gained over the bounded case.

Intuitively, to perform the bookkeeping involved in unbounded immediate triggering, 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 on the complexity of updates expressible by *Common(action: while, cpl: imm)* in conjunction with the external language FO*:

Theorem 6.3 .
1. All updates expressed by *Common(action: while, cpl: imm)*[FO*] are in EXPTIME.
2. *Common(action: while, cpl: imm)*[FO*] expresses precisely the EXPTIME updates on ordered databases.

Proof: To see (1), note that every *Common(action: while, cpl: imm)*[FO*] update can be simulated using a PSPACE Turing machine with an auxiliary dpda. The inclusion in EXPTIME then follows from [C71].

Consider (2). The simulation of an EXPTIME update proceeds as follows:

(i) given an ordered database, produce a relation *input* representing the encoding of the database as a Turing machine tape;

(ii) simulate the computation of a PSPACE Turing machine with an auxiliary dpda, producing the result database encoded on the tape.

(iii) decode the database encoded on the tape once the machine has halted.

The encodings and decodings in (i) and (iii) are standard and can each be done using a *while* program (e.g., see [AHV95]). Thus, they can be simulated by initial and final rules whose actions are the needed programs. For (ii), we need to show that *Common(action: while, cpl: imm)*[FO*] programs can simulate a PSPACE Turing machine with an auxiliary dpda. Let M be such a machine. Note that an input $w = w_0 \ldots w_{n-1}$ to M can be encoded in relational form $enc(w)$ as follows. Let *succ* be a binary succsesor relation on n elements, and *input* be a binary relation containing $\langle a_i, w_i \rangle$, $0 \leq i < n$,

where a_i is the element of rank i occurring in *succ*. We will construct a trigger program P_M containing a boolean relation *init* and a unary relation *state* such that:

$$(\dagger)\ P_M(\{init := true\}) \text{ on input } enc(w) \text{ halts with } state = \{\langle s \rangle\}$$
$$\text{iff}$$
$$M \text{ on input } w \text{ halts in state } s.$$

We need to simulate the PSPACE tape of M and its auxiliary dpda. We can assume that the dpda uses the alphabet $\{bottom, 0, 1\}$ where *bottom* is used in the standard way to mark the bottom of the store. Suppose M uses no more than n^k tape cells, where n is the length of the input. The tape is encoded in relational form in a relation *tape* of arity $k+1$. The first k columns contain the n^k k-tuples of elements in *succ*, in lexicographic order as determined by *succ*. The i-th cell is represented by the tuple $\langle \vec{a}_i, d \rangle$ where $\vec{a}_i$ is the i-th k-tuple and d is the content of cell i. In addition, a unary relation *state* holds the current state of M, a unary relation *top* holds the top of the store, a k-ary relation *head* holds a k-tuple representing an integer providing the position of the head, and a relation *control* encodes the instructions of M. These instructions are of the form: *"When in state p, if the current tape cell is d and the top of the dpda store is t, move to state q, overwrite d by d', move the head left or right, and pop t or push t' on the store"*. The dpda store is encoded using a sequence of recursive immediate firings of rules. The immeidate queue at level i of the recursion will contain a rule that corresponds to the i-th store symbol (from the bottom up).

Note that the computation of M can be viewed as a sequence of PSPACE computations interrupted by operations on the store. In particular, there is a *while* program w that simulates the PSPACE segments of the computation. The program P_M uses the following rules:

- r_{init}, triggered by an update to *init*, initializes *control*, *state head*, *top*, as well as *tape* using the encoding of the input given by relations *succ* and *input*. Next,r_{init} triggers rule r and rule r_{bottom} below.
- r does the following:
  1. run the *while* program w simulating the PSPACE segment in the computation of M until an instruction of M modifying the dpda store is reached.
  2. if the instruction requires popping the top of the store, set *pop:= true*.

3. if the instruction requires pushing x on the store ($x \in \{0,1\}$), then set $top := \{\langle x \rangle\}$ and trigger r and r_x.

- three rules r_x, $x \in \{bottom, 0, 1\}$ which do the following (using pseudocode):

  if $pop = true$ **then** $pop := false$;
  else trigger r and r_x.

The priorities of the rules are such that

$$pri(r_{init}) < pri(r) < pri(r_x),$$

for $x \in \{bottom, 0, 1\}$. It can be seen that, by construction, (†) holds. Thus, (ii) is shown, so (2) is proven. □

Note that in the simulations above, no delta relations are used.

As a direct consequence of Theorem 6.3, we have:

Theorem 6.4 If PSPACE $\neq$ EXPTIME then
Common(action: while, cpl: imm, q_{imm}: bounded)
is strictly subsumed by
Common(action: while, cpl: imm).

Theorems 6.3 and 6.4 continue to hold if rule actions are restricted to FO* programs.

6.2 Deferred triggering

We next consider *Common* restricted to deferred triggering. On the side of abstract languages, let $while(def)$ and $while_N(def)$ be the *while* and $while_N$ trigger languages restricted as follows:

- the initial run of the program does not modify the public database;
- the effect of the program is null (on all relations) when run after an external program transition causing no change to the public database;
- the effect of the program is null on the public database until the external program has halted.
- the effect of the final run of the program is null if no change to the public database occurred throughout the run of the external program.

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 several immediate queues at different levels of recursion). However, unlike the immediate queues, the deferred queue is not generally statically bounded.

The power of deferred triggering is critically dependent upon the restrictions placed upon the queue, including whether multiple occurrences of rules are allowed and the Δ-type of delta relations. 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*. The following is then easily shown.

Proposition 6.1 *Common(action: while, cpl: def, queues: no-multiple-rules)* is strictly subsumed by $while(def)$.

The strictness follows from the fact that a $while(def)$ program can remember in its private relations more information about the run of the external program than the *Common(action: while, cpl: def, queues: no-multiple-rules)* program. For example, there is a $while(def)$ program w which given external program e, defines the update $w[e]$ acting on ordered input $\mathbf{I}$ as follows: $w[e] = \mathbf{I}$ if e on input $\mathbf{I}$ makes at most $|\mathbf{I}|$ transitions, and $w[e] = \emptyset$ otherwise. It can be shown that this cannot be done by a *Common(action: while, cpl: def, queues: no-multiple-rules)* program as above.

Now suppose that multiple occurrences of rules are allowed, as in HiPAC. Consider *Common(action: while, cpl: def, Δ-type: global, queues: HiPAC)* where the *add*, $merge_{imm}$ and $merge_{def}$ mappings are defined as for HiPAC, and only global delta relations are used. We first show that deferred triggering is in this case computationally equivalent to $while_N$. The inclusion into $while_N$ follows from Theorem 6.1. For the converse, we have the following.

Lemma 6.1 For every $while_N$ program w there exists a *Common(action: while, cpl: def, Δ-type: global, queues: HiPAC)* trigger program P_w and a boolean relation $start \in sch(P_w)$ such that w and $P_w[start := true]$ define the same update over $sch(w)$.

Proof: Let w be a $while_N$ program. For convenience, we assume that loops *while* $i > 0$ *do* are decomposed into two instructions *test* i; *if* $i > 0$ *go to* $\mathbf{m}$ *else go to* $\mathbf{n}$, where $\mathbf{n}$, $\mathbf{m}$ are instruction labels. Essentially, the simulation goes as follows. The nonarithmetic instructions of w are easy to simulate – they are executed directly. The arithmetic instructions are

simulated as follows. To each integer variable i of w is associated a rule r_i. The value of i is given by the number of occurrences of r_i in the deferred queue. Increasing the value of i corresponds to firing r_i so that there is one additional occurrence of r_i in the queue; decreasing i is simulated by executing an occurrence of r_i. Testing whether $i > 0$ is done by checking whether there is an occurrence of r_i in the queue.

We now describe the above in more detail. The program P_w uses the relations in $sch(w)$ and additional private relations, which we describe as needed along the way. The computation of P_w is initiated by an update *start:= true*, which triggers a rule r_{wstart}. Its action sets *start:= false* and inserts into a relation *control* an encoding of the program w and a pointer to the first instruction.

Recall that deferred rules are executed in the second phase, by cyclic executions of deferred rules (a cycle is an iteration of the loop describing the second phase in the common framework). In our simulation of w, each cycle executes either *one* arithmetic instruction of w or all nonarithmetic instructions between two arithmetic instructions. It also modifies *control* so that the pointer points to the next instruction to be executed. The latter control step, as well as the execution of nonarithmetic instructions, is carried out by a rule $r_{control}$ which is always last in each cycle. More precisely, $r_{control}$ is first triggered by r_{wstart} and does the following:

1. decode the current instruction from *control* and:

   - if the current instruction is $R := \varphi$, execute the instruction and advance the pointer to the next instruction of w; go to (1).
   - if the current instruction is *while* φ then evaluate φ and move the pointer accordingly to the next instruction; go to (1).
   - if the current instruction is $i := i + 1$ then trigger r_i.
   - if the current instruction is $i := i - 1$ or *test* i then set $R_i := \{-\}$ or $R_i := \{test\}$, respectively, where R_i is a private relation of P_w for each integer variable i of w.
   - if the current instruction is *if* $i > 0$ *then goto* **m** *else goto* **n** then move the pointer to **m** if $exists_i = true$ and to **n** otherwise, where $exists_i$ is a boolean relation indicating if some occurrence of r_i has been executed in the current cycle. Reset $exists_i$ to *false*.
   - if the current instruction is the end of the program then set $end := true$, where *end* is a boolean private relation.

2. if $end = false$ trigger $r_{control}$.

Each rule r_i does the following. If $end = true$ then nothing is done. Otherwise,

1. if $R_i = \emptyset$ then trigger r_i;

2. if $R_i = \{-\}$ then set $R_i := \emptyset$;

3. if $R_i = \{test\}$ then execute $R_i := \emptyset$, $exists_i := true$ and trigger r_i.

Clearly, P_w simulates w. □

Thus, deferred triggering with multiple occurrences of rules provides as much computational power as $while_N$ (or relational machines). Note that the delta relations are not used at all in the simulation of Lemma 6.1. However, delta relations become important in the simulation of $while_N(def)$ as a trigger language. In fact, we need delta relation of Δ-type *local-fluid*.

Theorem 6.5 *Common(action: while, cpl: def, Δ-type: local-fluid, queues: HiPAC)* is equivalent to $while_N(def)$.

Proof: The subsumption of

Common(action: while, cpl: def, Δ-type: local-fluid, queues: HiPAC)

by $while_N(def)$ follows from Theorem 6.1. For the converse, let w be in $while_N(def)$, over schema $\mathbf{R} \subset sch(w)$. By Lemma 6.1, there exists a

Common(action: while, cpl: def, Δ-type: local-fluid, queues: HiPAC)

program P_w and a boolean relation $start \in sch(P_w)$, such that $P_w[start := true]$ defines the same update over $sch(w)$ as w. Recall from the construction of P_w that the program includes a boolean private relation *end* that signals the end of the simulation of w.

It remains to simulate w as a trigger program. Recall that w may perform nontrivial computations on its private relations, every time a transition is caused by the external program. On the other hand, a

Common(action: while, cpl: def, Δ-type: local-fluid, queues: HiPAC)

program cannot carry out any computation before the external program has halted, except maintaining the deferred queue. To get around the problem, we use a deferred rule whose Δ-type is *local-fluid* and which remembers the

intermediate instances in the computation of the external program. Then the computation of w on these instances can be simulated when the deferred queue is executed, using P_w. This requires some care, because of the interaction of the cycles of P_w with the other rules.

We now describe this more precisely. We use instead of P_w a version P'_w that operates on a copy of $sch(P_w)$ consisting of a relation Q' for each $Q \in sch(P_w)$. In particular, there is a copy $\mathbf{R}'$ of the public database schema $\mathbf{R}$, P'_w is triggered by an assignment $start' := true$, and end' signals the end of the simulation of w. The program P'_w works similarly to P_w, with one difference. Recall that P_w simulates a single run of w and halts. The only way for P_w to halt is for the deferred queue to become empty. However, the values of integer variables are remembered using occurrences of rules in the queue, and the values of these variables are persistent between turns of w as a trigger program. Therefore, we modify P'_w so that the rules used to encode integer values are preserved in the queue until the last turn of w, which constitutes the deferred phase. This is detected using the boolean relation *deferred-phase* discussed below. The details of this modification are straightforward and are omitted.

Besides P'_w, we use two additional rules:

- a rule r whose Δ-type is *local-fluid*, and $ev(r) = \{R^x, run\text{-}r^x \mid R \in \mathbf{R}, x \in \{+,-\}\}$, where *run-r* is a private boolean relation used to control firings of r.
- a rule r_{last} that signals the end of a cycle.

The priorities of the rules are the following:

$$pri(q) < pri(r) < pri(r_{last})$$

where $q \in Rules(P'_w)$.

Additionally, we use the following private boolean relations:

- *init* to indicate whether the needed initialization of the computation has taken place;
- *deferred-phase* used to inhibit r in the simulation of the "deferred" phase of w, i.e. the execution of w after the external program has halted. It is also used to signal to P'_w that the rules encoding values of integer variables need not persist after the end of the last turn, thus allowing the program to terminate.

Recall that by definition, *all* private boolean relations start out being *false*.

The rule r does the following. If the value of *run-r* at the time of triggering of r is *false* or *deferred-phase* is *true* then nothing is done. Otherwise, perform in order:

1. if $init = false$ then do
   - $init := true$;
   - trigger r_{last};
   - $end' := true$;
   - $\mathbf{R}'' := \mathbf{R}$ (this saves the public database instance at the end of the execution of the external program in $\mathbf{R}''$).
2. if $end' = false$ then do
   - assign to $\mathbf{R}$ its value at the time of triggering of r (reconstructed using the current instance saved in $\mathbf{R}''$ and the delta relations for $\mathbf{R}$). This triggers r. Note however that the value of *run-r* when this happens is *false*, so this rule will be inhibited.
   - *run-r* $:=$ *true*; as a side effect, this triggers again an inhibited occurrence of r (since *run-r* at the time of triggering is *false*).
   - $\mathbf{R} := \mathbf{R}''$ and *run-r* $:=$ *false*. This triggers r. Its delta relations for $\mathbf{R}$ of the triggered rule are the same as those of the currently executing occurrence of r.

   The purpose of this maneuver is to trigger again r with the same delta relations for $\mathbf{R}$, at the cost of also triggering a useless version of r which is inhibited when its turn comes, using the *run-r* relation.
3. if $end' = true$ then do
   - $end' := false$;
   - assign to $\mathbf{R}'$ the value of $\mathbf{R}$ at the time of triggering of r (reconstructed using the current instance saved in $\mathbf{R}''$ and the delta relations for $\mathbf{R}$).
   - $start' := true$ (this triggers the first rule of P'_w).

The rule r_{last} does the following:

1. if $end' = false$ then trigger r_{last};

2. if $end' = true$ then do

   - if *deferred-phase* $=$ *false* then *deferred-phase:*$=$ *true*; $start' :=$ *true*
   - else $\mathbf{R}:= \mathbf{R}'$ (this copies the content of $\mathbf{R}'$ into $\mathbf{R}$).

Let us trace the simulation. Suppose the consecutive public database instances over $\mathbf{R}$ in the execution of the external program are

$$\mathbf{I}^0, \mathbf{I}^1, \ldots \mathbf{I}^n, \mathbf{I}^{curr}.$$

The deferred queue when the external program halts is

$$r(\Delta_0), r(\Delta_1), \ldots r(\Delta_n)$$

where Δ_i represents the incremental information between $\mathbf{I}^i$ and $\mathbf{I}^{curr}$. The first cycle is a special case, detected by *init* $=$ *false*. Once the initialization is done and the first cycle executed, subsequent cycles are of the form

$$\vec{q}\ r(\Delta_i) \ldots r(\Delta_n)\ r_{last}$$

where $1 \leq i \leq n$ and $\vec{q}$ denotes the rules of P'_w in the current cycle. The rules $\vec{q}$ execute part of the simulation of w on $\mathbf{I}^{i-1}$. While this happens, the rules $r(\Delta_i) \ldots r(\Delta_n) r_{last}$ trigger themselves again (with the same delta relations). The end of the simulation of w by P'_w is signaled by $end' = true$ (but the rules encoding the values of the integer variables of w at the end of its run remain in the queue). Then the rule $r(\Delta_i)$ reconstructs $\mathbf{I}_i$ into $\mathbf{R}'$ and triggers P'_w for another simulation of w. The other rules trigger themselves with the same delta relations. Eventually, no more occurrences of r are left. This is detected by r_{last} when $end' = true$ (signaling that there has been no execution of an occurrence of r since the termination of the simulation of the last round w). This causes the deferred phase in the simulation to begin. The rules r are definitively inhibited using *deferred-phase* and P'_w is triggered for a last run corresponding to the deferred run of w following the execution of the external program. Lastly, the final result is copied from $\mathbf{R}'$ into $\mathbf{R}$. □

Theorem 6.5 implies that

Common(action: while, cpl: def, Δ-type: local-fluid, queues: HiPAC)

in combination with any nontrivial external language expresses the same queries as $while_N$, and in particular is complete on ordered databases:

Corollary 6.1 Let E be an external language such that for each schema $\mathbf{R}$ there exists p in E such that $p(\mathbf{I}) \neq \mathbf{I}$ for each $\mathbf{I}$ over $\mathbf{R}$.

Common(action: while, cpl: def, Δ-type: local-fluid, queues: HiPAC)[E]

expresses the same updates as $while_N$, and in particular all computable updates on ordered databases.

As we have seen from Proposition 6.1, if no multiple occurrences of rules are allowed in queues, then *Common* with deferred triggering is subsumed by *while*(*def*), so there is a dramatic loss of power. As an intermediate possibility, 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 result which provides information on the complexity and expressiveness of *Common(action: while, cpl: def, queues: no-multiple-*Δ*)*.

Theorem 6.6 .
1. All updates expressible in

Common(action: while, cpl: def, queues: no-multiple-Δ*)*[FO*]

are in EXPSPACE.
2. *Common(action: while, cpl: def, queues: no-multiple-*Δ*)*[FO*] expresses precisely the EXPSPACE updates on ordered databases.

Proof: Consider (1). The inclusion in EXPSPACE follows from the fact that the length of the deferred queue is exponentially bounded (as no multiple occurrences of rules with the same delta relations are allowed), and the action of each rule is a *while* program which requires PSPACE.

For (2), the proof goes along the same lines as that for Theorem 6.3. The simulation of an EXPSPACE update proceeds as follows:

(i) given an ordered database, produce a relation *input* representing the encoding of the database as a Turing machine tape;

(ii) simulate the computation of a PSPACE Turing machine with an auxiliary dpda, producing the result database encoded on the tape.

(iii) decode the database encoded on the tape once the machine has halted.

The encodings and decodings in (i) and (iii) do not pose problems. For (ii), we show that a *Common(action: while, cpl: def, queues: no-multiple-*Δ*)*[FO*] program can simulate 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.

More precisely, let M be an EXPSPACE Turing machine, which on input of length n uses no more than 2^{n^k} tape cells. As described in the proof of Theorem 6.3, an input $w = w_0 \ldots w_{n-1}$ to M can be encoded in relational form $enc(w)$ using a binary successor relation *succ* on n elements, and a binary relation *input* consisting of the pairs $\langle a_i, w_i \rangle$ $0 \leq i < n$, where a_i is the element of rank i occurring in *succ*. We will construct a trigger program P_M containing a boolean relation *init* and a unary relation *state* such that:

(†) $P_M[\{init := true\}]$ on input $enc(w)$ halts with $state = \{\langle a \rangle\}$ iff M on input w halts in state a.

In addition to *succ*, *init*, *state* and R_{input}, P_M uses the following relations:

- a unary relation *cell* that is used to remember the content of tape cells;
- a relation *counter* of arity $k+1$ that is used to count up to 2^{n^k}. This is used to identify tape cells. In the first k columns *counter* contains the n^k tuples of arity k using elements from *succ*, in lexicographic order as given by *succ*. *Counter* encodes a binary integer of length n^k: the digit d in position i is given by the tuple $\langle \vec{a}_i, d \rangle$, where $\vec{a}_i$ is the i-th k-tuple of elements of *succ* in lexicographic order.
- a relation *head* of arity $k+1$ which encodes an integer up to 2^{n^k} in the same manner as *counter*. This is used to identify the position of the head of M.
- a relation *control* which encodes the instructions of M. Each instruction is of the form *"when in state p and reading a on the tape, move to state q, overwrite a by b, and move the head right or left"*. We denote the encoding of integer n into *head* or *counter* by $enc(n)$. By abuse of notation, we sometimes denote the integers encoded by *head* and *counter* by the relations themselves.

The trigger program P_M uses two rules:

- rule r_{init} which is triggered by an update to *init* and which initializes the tape by triggering 2^{n^k} occurrences of rule r below. More precisely, this is done as follows (using pseudo-code):

  $counter := enc(0); state := \{\langle start \rangle\}$; initialize *control*;
  while $counter < 2^{n^k}$ **do**
  if $counter < n$ **then** trigger $r(counter, w_{counter})$
  else trigger $r(counter, \flat)$;
  $counter := counter + 1$;
  od

  The symbol $\flat$ is the blank symbol of M's tape. The command *trigger r(P,Q)* stands for a sequence of commands triggering r so that the values of relations *counter* and *cell* at the time of triggering are P and Q. This involves resetting *counter* and *cell* and results in undesired firings of r which are inhibited using a similar technique as in the proof of Theorem 6.5 (details are omitted).

- rule r is triggered by updates to *counter* and *cell* and has delta relations of type *local-fluid*. The idea is that an occurrence of r in the queue provides the value of the tape cell indexed by *counter* in the current tape. These are provided by the values of *counter* and *cell* at the time of triggering. The rule r does the following (using again pseudocode):

  if $head \neq counter$ **then** trigger *r(restored(counter),restored(cell))*
  else do
  trigger *r(restored(counter),new(cell));*
  $head := new(head)$;
  $state := new(state)$;
  od

  where *restored(counter)* and *restored(cell)* denote the values of *counter* and *cell* at the time of triggering of r, as reconstructed from their current values and the corresponding local-fluid delta relations. The expressions *new(cell)*, *new(state)* and *new(head)* denote the new values of the head, state, and tape cell as provided by the *control* relation using the current state provided in *state* and the content of the tape cell as provided by *restored(cell)*.

It can be seen that (†) holds by construction. Thus, $P_M[\{init := true\}]$ simulates M. This proves (ii) and (2). □

We also note that Theorem 6.6 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 6.7 *Common(action: while, cpl: def, queues: no-multiple-rules)* is strictly subsumed by

Common(action: while, cpl: def, queues: no-multiple-Δ)

which is strictly subsumed by

Common(action: while, cpl: def, queues: HiPAC).

Proof: The subsumption of *Common(action: while, cpl: def, queues: no-multiple-rules)* by *Common(action: while, cpl: def, queues: no-multiple-Δ)* is immediate. The strictness follows from Theorem 6.6 (2): *Common(action: while, cpl: def, queues: no-multiple-Δ)*[FO*] expresses EXPSPACE-complete updates, whereas *Common(action: while, cpl: def, queues: no-multiple-rules)* is subsumed by *while(def)* so *Common(action: while, cpl: def, queues: no-multiple-rules)*[FO*] expresses only updates in PSPACE. Since PSPACE ≠ EXPSPACE by the Space Hierarchy Theorem, the strict subsumption follows.

Similarly to Theorem 6.1, it can be shown that *Common(action: while, cpl: def, queues: no-multiple-Δ)* is subsumed by $while_N(def)$. By Theorem **??**, *Common(action: while, cpl: def, queues: HiPAC)* is equivalent to $while_N(def)$. It follows that *Common(action: while, cpl: def, queues: no-multiple-Δ)* is subsumed by *Common(action: while, cpl: def, queues: no-multiple-Δ)*. The strict subsumption follows from a complexity argument. By Theorem **??**, all updates in *Common(action: while, cpl: def, queues: no-multiple-Δ)*[FO*] can be evaluated in EXPSPACE. On the other hand, $while_N(def)$[FO*] expresses all computable updates on ordered databases, in particular updates that are not in EXPSPACE. □

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. However, if one assumes that immediate programs have access to the *external-done* flag, then $while_{(N)}(def)$ is strictly subsumed by $while_{(N)}(imm)$.

Things are complicated for the prototypes 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.

6.3 Mixed triggering

We now examine the power of *Common* with both immediate and deferred triggers. We have seen that *Common* is subsumed by $while_N$ and so by trigger machines (Theorem 6.1). How about the converse? It turns out that this does not hold, because of technical subtleties involving the persistent integer variables of $while_N$ trigger programs (and equivalently, the persistent tape of trigger machines). However, *Common* can essentially simulate $while_N$ trigger programs with no persistent integer variables, even when rule actions are limited to FO*. We show this next.

By Theorem 6.5, *Common* with deferred rules is equivalent to $while_N(def)$. The simulation used the queue discipline of HiPAC. On the other hand, *Common* with immediate rules only has complexity bounded by EXPTIME, as shown in Theorem 6.3. It is clear then that to obtain the full power of $while_N$, which has no complexity bound, one has to use a triggering discipline where deferred rules recursively generated in the course of the execution of immediate rules can be integrated in 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

Common(action: FO*, $merge_{imm}$, $merge_{def}$: Starburst, add: HiPAC)

which mixes elements of the queuing disciplines of Starburst and HiPAC. In particular, the $merge_{imm}$ mapping of Starburst integrates the deferred queues produced while executing immediate rules back into the immediate

queue. This is done using the *add* mapping of HiPAC, which allows for multiple occurrences of rules unlike the *add* mapping of Starburst. Note that in terms of nested transactions, this corresponds to executing the deferred rules generated by a subtransaction immediately following that subtransaction, rather than placing them at the top level, as is done in HiPAC.

We first note that the analog of Lemma 6.1 holds for the deferred restriction of the above.

Lemma 6.2 For every $while_N$ program w there exists a

$$Common(action:\ FO^*,\ cpl:\ def,\ merge_{imm},\ merge_{def}:\ Starburst,\ add:\ HiPAC)$$

trigger program P_w and a boolean relation *start* such that w and $P_w[start := true]$ define the same update over $sch(w)$.

Proof: The basic idea for the simulation of w is the same as in the proof of Lemma 6.1. The main problem is the simulation of arithmetic. This was done in the proof of Lemma 6.1 by associating to each integer variable i of w a rule r_i. The value of i was represented by the number of occurrences of r_i in the current execution queue. The simulation is now complicated by the fact that deferred rules are no longer fired in cycles as in HiPAC, but new deferred rules are instead immediately integrated into the current queue. Since different rules are totally ordered in the queue by their priority, one cannot for example decrease the number of occurrences of r_i while there are still occurrences of r_j such that $pri(r_j) < pri(r_i)$. To get around the problem, we now use a single rule r. Intuitively, the rule r must replace the various rules used in the proof of Lemma 6.1. To keep track of which rule r is supposed to simulate, we use a relation *rule* whose value at the time of triggering of r identifies that rule. Thus, to simulate a firing of some rule q, q is placed in relation *rule* and r is triggered. When r is executed, it first reconstructs the value of *rule* at the time of its triggering using a delta relation of type *local-fluid*, then simulates the rule found in *rule*. Apart from this modification, the simulation procedes in the same way as in the proof of Lemma 6.1. □

Let $while_N(imm, def)$ denote the $while_N$ trigger programs satisfying the following:

- the initial run does not change the state of the public database;

- the effect of the program is null (on all relations) if no change to the public database occurred since its previous turn;
- the effect of the final run of the program is null if the public database remains unchanged throughout the run of the external program.
- the values of the integer variables of the program are not persistent between runs (i.e., they are reinitialized to zero before each run).

We can now show:

Theorem 6.8 *Common(action: FO*)* subsumes $while_N(imm, def)$.

Proof: Let w be a $while_N(imm, def)$ trigger program over public database schema $\mathbf{R}$. We construct a

Common(action: FO, cpl: def, merge$_{imm}$, merge$_{def}$: Starburst, add: HiPAC)*

program that is equivalent to w. By Lemma 6.2, there exists a

Common(action: FO, cpl: def, merge$_{imm}$, merge$_{def}$: Starburst, add: HiPAC)*

trigger program P_w and a boolean relation *start* such that w and $P_w[start := true]$ define the same update over $sch(w)$. By examination of the proof of Lemma 6.2 it can be seen that we can assume without loss of generality that the value of *start* when $P_w[start := true]$ ends is *true*, and that all rules in P_w have null effect if $start = false$. (We can thus use the value of *start* to inhibit undesired firings of rules in P_w.) We also use a version P'_w which is the same as P_w except that $\mathbf{R}$ is replaced by $\mathbf{R}'$ and *start* by *start*$'$.

The program used to simulate w as a trigger program uses in addition to the private relations of P_w and P'_w the following private boolean relations:

- *init*, used to identify the initialization phase of w;
- *deferred*, used to signal the beginning of the simulation of the deferred phase of w.

The program uses the rules of P_w and P'_w and the following rules:

- r_{init}, triggered in immediate mode by any update to $\mathbf{R}$. Its delta relations are global. The rule does the following: if $init = true$ do nothing. Otherwise,

1. set $\mathbf{R}' :=$ restored($\mathbf{R}$) (this restores into $\mathbf{R}'$ the initial database state, using the current instance and the delta relations);
2. set $start' := true$ (this triggers P'_w);
3. set $init := true$;

- r_w, triggered in immediate mode by any update to $\mathbf{R}$. If $deferred = true$, r_w does nothing. Otherwise, its only action is to set $start := true$ (thus triggering P_w if the previous value of $start$ is $false$).
- r_{reset}, triggered in immediate mode by any update to $\mathbf{R}$. If $deferred = true$, r_{reset} does nothing. Otherwise, its action consists of setting $start$ to $false$.
- r_{def}, triggered in deferred mode by any update to $\mathbf{R}$. If $start = true$ or $deferred = true$, r_{def} does nothing. Otherwise, it sets $deferred := true$ and $start := true$ (the latter triggers P_w).

The priorities of the rules are such that:

$$pri(r_{init}) < pri(q') < pri(r_w) < pri(q) < r_{def} < r_{reset}$$

where $q \in Rules(P_w)$ and $q' \in Rules(P'_w)$.

Let us trace the simulation. Each nontrivial transition caused by the external program triggers r_{init}, r_w and r_{reset} in immediate mode and r_{def} in deferred mode. However, the effect of r_{init} is null except for its first run. This simulates the initialization run of w by restoring the original state and triggering P'_w. Then rule r_w triggers P_w, thus simulating a usual run of w. Note that the execution of P_w may trigger again r_{init}, r_w, and r_{reset} in immediate mode and r_{def} in deferred mode. All of the triggered rules (including r_{def}) are reintegrated into the immediate queue. Thus, the immediate queue after the execution of P_w is of the form

$$(r_{init})^* \ (r_w)^* \ (r_{def})^* r_{reset} (r_{reset})^*.$$

In the above queue, we only wish for r_{reset} to execute. The execution of the other rules is undesired and is inhibited, or has null effect. More specifically, r_{init} is inhibited using $init$, which is $true$ (it became true at the first run of r_{init}). The occurrences of r_w have null effect, since their action is to set $start := true$ and the value of $start$ is already true. Thus, they do not trigger P_w. The occurrences of r_{def} are inhibited since $start = true$. Lastly, r_{reset} sets $start := false$, and further occurrences of this rule repeat the same action

and so have null effect. Control is now returned to the external program. When the next nontrivial transition occurs, the process is repeated.

The deferred rule r_{def} is used to trigger the final run of w. Note that although r_{def} is triggered at each transition, only its first occurrence in the deferred queue has a non null effect, since the others are inhibited using the *deferred* flag. Also, the action of P_w may trigger again the immediate rules, which are however inhibited by the fact that $deferred = true$. □

Note that *Common(action: FO*)* is not subsumed by $while_N(imm, def)$: the lack of persistent integer variables in $while_N(imm, def)$ prevents performing the bookkeeping needed to simulate the unbounded deferred queues generated by *Common(action: FO*)* programs. As one might expect, things fall into place in the bounded case. Let $while(imm, def)$ be the *while* trigger language restricted in the same manner as $while_N(imm, def)$ (the restriction involving persistence of integer values is vacuous since there are no integer variables). It is straighforward to show (proof omitted):

Theorem 6.9 *Common(action: FO*,* q_{imm}*: bounded,* q_{def}*: bounded)* is equivalent to $while(imm, def)$.

6.4 Prototypes vs. abstract languages

We next relate the prototypes to the abstract trigger languages *while* and $while_N$ (and their variations). This uses the results on the restrictions of the common framework established earlier in this section.

As is clear by now, the language *while* plays an important role. Indeed, we have[4]:

Theorem 6.10 .
1. ARDL is equivalent to $while(imm, def)$.
1′. ARDL(imm) is equivalent to $while(imm)$.
1″. ARDL(def) is strictly subsumed by $while(def)$.
2. Starburst is strictly subsumed by $while(def)$.
3. Sybase is equivalent to $while(imm)$.

Proof: (1) follows from Theorem 6.9 and the fact that ARDL is equivalent to *Common(action:* while, q_{imm}*: bounded,* q_{def}*: bounded)*.

[4] ARDL(imm), and ARDL(def) denote the restrictions of ARDL to immediate resp. deferred rules.

To see (1), (1′) and (1″), note first that ARDL is equivalent to *Common(action:* while, q_{imm}*: bounded,* q_{def}*: bounded)*, ARDL(*imm*) is equivalent to *Common(action: while, cpl: imm,* q_{imm}*: bounded)*, and ARDL(*def*) is equivalent to *Common(action:* while, *cpl: def, queues: no-multiple-rules)*. These equivalences follow from the following observations: (i) ARDL has only global delta relations, so their number is statically determined; (ii) Rule actions are Datalog$^{\neg\neg}$ programs, and Datalog$^{\neg\neg}$ is equivalent to *while*, which can be simulated by cascading rules with FO* actions; (iii) The immediate and deferred queues consist of at most one rule each. (iv) The depth of recursion of immediate triggering is at most two, since there is only one immediate rule whose action is idempotent. Then (1), (1′) and (1″) follow from the above equivalences and Theorems 6.9, 6.2 and Proposition 6.1, respectively.

(2) follows from Proposition 6.1 and the fact that Starburst is subsumed by *Common(action:* while, *cpl: def, queues: no-multiple-rules)*. (3) follows from Theorem 6.2 and the fact that Sybase is equivalent to *Common(action: while, cpl: imm,* q_{imm}*: bounded)*. To see the latter, note that the depth of recursion of immediate triggering in Sybase is bounded by the number of relations in the schema, as a consequence of the syntactic nonrecursiveness condition satisfied by Sybase programs (see Section 5.2). □

The complexity of the updates expressed by the prototypes is summarized in the following. The result follows immediately from Theorem 6.10 and the fact that *while*[FO*] expresses precisely the PSPACE updates on ordered databases.

Theorem 6.11 ARDL[FO*], ARDL(*imm*)[FO*], ARDL(*def*)[FO*], Starburst[FO*], and Sybase[FO*] express precisely the PSPACE updates on ordered databases.

Postgres and HiPAC are no longer subsumed by *while*, and their complexity goes beyond PSPACE. For Postgres, we have the following. Recall that Postgres works only on ordered databases and with tuple-at-a-time external programs, so all comparisons assume the above.

Theorem 6.12 .
1. Postgres is strictly subsumed by $while_N(imm)$ and strictly subsumes *while*(*imm*) if PSPACE $\neq$ EXPTIME.
2. Postgres[*while*] expresses precisely the EXPTIME updates on ordered databases.

Proof: Consider (1). By Theorem 6.2, *while(imm)* is equivalent to *Common(action: while, cpl: imm, q_{imm}: bounded)*, which is clearly subsumed by Postgres. Thus, Postgres subsumes *while(imm)*. For the strict subsumption, note that *while*(*imm*)[FO*] is included in PSPACE. On the other hand, Postgres[FO*] expresses EXPTIME-complete updates. Indeed, from Theorem 6.3 it follows that *Common(action: while, cpl: imm)*[FO*] expresses EXPTIME-complete updates. Postgres differs from *Common(action: while, cpl: imm)* in that (i) the *ev* mapping is single-valued, and (ii) delta relations are restricted to be of type *local-fixed*. However, note that in the simulation proving (2) of Theorem 6.3, no delta relations are used and rules are triggered by updates to just one relation. Thus, the proof goes through for Postgres, so Postgres[FO*] expresses EXPTIME-complete updates. It follows that Postgres strictly subsumes *while(imm)* if PSPACE $\neq$ EXPTIME.

As noted above, Postgres is subsumed by *Common(action: while, cpl: imm)*. The latter is subsumed by *while*$_N$(*imm*), by Theorem 6.1. The strict subsumption follows from the fact that *Common(action: while, cpl: imm)*[FO*] is included in EXPTIME by Theorem 6.3, whereas *while*$_N$(*imm*)[FO*] expresses all computable updates on ordered databases, so has no complexity bound.

(2) follows from the above remarks and from Theorem 6.3. □

We consider now HiPAC. We denote by HiPAC(*imm*) and HiPAC(*def*) the set of HiPAC programs using only immediate, respectively deferred rules.

Theorem 6.13 .
1. HiPAC(*def*) is equivalent to *while*$_N$(*def*).
2. HiPAC(*imm*) is strictly subsumed by *while*$_N$(*imm*) and strictly subsumes *while*(*imm*) if PSPACE $\neq$ EXPTIME.
3. HiPAC(*imm*)[FO*] expresses precisely the EXPTIME updates on ordered databases, and HiPAC(*def*)[FO*] expresses all computable updates on ordered databases.

Proof: Consider (1). The subsumption of HiPAC(*def*) by *while*$_N$(*def*) follows from Theorem 6.1. For the converse, note that *Common(action: while, cpl: def, Δ-type: local-fluid, queues: HiPAC)*subsumes *while*$_N$(*def*) by Theorem 6.5 and also that *Common(action: while, cpl: def, Δ-type: local-fluid, queues: HiPAC)* is a restriction of HiPAC(*def*). Thus, HiPAC(*def*) subsumes *while*$_N$(*def*).

For (2), note first that HiPAC(*imm*) is equivalent to Postgres (see the direct proof of this fact within the proof of Theorem 6.14. Next, use Theorem

6.12. Consider (3). The fact that HiPAC(*imm*)[FO*] expresses EXPTIME on ordered databases follows again from the equivalence of HiPAC(*imm*) with Postgres and from Theorem 6.12. The fact that HiPAC(*def*)[FO*] expresses all computable updates on ordered databases follows from Corollary 6.1 and the fact that

Common(action: while, cpl: def, Δ-type: local-fluid, queues: HiPAC)

is a restriction of HiPAC(*def*). □

6.5 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. Using this, we next establish relationships among the prototypes themselves.

Theorem 6.14 The relationships in Figure 1 hold among the prototypes (and their *immediate* and *deferred* restrictions, where appropriate).

Proof: The equivalence of ARDL(*imm*) and Sybase follows from the fact that they are both equivalent to *while(imm)* by Theorem 6.10. The strict subsumption of both prototypes by HiPAC(*imm*) assuming PSPACE $\neq$ EXPTIME follows from the above and Theorem 6.13 (2). To see the strictness of the subsumption of HiPAC(*imm*) by HiPAC, note that HiPAC(*imm*)[FO*] expresses EXPTIME on oredered databases by Theorem 6.13 (3), and HiPAC(*def*)[FO*] (and therefore HiPAC[FO*]) expresses all computable updates on ordered databases.

Consider the equivalence of HiPAC(*imm*) and Postgres on ordered databases and relative to tuple-at-a-time external programs. Note first that, by the descriptions in Section 5.2, Postgres is a restriction of HiPAC(*imm*) where (i) *ev* is single valued and (ii) all delta relations are of type *local-fixed*. In view of the tuple-at-a-time restrictions, the comparison is made when in both cases rule actions are *while* programs operating tuple-by-tuple. So HiPAC(*imm*) subsumes Postgres by definition. To see the converse, note that (i) multi-valued *ev* mappings can be simulated by single-valued mappings with duplication of rules, and (ii) the delta-relations of type *global* and *local-fluid* of HiPAC(*imm*) can be simulated by Postgres. Indeed, *global* delta relations can be maintained explicitly using private relations. Consider the simulation of *local-fluid* delta relations by *local-fixed* delta relations. Consider a

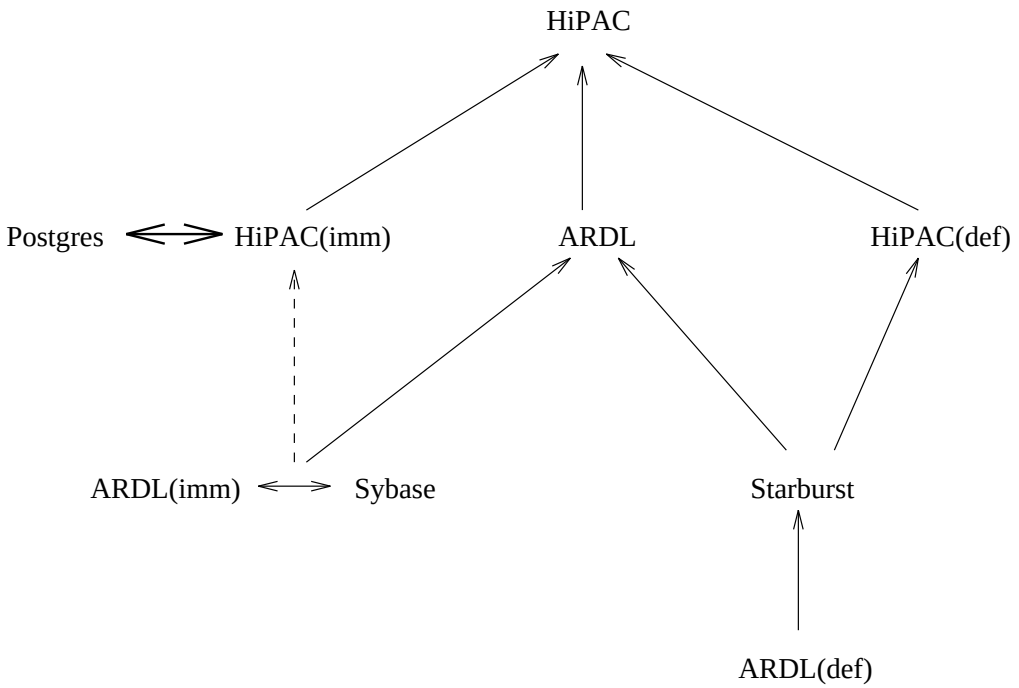

Figure 1: Relative expressiveness of prototypes. Solid single arrows indicate strict subsumption. The broken single arrow indicates subsumption, and strict subsumption assuming that PSPACE $\neq$ EXPTIME. The double solid arrow indicates equivalence. The double boldfaced arrow indicates equivalence on ordered databases and tuple-at-a-time external programs.

rule r triggered by an update to some relation R. Note that at the time of imediate triggering of r, the two types of delta relations associated with R coincide. However, they may become out of date as R is updated by other rule occurrences executed before the one under consideration. Essentially, at any given time we have a stack of rule occurrences holding *local-fixed* delta relations for R, of the form

$$r(\Delta_R^{+-}(I_0, I_1)), r(\Delta_R^{+-}(I_1, I_2)), \ldots, r(\Delta_R^{+-}(I_{n-1}, I_n)),$$

where $\Delta_R^{+-}(I_i, I_{i+1})$ is the *local-fixed* delta relation associated with the transition $\langle I_i, I_{i+1} \rangle$, and $I_1, I_2, \ldots, I_n$ are the consecutive values of R. Clearly,

the *local-fluid* versions of these delta relations can be recreated in private relations as the stack is executed, by cumulating the *local-fixed* delta relations of R as the rules are executed.

The strict subsumption of ARDL(imm) by ARDL is due to the fact that the former does not "know" when the external program has ended. As an example, there is clearly an ARDL program M over public schema R which given an external program E over R defines the update $M(E)(R) = R$ if E causes no notrivial transition, and $M(E) = \neg E(R)$ otherwise. It is easily seen that there is no ARDL(imm) program with the same semantics.

Consider the strict subsumption of ARDL by HiPAC. The subsumption follows from the fact that ARDL is subsumed by *Common(action: FO*, q_{imm}: bounded, q_{def}: bounded)* which is clearly subsumed by HiPAC. The strictness follows from the fact that ARDL[FO*] is in PSPACE by Theorem 6.11, whereas HiPAC(*def*)[FO*] (and therefore HiPAC[FO*]) expresses all computable updates on ordered databases by Theorem 6.13.

The subsumption of ARDL(*def*) by Starburst is obvious. For the strictness, consider the public database schema P, Q, *flag* and the update which, given external program E, sets *flag* = *true* iff some transition in the execution of E modifies P. Clearly, there is a Starburst program that defines this update. Indeed, it is enough to have a deferred rule for P. If the rule occurs in the deferred queue, P has been modified. On the other hand, this is not feasible in ARDL, because there is only one deferred rule r_{def} triggered by any update to the public database. Thus, an ARDL program has the same effect in the case where the external program modifies P and then resets it to its initial value, as in the case of an external program doing the same for Q. In both cases, the deferred queue consists of r_{def} where all global delta relations are empty.

To see the subsumption of Starburst by ARDL, it is enough to note that ARDL can simulate the evolution of the deferred queue of Starburst throughout the run of the external program, using private relations and the immediately triggered rule r_{imm}. The deferred rule r_{def} then simulates the execution of the deferred queue of Starburst as encoded in the private relations. The strictness follows from the fact that ARDL has immediate triggering, and immediately triggered rules can affect the execution of the external program. This cannot be done by Starburst.

The subsumption of Starburst by HiPAC(imm) is immediate. The strictness follows from the fact that Starburst[FO*] is in PSPACE, whereas HiPAC(*def*)[FO*] expresses all computable updates on ordered databases, by Theorem 6.13.

Finally, the strictness of the subsumption of HiPAC(*def*) by HiPAC follows from the fact that HiPAC allows immediately triggered rules which can affect the execution of the external program. □

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.

7 Semantic Consistency of Triggers

We return in this section to the issue of semantic consistency of triggers. When an external program is executed in conjunction with a trigger program, its effect is modified by the triggers. As discussed 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

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

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 3.3). 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 7.1 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 public database R is semantically consistent:

$$\begin{array}{lcl} R & := & \{t \mid (\neg \mathit{third} \wedge R(t)) \vee (\mathit{third} \wedge R(t) \wedge \varphi(R))\}; \\ \mathit{third} & := & \mathit{second}; \\ \mathit{second} & := & \mathit{true} \end{array}$$

where *second* and *third* are boolean. Indeed, suppose first that $\varphi(R)$ is valid. Then each run of the trigger program has null effect on R so it is obviously semantically consistent. Conversely, suppose $\varphi(R)$ is not valid, so there exists I over R such that $\varphi(I)$ is false. Let M_e be the external program $R := R \cup \{t\}$ and M'_e be $R := R - \{t\}$; $R := R \cup \{t\}$. Clearly, M_e and M'_e are euquivalent as update programs. However, $M_t[M_e]$ and $M_t[M'_e]$ do not define the same update. Indeed, $M_t[M_e](I) = I \cup \{t\} \neq \emptyset$ whereas $M_t[M'_e](I) = \emptyset$. □

One can relativize the notion of semantic consistency to languages for external and trigger programs in the natural way. The proof of Theorem 7.1 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*(Datalog) = $\{e^* \mid e \in$ RA(Datalog) and ';' occurs in $e\}$.

Proposition 7.1 Datalog trigger programs are semantically consistent relative to RA*(Datalog) external programs.

Proof: Let P be a Datalog trigger program and e^*, f^* be equivalent RA*(Datalog) programs. Let insert_P be an operator defined inductively on RA(Datalog) programs as follows:

- $\mathit{insert}_P(Q) = Q$ if Q is a Datalog program;
- $\mathit{insert}_P(g; h) = \mathit{insert}_P(g);\ P\ ; \mathit{insert}_P(h)$;

- $insert_P(g^*) = (insert_P(g))^*$.

Clearly, $P[g]$ is equivalent by definition to $P;insert_P(g);P$ for every RA(Datalog) program g. Next, let *union* be the operator from RA(Datalog) programs to Datalog programs defined by $union(Q) = Q$ if Q is a Datalog program, $union(g;h) = union(g) \cup union(h)$, and $union(g^*) = union(g)$ (thus, *union* collects all Datalog rules occurring in the RA(Datalog) program). It is easy to see that all RA*(Datalog) programs g^* are equivalent to $(union(g))^*$. Now $P[e^*]$ is equivalent to $P;insert_P(e^*);P$, and $insert_P(e^*)$ equals $(insert_P(e))^*$ which is equivalent to $(union(insert_P(e))^* = (union(e) \cup P)^*$. Thus, $P[e^*]$ is equivalenbt to $P;(union(e) \cup P)^*;P$. Similarly, $P[f^*]$ is equivalent to $P;(union(f) \cup P)^*;P$. Since $(union(e))^*$ and $(union(f))^*$ are equivalent, $(union(e) \cup P)^*$ and $(union(f) \cup P)^*$ are equivalent, as are P; $(union(e) \cup P)^*;P$ and $P;(union(f) \cup P)^*;P$. So $P[e^*]$ and $P[f^*]$ are equivalent. □

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 $M_{<x,y>}$ 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 $x, y \in S$, $M_{<x,y>}$ is uniformly equivalent to $M'_{<f(x),f(y)>}$, that is: for $\mathbf{I}$ over $\mathbf{R}$, $M_{<x,y>}$ halts in state y with output $\mathbf{J}$ over $\mathbf{R}$ iff $M'_{<f(x),f(y)>}$ 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:

Theorem 7.2 Every trigger machine is weakly semantically consistent.

Thus, trigger equivalence of two external programs guarantees that the final effects of the programs in conjunction with any trigger program are always the same.

In particular, the weak semantic consistency of trigger programs 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, and these portions of code are optimized independently of each other. For example, optimization cannot be based on data flow analysis that goes across trigger invocation boundaries. This is illustrated in the following.

Example: Consider the external program

$$\begin{array}{lcl} R & := & R \cap S\ ; \\ Q & := & R \cup S \end{array}$$

Triggering occurs between the two commands. One might be tempted to replace the program by the folowing simpler program:

$$\begin{array}{lcl} R & := & R \cap S\ ; \\ Q & := & S \end{array}$$

since after execution of the first instruction, $R \subseteq S$ and so $R \cup S = S$. Note that the modified code does not cross trigger invocation boundaries. However, the analysis propagates information across invocation boundaries and is incorrect: the first external program is *not* trigger equivalent to the second. It is easily seen that the two programs have different effects in conjunction with some trigger programs. □

8 Conclusions

The paper initiated a formal investigation of the semantics and expressiveness of active databases. First, we introduced an abstract model based on relational machines, and used it as a framework to formally articulate some basic concepts relating to the semantics and expressiveness of active

databases. The model captures the interaction of external programs with the trigger system; this interaction lies at the core of the active database paradigm. We were able to obtain a number of results concerning the power of various abstract languages for triggers and for external programs.

Two abstract languages emerge as central: *while* (with its rule-based variant Datalog$^{\neg\neg}$) subsumes most active database prototypes. And, $while_N$ (or the relational machine) subsumes all prototypes we considered. Thus, abstract languages such as *while* and $while_N$ can serve as a yardstick for comparing different active database systems.

Although the abstract framework of relational machines is a very useful high-level model, it remains far from the programming style of real active databases. To be able to understand their programming features and computational paradigm in more detail, we next developed a formal framework that retains the main aspects of that style, and within which various prototypes can be formally described. In particular, we provided formal "snapshots" of ARDL, HiPAC, Postgres, Starburst, and Sybase, subject to some unifying assumptions spelled out in the paper. The results highlight the interplay of various programming features of active databases and their impact on expressive power and complexity. Thus, we examined the power of immediate triggering and deferred triggering, the mix of the two, the impact of using different types of delta relations, of various queuing disciplines, etc. We were able to provide a rather complete classification of the above prototypes with respect to their expressiveness (this is summarized in Figure 1). The results provide considerable insight into the computational paradigm of active databases.

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.

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

[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.

[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 Intl. 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 Intl. 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.

[CBB+89] S. Chakravarthy, B. Blaustein, A. Buchmaann, M. Carey, U. Dayal, D.Goldhirsch, M. Hsu, R. Jauhari, R. Ladin, M. Livny, D. McCarthy, R. McKee, and A. Rosenthal. Hipac: a research project in active time-constrained databases management. Technical report, Xerox Advanced Information Technology, July 1989.

[Coh86] D. Cohen. Programming by specification and annotation. In *Proc. of AAAI*, 1986.

[CTF88] M. A. Casanova, L. Tucherman, and A. L. Furtado. Enforcing inclusion dependencies and referential integrity. In *Proc. of Intl. 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 Intl. Conf. on Very Large Data Bases*, 1990.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 577–589, 1991.

[C71] S.A. Cook. Characterizations of Pushdown Machines in Terms of Time-Bounded Computers. *J. of the ACM*, Vol. 18, No. 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.

[Don88] G. Dong. On the composition and decomposition of datalog program mappings. In *Proc. of Intl. Conf. on Database Theory*, pages 87–101, 1988.

[DT92] G. Dong and R. Topor. Incremental evaluation of datalog queries. In *Proc. of Intl. 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 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.

[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.

[Im86] N. Immerman. Relational queries computable in polynomial time. *Inf. and Control*, 68:86–104, 1986.

[IL90] N. Immerman and E.S. Lander. Describing graphs: a first-order approach to graph cannonization. In *Complexity theory retrospective*, ed. A. Selman, pages 59-81, Springer-Verlag, 1990.

[Kan91] P. C. Kanellakis. Elements of relational database theory. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1074–1156. Elsevier, 1991.

[K91] P.G. Kolaitis. The Expressive Power of Stratified Logic Programs. Information and Computation 90:1, pages 50-66, 1991.

[KV92] P.G. Kolaitis and M.Y. Vardi. Fixpoint logic vs. infinitary logic in finite-model theory. In *Proc. Symp. on Logic in Computer Science*, pages 46-57, 1992.

[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 Intl. 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 Intl. Conf. on Very Large Data Bases*, pages 315–326, 1992.

[Sto86] M. Stonebreaker 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.

[Var82] M.Y. Vardi. The complexity of relational query languages. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 137–146, 1982.

[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.

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