A Logical Approach to Cooperative Information Systems

Elisa Bertino Barbara Catania

Dipartimento di Scienze dell'Informazione, University of Milan, Italy

Vincenzo Gervasi Alessandra Raffaetà

Dipartimento di Informatica, University of Pisa, Italy

Abstract

"Cooperative information system management" refers to the capacity of several computing systems to communicate and cooperate in order to acquire, store, manage, query data and knowledge. Current solutions to the problem of cooperative information management are still far from being satisfactory. In particular, they lack the ability to fully model cooperation among heterogeneous systems according to a declarative style. The use of a logical approach to model all aspects of cooperation seems very promising.

In this paper, we define a logical language able to support cooperative queries, updates and update propagation. We model the sources of information as deductive databases, sharing the same logical language to express queries and updates, but containing independent, even if possibly related, data. We use the Obj-U-Datalog [5] language to model queries and transactions in each source of data. Such language is then extended to deal with active rules in the style of Active-U-Datalog [4,22], interpreted according to the PARK semantics proposed in [23]. By using active rules, a system can efficiently perform update propagation among different databases. The result is a logical environment, integrating active and deductive rules, to perform update propagation in a cooperative framework.

Key words: Deductive databases, Active rules, Heterogeneous databases, Cooperation.

1 Introduction

The evolution of current information processing systems towards larger and more heterogeneous systems has made evident the strong need for systems

and tools enabling cooperative information system management. Cooperative information system management refers to the capability of several computing systems to communicate and cooperate in order to acquire, store, manage, query data and knowledge. Many application environments, such as workflow management systems, telecommunications network management, digital libraries, health-care provisioning and monitoring, strongly need tools enabling cooperative flows of information among their data management systems.

The development of a cooperative information management system entails addressing different problems, ranging from the differences in hardware/software platforms, to heterogeneity in the database management systems (DBMS), to semantic data heterogeneity, to operational issues such as update propagation and consistency maintenance for related information. Solutions to these problems are provided by efforts in different areas. In particular, hardware/software heterogeneity issues are investigated in the areas of communication networks and operating systems. DBMS heterogeneity (intended as heterogeneity in data models and in query and manipulation languages of the various DBMS) and semantic data heterogeneity are addressed by the area of multidatabase systems [8,40]. Operational issues have been addressed by research on transaction models and mechanisms [19]; also active rules have been applied to the specific issue of update propagation [11,18,26]. A logical approach to enforcing integrity constraints in an heterogeneous environment has been proposed in [12], where temporal logic is extended to explicitly model events and distributed rules. On the contrary, in [44] consistency is not enforced, and different databases are allowed to contain inconsistent information. A supervisory database specifies rules to deal with conflicts as a particular case of integration of information coming from multiple knowledge bases.

Current solutions to the problem of cooperative information management are, however, still far from being satisfactory. In particular, they lack the ability to fully model cooperation among heterogeneous systems according to a declarative style. In most cases, solutions are based on *ad-hoc* application programs acting as bridges among the various systems. In such a situation, understanding which information and actions a system requires from other systems is very difficult, especially when the number of involved systems is large. If a system is added or removed, new bridges must be defined and analyzed. Moreover, reasoning about the cooperative system as a whole is very difficult. This situation calls for a declarative approach allowing one to fully model all aspects of cooperation and to provide the basis on which properties of cooperative information systems can be proved. To this purpose, the use of a logical approach seems very promising.

In order to model cooperative information management using a logical approach, we should be able to model:

- different sources of data, intended as different databases storing (possibly related) data and intensional knowledge on data;
- cooperative query execution, intended as the ability of several data management systems to collectively provide information to answer user queries;
- updates on data and update propagation among the various data sources.

In this paper, we define a logical language able to support cooperative queries, updates and update propagation. We model the sources of information as deductive databases, sharing the same logical language to express queries and updates, but containing independent, even if possibly related, data.

Several approaches have been proposed integrating logic and updates (see for example [1,2,6,7,13,14,32,34,35,38,45]). The language we propose is based on the U-Datalog language [6], and extends it with support for active rules and with the ability to model heterogeneous databases. U-Datalog has been introduced with the aim of providing a set-oriented logical update language, guaranteeing update parallelism in the context of a Datalog-like language. In U-Datalog, update atoms appear in rule bodies, thus integrating updates and queries and exploiting the power of Datalog queries to select the data to be modified. The execution of a goal (also called a *transaction*) in U-Datalog is based on a deferred semantics, by which several updates are generated from predicate evaluation, but not immediately executed; rather, they are collected and are executed only at the end of the query-answering process. In U-Datalog, updates are expressed by using constraints. For example, $+p(a)$ states that in the new state $p(a)$ must be true whereas $-p(a)$ states that in the new state $p(a)$ must be false. Each atom solution generates a set of updates.

The language obtained by extending U-Datalog with active rules and support for heterogeneous databases is called *Heterogeneous U-Datalog* (HU-Datalog for short). We define a *HU-Datalog system* as a collection of heterogeneous databases and a set of active rules operating on such databases. In particular, a HU-Datalog system provides the following features:

(1) *Multiple sources of data and cooperative query execution.* The basic model for each deductive database is a U-Datalog database. As such, it consists of an extensional database (i.e., a set of facts) and an intensional database represented by a set of U-Datalog rules.

The use of U-Datalog provides the ability to model, not only queries, but also updates inside each database. However, U-Datalog alone does not support cooperation among several databases. Each database has no knowledge about the others. In order to enable cooperative query execution, we model each U-Datalog database as an Obj-U-Datalog *object* [5]. Obj-U-Datalog has been defined as an object-based extension of U-Datalog. Each Obj-U-Datalog object has a state (a set of facts, i.e. an extensional database) and a set of methods (an intensional database).

Objects cooperate through message passing. To this purpose, U-Datalog rules have been extended to request the evaluation of an atom in a different object, i.e. in our case, in a different database. This means that during the evaluation process a database may require the evaluation of a subquery to another database. Message passing is expressed through labeled atoms in rule bodies.

By modeling each database as an Obj-U-Datalog object, information deriving from different sources can be retrieved and updated in a homogeneous way, thus providing cooperative query and update execution. In particular, the use of Obj-U-Datalog makes each source of data aware of the other sources of data belonging to the considered environment.

It is important to note that, if we ignore update atoms, Obj-U-Datalog can be interpreted as an amalgamated knowledge base, as described in [44], based on the classical truth values. The main novelty of our approach is the extension of such amalgamated knowledge bases to deal with updates and, as we will see in the following, actions.

(2) *Active rules.* In order to exhibit a reactive behavior, active rules in the style of Active-U-Datalog [4,22] are included in the language, according to the PARK semantics [23].

The PARK semantics has been designed with the intent of overcoming the limitations of most previously defined semantics for active rules. In particular, given a set of ECA (Event-Condition-Action) rules, i.e. rules of the form "ON event IF condition THEN action", the PARK semantics satisfies several properties. First of all, it is non-ambiguous, i.e., it always guarantees execution confluence. Moreover, it is flexible with respect to conflict resolution. A conflict is a situation where two or more active rules can be fired and one of these rules requires the insertion of an atom a in the database, whereas at least one of the others requires the deletion of a from the database. A conflict resolution policy is a method to determine which actions should be executed in presence of a conflict and which others should be suppressed. Under the PARK semantics, the conflict resolution policy can be chosen according to specific application requirements. A fixpoint semantics is used to determine the result of the application of a set of active rules. The fixpoint semantics has been chosen because it has a clear mathematical foundation and can be directly implemented. The proposed semantics guarantees the termination of the evaluation process. As it has been pointed out in [23], all other proposed semantics fail to satisfy at least one of these important requirements.

Active-U-Datalog rules are *local* in that each rule refers to a single data source. A rule is triggered when specific events (updates on the considered database) occur. The action is the execution of a set of updates on the considered data source; thus, local active rules can modify only the database into which they are defined.

(3) *Update propagation.* To support update propagation among heterogeneous databases, *global* active rules are used. They support consistency

maintenance among different data sources. These rules can fire upon the occurrence of complex events related to all the databases in a HU-Datalog system, can verify conditions spanning all the databases, and can perform actions consisting of updates against all the databases.

The use of active rules for update propagation in the context of heterogeneous databases has been addressed by a few other proposals [11,18,26]. The main difference of our proposal is that the use of the PARK semantics to integrate active and deductive rules makes the approach much more flexible with respect to the problem of conflict resolution. Often, in active database systems, conflicts are solved by assigning priority to rules [27,42,47]. On the contrary, the PARK semantics allows the application programmer to choose the best conflict resolution policy to apply in a particular case. In solving a conflict, information about the structure and the current state of the system can be considered. This is particularly important in a cooperative framework, where update propagation often depends on information distributed among various databases. For example, the resolution policy could include queries on the state of a different database before deciding how to solve a conflict, a behavior that cannot be simulated by using priorities.

Our approach also integrates active and deductive rules. With respect to active rules, the field has produced various results, both commercial [3,17,29,39,41] and academic [15,16,21,25,27,36,42,43,46,47] (the latter usually being more flexible than the former). The semantics we propose for active rules differs from the semantics of most other proposals in that it guarantees termination, polynomial complexity, and confluence; almost all the other proposals fail in satisfying at least one of these requirements. In particular, in order to assign a clear semantics to active rules, several works proposed to integrate active rules in a deductive framework. In giving a semantics to these active-deductive languages, two main approaches emerged. The first one is based on *unifying* the different paradigms under a common semantics (often by using compilation techniques) [10,33,37,49]. The second one is based on *integrating* specific, different semantics [4,20,22]. In this work, we follow the latter approach. The use of the PARK semantics allows one even in this case to handle updates generated by deductive rules and updates generated by active rules in a uniform way. For example, with respect to U-Datalog and Obj-U-Datalog [5,6], there is no need to define *a priori* the behavior to be taken when conflicts arise. Thus, our approach can also be seen as an extension of U-Datalog to deal with multiple policies for conflict resolution. As far as we know, no other approach of this kind has been proposed yet in the context of deductive databases.

The paper is organized as follows. In Section 2, we briefly recall the main features of U-Datalog and Object-U-Datalog, on which our proposal is based. Sections 3 and 4 describe respectively the syntax and semantics of HU-Datalog, illustrating them with several examples. In particular, Section 4 introduces an

extension of the PARK semantics to deal with multi-theories and communicating objects. Finally, Section 5 outlines some conclusions, discusses relations between our work and recent work on intelligent agents, and presents future research directions.

2 Overview of U-Datalog and Object-U-Datalog

In this section, we informally present the basic notions of U-Datalog and Obj-U-Datalog. We refer the reader to [5,6] for an extensive description of these languages. Moreover, we assume that the reader is familiar with the basic logic programming concepts [31].

A U-Datalog program (or database) consists of an extensional database *EDB*, that is, a set of ground atoms, and an intensional database *IDB*, that is, a set of rules of the following form:

$$H \leftarrow U_1, \ldots, U_i, B_{i+1}, \ldots, B_n. \qquad n \geq 1$$

where U_j, $j = 1, ..., i$, are *update atoms* and H, B_k, $k = i+1, ..., n$, are atoms, in the usual logic programming sense. An update atom is an atom prefixed by the symbol $+$, to denote an insertion, or by the symbol $-$, to denote a deletion. The intuitive meaning of such a rule is "if $B_{i+1}, \ldots, B_n$ are true, then H is true and the updates $U_1, \ldots, U_i$ are requested". Predicates defined in the extensional database and predicates defined in the intensional database are disjoint.

A query is a rule with no head. Since, as we will see, U-Datalog queries may generate updates, a U-Datalog query is also called *simple transaction*. A *complex transaction* is a sequence of simple transactions, denoted by $T_1; \ldots; T_n$.

For simplicity, it is assumed that U-Datalog databases are safe. A rule is *safe* if each variable in the head occurs in a non-update atom in the body. Often this condition is relaxed, assuming that the rule is *safe with respect to a query*; in this case, it should be safe when the head is unified with constants in the query.

Example 1 *Consider the following U-Datalog program, related to a student database:*

```
EDB: student(john).     student(mary).      student(frank).
     exam(engl).        exam(math).         exam(phys).

IDB: pass(S,E) ← student(S), exam(E), +passed(S,E).
     leave(S) ← −student(S).
```

This database contains information about exams (`exam` *predicate), students (*`student` *predicate) and the exams they passed (*`passed` *predicate). The database can be modified by the predicates* `pass` *and* `leave` *that, respectively, add the fact that a student passed an exam and delete the student when he/she leaves the school. The first rule is safe, but the second rule is not. However, the second rule is safe with respect to the transaction* `leave(john)`*. Of course, the second rule can be replaced with* `leave(S)` ← `student(S), −student(S)` *which is always safe.* ◇

The semantics of U-Datalog is essentially given in three steps. The first one, called *marking phase*, generates a set of solutions for a given transaction. Each solution contains a set of bindings and a set of updates. These updates are executed only in the *update phase*, if they are *consistent*. A set of updates is consistent if it does not require the insertion and the deletion of the same fact. If the set of updates is not consistent the transaction is *aborted* and all the updates are discarded. The third step is related to the execution of complex transactions. In this case, the extensional database is updated after each transaction execution. If a transaction aborts, the entire complex transaction aborts as well.

Example 2
Consider again the U-Datalog program introduced in Example 1. The transaction `pass(john,math)` *causes the insertion of the fact* `passed(john,math)` *in the database. In this case, bindings for variables contained in the head of the rule are established by the transaction. Now consider the transaction* `pass(X,Y)`*. In this case, a set of facts is inserted, in particular all the facts of the form* `passed(s,e)`*, where* `student(s)` *and* `exam(e)` *belong to EDB. A set of bindings of the form* $X =$ `s`, $Y =$ `e` *is also returned to the user. Note that updates are executed since they are consistent.*

Now suppose that students are associated with at least one tutor. Moreover, suppose that two tutors, for some particular reason, need to be assigned each the students of the other. The rules to do that are the following:

`change(`T_1, T_2`)` ← `-tutor(S,`T_1`), +tutor(S,`T_2`), tutor(S,`T_1`).`
`change(`T_1, T_2`)` ← `-tutor(S,`T_2`), +tutor(S,`T_1`), tutor(S,`T_2`).`

Now consider the transaction `change(mark,victor)`*. If in the database there exists at least one student -say* `john`*- having both* `mark` *and* `victor` *as tutors, two pairs of inconsistent updates are generated from the previous rules (respectively* `+tutor(john,mark)`,`-tutor(john,mark)` *and* `+tutor(john,victor)`, `-tutor(john,victor)`*). Thus, an abort is returned and the extensional database is left unchanged.* ◇

Obj-U-Datalog [5] has been defined starting from the consideration that a U-Datalog database $EDB \cup IDB$ can be seen as an object where EDB is the

object state and *IDB* represents the set of methods to manipulate (i.e., query and update) such state. Thus an Obj-U-Datalog program consists of a set of object databases, each of which is a U-Datalog database, extended so that distinct databases can refer to each other. Such references are supported by a labeling mechanism. Thus, a method becomes a rule of the form:

$$H \leftarrow U_1, \ldots, U_i, B_{i+1}, \ldots, B_w, db_1 : B_{w+1}, \ldots, db_p : B_z, X_1 : B_{z+1}, \ldots, X_q : B_r.$$

where U_k, $k = 1, ..., i$, are update atoms, B_j, $j = i+1, ..., w$, are atoms whose predicates are defined in the object database in which the rule is defined, B_h, $h = w+1, ..., r$, are atoms defined in other object databases. Such object databases are referred to through the use of *labels*. In particular, db_m, $m = 1, ..., p$, are ground labels whereas X_s, $s = 1, ..., q$, are variables to which ground labels should be assigned during the evaluation.

Example 3 *Suppose that in the school considered in Example 1 there is a library. Library information is maintained in a database which is different from the one containing information about the student. Moreover, a third database stores information about the teachers. The library would like to use some information about students and teachers in order to automatize loans. To accomplish that, the following Obj-U-Datalog program can be defined:*

```
school:: student(john).          student(mary).      student(frank).
         exam(engl).             exam(math).         exam(phys).
         passed(john,engl).      passed(john,math).
         passed(mary,phys).      passed(frank,engl).

         pass(S,E) ← student(S),exam(E),+passed(S,E).
         leave(S) ← student(S),−student(S).

teach::  prof(william).          prof(isaac).        prof(eliza).
         teaches(engl,william).  teaches(math,isaac).
         teaches(phys,isaac).    teaches(cs,eliza).

lib::    book(hamlet).           book(principia).
         sect(engl,hamlet).      sect(phys,principia).
         other_user(helen).      other_user(andrea).
         loan(hamlet,john).      loan(principia,frank).

         user(X) ← school:student(X).
         user(X) ← teach:prof(X),teach:teaches(E,X),school:exam(E).
         user(X) ← other_user(X).
         deny_loan(B,U) ← user(U),loan(B,Y).
         deny_loan(B,U) ← book(B),request(X,U).
         return(B,U) ← loan(B,U),−loan(B,U).
```

Three object databases have been defined, labeled `school`, `teach`, *and* `lib`. *Predicates defined in the database* `school` *have the same meaning of predi-*

cates introduced in Example 1. The database `teach` *contains information about teachers and courses they teach; it does not include any deductive rules. The library database* `lib` *contains information about users that are neither students nor teachers (*`other_user` *predicate), books (*`book` *predicate), the section of the library to which the book is assigned (*`sect` *predicate), and about who borrowed a book (*`loan` *predicate). It also contains rules to specify library users, and the policy to deny a loan and to return a book. In particular, a loan is denied if it concerns a book already on loan, or if it is from a user who was requested to return back a book he borrowed (and has not yet complied).*

Notice how rules in the library database use information about students and exams, stored in the school database, and about teachers and their courses, stored in the teacher database, in order to manage loans. In particular, a user can be either a student of the `school` *(first rule defining predicate* `user`*) or a professor (*`teach:prof(X)`*) teaching a course of the* `school` *(second rule defining predicate* `user`*) or a person recorded as user inside the database of the library itself (third rule defining predicate* `user`*).* ◇

As for U-Datalog programs, the semantics of Obj-U-Datalog programs is given in three steps. During the marking phase, however, objects may request the evaluation of some atoms in other objects (specified by labels), thus requiring some form of *context switching*, to solve the query in another object.

3 Syntax of HU-Datalog

In Example 3, we have informally shown how Object-U-Datalog supports interaction among several databases, as long as each of them knows (at least partially) the structure of the others. This condition is too restrictive when applied to the case of completely heterogeneous databases; hence in this case, a different mechanism for integration should be used. This mechanism can be provided by local and global active rules. However, explicit interaction via message passing must also be guaranteed, in order to preserve the expressive power of Object-U-Datalog for those cases in which some databases are indeed tightly coupled. In the following, we present a language called *Heterogeneous U-Datalog* (HU-Datalog for short) having all the previous characteristics. HU-Datalog is thus an extension of Object-U-Datalog.

We consider a many-sorted signature $\Sigma = \{\Sigma_{db}, \Sigma_v\}$, containing only constant symbols. Σ_{db} is the set of database identifiers, whereas Σ_v is the set of constant value symbols. Sets Σ_{db} and Σ_v are disjoint. We consider moreover a set of predicate symbols Π, partitioned into extensional predicate symbols Π^e, intensional predicate symbols Π^i and update predicate symbols Π^u, defined as $\Pi^u = \{+p, -p \mid p \in \Pi^e\}$.

A family of sets of variable symbols for each sort $V = \{V_{db}, V_v\}$ is considered. Terms are defined as usual for each sort of our language, being a term either a constant or a variable. We denote with $Term_{db}$ the set $\Sigma_{db} \cup V_{db}$, and with $Term_v$ the set $\Sigma_v \cup V_v$. We denote with (Π, Σ, V)-atom an atom whose predicate belongs to Π and whose terms are in $\Sigma \cup V$. The notion of substitution is refined to take into account the many-sorted language. A substitution is a pair of functions $\theta = \{\theta_{db}, \theta_v\}$, $\theta_{db} : V_{db} \rightarrow Term_{db}$, $\theta_v : V_v \rightarrow Term_v$, which maps each variable to a term of the appropriate sort.

Update atoms are extensional atoms prefixed by $+$, to denote insertion, and by $-$, to denote deletion. Cooperation among databases in the system is represented by labeled atoms of the form $db : p(\tilde{t})$, where $db \in Term_{db}$, meaning that $p(\tilde{t})$ must be solved in db.

In the following, using the concepts defined above, we formally introduce the notion of HU-Datalog database, HU-Datalog system, and transaction.

Definition 4 (HU-Datalog database)
A HU-Datalog database *$DB = EDB \cup IDB \cup AR$ consists of an extensional database EDB, an intensional database IDB and a set of active rules AR. The EDB is a set of ground extensional atoms, called the* state *of DB.*

The IDB is a set of deductive rules *of the form*

$$H \leftarrow U_1, \ldots, U_i, B_{i+1}, \ldots, B_w, db_1 : B_{w+1}, \ldots, db_p : B_z, X_1 : B_{z+1}, \ldots, X_q : B_r.$$

where

(1) H is a (Π^i, Σ, V)-atom;
(2) $U_1, \ldots, U_i$ are (Π^u, Σ, V)-atoms, constituting the update part of the rule;
(3) $B_{i+1}, \ldots, B_w$ are $(\Pi^i \cup \Pi^e, \Sigma, V)$-atoms, constituting the unlabeled part of the condition (that is, they refer to the database where the rule is defined);
(4) $db_1 : B_{w+1}, \ldots, db_p : B_z$ are labeled $(\Pi^i \cup \Pi^e, \Sigma, V)$-atoms referring to specific databases;
(5) $X_1 : B_{z+1}, \ldots, X_q : B_r$ are labeled $(\Pi^i \cup \Pi^e, \Sigma, V)$-atoms referring to databases that are not yet specified;
(6) $X_1, \ldots, X_q$ are variables in V_{db} and must appear as arguments of an extensional atom in $B_{i+1}, \ldots, B_z$.

The update part $(U_1, \ldots, U_i)$ and the set of conditions $(B_{i+1}, \ldots, B_w, db_1 : B_{w+1}, \ldots, db_p : B_z, X_1 : B_{z+1}, \ldots, X_q : B_r)$ cannot be both empty.

The AR is a set of rules of the form

$$E_1, \ldots, E_i, B_{i+1}, \ldots, B_w, db_1 : B_{w+1}, \ldots, db_p : B_z, X_1 : B_{z+1}, \ldots, X_q : B_r \rightarrow U_1, \ldots, U_k.$$

where $E_1, \ldots, E_i$ is the event part (E_j is a (Π^u, Σ, V)-atom, $j = 1, ..., i$), $(B_{i+1}, \ldots, B_w, db_1 : B_{w+1}, \ldots, db_p : B_z, X_1 : B_{z+1}, \ldots, X_q : B_r)$ is the condition part (B_j is a $(\Pi^i \cup \Pi^e, \Sigma, V)$-literal, $j = i+1, ..., r$), and B_j can also be a negative atom (denoted with $\neg p(\tilde{t})$) where negation is understood as negation as failure. $U_1, \ldots, U_k$ is the action part (U_j is a (Π^u, Σ, V)-atom, $j = 1, ..., k$), which cannot be empty. We require two safety conditions for active rules: each variable occurring in a rule head should also occur in the body of the same rule and each variable occurring in a negated literal in the rule body must also occur in some positive literal in the rule body. □

The intuitive meaning of a deductive rule is: "if $B_{i+1}, \ldots, B_w$ are true in the database where the rule is defined, B_{w+1} is true in db_1, ..., B_z is true in db_p, B_{z+1} is true in the database to which X_1 is instantiated, ..., B_r is true in the database to which X_q is instantiated, then H is true and, as a side effect, the updates $U_1, \ldots, U_i$ are requested". These updates are local, that is, they change the state of the database in which the rule itself is defined. To ensure encapsulation, labeled updates, i.e. updates to be executed on another database, are not allowed in deductive rules. In this way, the state of a database can only be modified through its public interface, which is the set of its intensional predicates. On the contrary, the knowledge contained in the state can be freely queried by any other database.

Databases cooperate by using labeled atoms to request the evaluation of the atom in the context of the database identified by the label. Such label can be either a constant (providing a static communication channel) or a variable (providing a dynamic communication channel). In the latter case, the label identifying the cooperating database is computed by using data contained in some database. Such an approach gives high flexibility for setting up complex, dynamic communication structures.

While deductive rules give deductive power to our framework, active rules allow the system to autonomously react to the current (possibly inconsistent) state and to take appropriate actions to ensure desired properties with respect to the final state. The intuitive meaning of an active rule is: "If the events $E_1, \ldots, E_i$ occur and $B_{i+1}, \ldots, B_w$ are true in the database where the rule is defined, B_{w+1} is true in db_1, ..., B_z is true in db_p, B_{z+1} is true in the database to which X_1 is instantiated, ..., B_r is true in the database to which X_q is instantiated, then execute actions $U_1, \ldots, U_k$". These active rules are only triggered by local updates and only modify the state of the database which they belong to, since both *events* and *actions* are update atoms referring to the database itself. We call them *local* active rules to distinguish them from the *global* ones associated with the entire system (see Definition 5). Databases can use local active rules to enforce certain properties of the data, reacting to consistency-breaking changes by updating other data, *regardless of the source* of changes. Local active rules also simplify the deductive part, allowing one to

centralize certain policies in a single place instead of scattering them between several rules.

Based on the definition of HU-Datalog database, a HU-Datalog system can be defined as follows.

Definition 5 (HU-Datalog system)
A HU-Datalog system $\Xi = \langle \{db_1 :: DB_1, \ldots, db_s :: DB_s\}, AR \rangle$ *consists of a set of HU-Datalog databases* $DB_1, ..., DB_s$*, respectively identified by* $db_1, \ldots, db_s$ *(*$db_i \in \Sigma_{db}$*), and of a set of active rules* AR *of the form:*

$$db_{k_1} : E_1, \ldots, db_{k_i} : E_i, db_{h_1} : B_{i+1}, \ldots, db_{h_p} : B_z, X_1 : B_{z+1}, \ldots, X_q : B_r \rightarrow db'_1 : U_1, \ldots, db'_k : U_k, X'_1 : U_{k+1}, \ldots, X'_t : U_m.$$

where $db_{k_1} : E_1, \ldots, db_{k_i} : E_i$ *is the event part (*E_j *is a* (Π^u, Σ, V)*-atom,* $j = 1, ..., i$*),* $db_{h_1} : B_{i+1}, \ldots, db_{h_p} : B_z, X_1 : B_{z+1}, \ldots, X_q : B_r$ *is the condition part (*B_j *is a positive or negative* $(\Pi^i \cup \Pi^e, \Sigma, V)$*-atom,* $j = i+1, ..., r$*), and* $db'_1 : U_1, \ldots, db'_k : U_k, X'_1 : U_{k+1}, \ldots, X'_t : U_m$ *is the action part (*U_j *is a* (Π^u, Σ, V)*-atom,* $j = 1, ..., m$*) which cannot be empty. These rules must satisfy the safety conditions for active rules presented in Definition 4.*

We call state *of a HU-Datalog system* Ξ *the tuple* $\langle EDB_1, \ldots, EDB_s \rangle$*, where* EDB_i *is the extensional database of* db_i*.* □

In a HU-Datalog system $\Xi = \langle \{db_1 :: DB_1, \ldots, db_s :: DB_s\}, AR \rangle$ we distinguish a *deductive* part, consisting of the tuple of deductive databases, i.e. $\langle EDB_1 \cup IDB_1, \ldots, EDB_s \cup IDB_s \rangle$, and an *active* part, including the local active rules $\langle AR_1, \ldots, AR_s \rangle$ and the global ones AR.

The rules in AR differ from the local active rules in that all atoms, included events and actions, are labeled. The reason is that those rules refer to the entire system, as they enable update propagation among different sources of data, and they ensure consistency *across* databases. Since those rules are aware of the structure of each database, they maintain all semantics relationships among relations stored in different databases. If an update is executed with respect to one of such relation, the global rules ensure that the other databases are updated appropriately. Because of their nature, global active rules play the role of *mediators* [48] in integrating heterogeneous databases.

In order to ensure encapsulation, transactions to be executed in a HU-Datalog system cannot contain update atoms. However their execution may generate updates indirectly, because of the invocation of rules with update atoms in their bodies. Thus a transaction may contain two different kinds of atoms: labeled ones and unlabeled ones. Unlabeled atoms stand for the request of an atom refutation in all the databases composing the system, while labeled atoms are directed to a particular database. We do not restrict labels in transaction to

be constant as required in Obj-U-Datalog, thus allowing dynamic cooperation to be established also at transaction level.

Definition 6 (Transaction) *A* simple transaction *has the form*

$$B_1, \ldots, B_w, db_1 : B_{w+1}, \ldots, db_p : B_z, X_1 : B_{z+1}, \ldots, X_q : B_r.$$

where $B_1, \ldots, B_r$ *are* $(\Pi^i \cup \Pi^e, \Sigma, V)$*-atoms,* $db_1, \ldots, db_p$ *are database identifiers and* $X_1, \ldots, X_q$ *are variables in* V_{db} *that must appear as arguments of an extensional atom in* $B_1, \ldots, B_z$*.*

A complex transaction T *is a sequence of simple transactions* $T_1; \ldots; T_k$*.* □

It should be clear that a transaction provides different functions: the query function, in that it returns a set of bindings, and the update function with a transactional behavior [24]. As we will see in Section 4.3, the transactional behavior ensures that all the updates are executed or, in case of ungroundness, none of them is executed. We always assume that our rules are safe with respect to a transaction.

Example 7 *Consider again to the problem of integrating the student database of a school and the loan database of a library. This time, we consider* active *and fully* heterogeneous *databases. In particular, each database knows nothing about the structure of the other, as there is no message passing between them (compare this assumption with the ones that the database* `lib` *made about the structure of databases* `school` *and* `teach` *in Example 3).*

```
school:: student(john).        student(mary).        student(frank).
         exam(engl).           exam(math).           exam(phys).
         passed(john,engl).    passed(john,math).
         passed(mary,phys).    passed(frank,engl).

         pass(S,E) ← student(S), exam(E), +passed(S,E).
         leave(S) ← student(S), −student(S).

         −student(S), passed(S,E) → −passed(S,E)

lib::    user(john).           user(mary).
         user(frank).          user(pat).
         book(hamlet).         book(principia).
         sect(engl,hamlet).    sect(phys,principia).
         loan(hamlet,john).    loan(principia,frank).

         deny_loan(B,U) ← request(X,U), book(B).
         deny_loan(B,U) ← loan(B,Y), user(U).
         return(B,U) ← loan(B,U), −loan(B,U).

         −user(U), loan(B,U) → +request(B,U).
         −loan(B,U), request(B,U) → −request(B,U).
```

Predicates defined in the databases `school` *and* `lib` *have the same meaning of predicates defined in Example 3. However, now active rules are included in each database. In particular, the only active rule in* `school` *takes care of removing from the database any information related to the exams passed by a student when the student is removed from the database (whatever is the cause of such removal).*

In `lib`*, two active rules take care of requesting all the books someone has on loan when he/she is revoked his/her user status, and of removing any pending request for a book when it is returned to the library.*

We want to integrate these two databases so that when a student joins the school, he/she is automatically considered a user of the library, and when he/she leaves the school, he/she is removed from the users of the library. Moreover, we want that when a student passes an exam, the library is able to require back all the books the student has on loan from the section corresponding to the passed exam. This behavior can be obtained by adding the following active rules to the global AR set:

`school:+student(S)` $\rightarrow$ `lib:+user(S).`
`school:−student(S)` $\rightarrow$ `lib:−user(S).`
`school:+passed(S,E),lib:loan(B,S),lib:sect(E,B)` $\rightarrow$ `lib:+request(B,S).`

When Frank passes physics, this information is recorded in the school database by executing the transaction `school:pass(frank,phys)`*. The first deductive rule of* `school` *causes the insertion of* `passed(frank,phys)` *in the school database. Such insertion in turn fires the last global active rule, whose conditions are met by* `lib:loan(principia,frank)` *and* `lib:sect(phys,principia)`*. Therefore* `request(principia,frank)` *is added to the library database, disallowing any further loan to Frank until the book is returned. Should John leave the school, the second global active rule would fire, causing the library to remove him from the user list. As part of the local processing activity, the first active rule in the library database would fire, causing the insertion of* `request(hamlet,john)` *in that database.*

When a book is returned by a transaction `lib:return(book,user)`*, the last deductive rule in* `lib` *requires the deletion of the loan record. Such deletion fires the last local active rule in* `lib` *that removes any pending request for the book.*

As another example, consider the problem of a student who wants to move to a different school. We want that exams, passed in the school the student is leaving and common to both schools, are confirmed in the school where he/she is moving. Such information flow can be easily achieved by the following global ARs:

```
school:+move(S,T) → school:−student(S),T:+student(S).
school:+move(S,T),school:passed(S,E),T:exam(E) → T:+passed(S,E).
```

This solution works if the two schools have the same database schema. In Example 32 we will consider the case of two schools with different but related information. Notice also that the identifier of the database of the new school is obtained from the `move` *predicate and is dynamically used by means of the variable* T. ◇

4 Semantics

The semantics of a HU-Datalog system is given in three steps. In the first step, we compute the model of the deductive part and collect the set of bindings that satisfies the transaction, and the requested updates. This step corresponds to the marking phase in Obj-U-Datalog [5]. However, if the result of this step is a set of inconsistent updates, we do not abort the transaction as Obj-U-Datalog does. Instead, we solve the conflicting updates in the second step. The notion of inconsistency is here extended to sets of labeled updates. A set of labeled updates is *consistent* if it contains no opposite updates labeled by the same database identifier, i.e. $db{:}+r(\tilde{t})$ and $db{:}-r(\tilde{t})$.

In the second step, we compute the semantics of the active part of the system, according to the model and the updates collected in the first step. The result of this step is the set of consistent updates requested either from the deductive and/or the active part, in which any conflict has been solved by a parametric policy. Finally, we describe how the two semantics fit together and how we apply the computed updates to the extensional databases of the HU-Datalog databases, thus obtaining the new state of the system.

The observable property of the transaction consists of a set of bindings (the answer), the new system state, and the termination status (commit/abort) of the transaction itself. Actually, our transactions always commit because of the conflict resolution policy of the active part, that solves any inconsistency. Although our semantics never produces aborts by itself, we prefer to keep the flag which indicates the success/failure of a transaction to be consistent with the semantics of U-Datalog and Obj-U-Datalog, and to allow for a smooth extension of our semantics to situations where transactions can go wrong (e.g., implementation-related failures).

The semantics of the deductive part is defined as the collection of the semantics of independent databases, as proposed in [5]. Each database interacts with another database only through explicit *context switches*, that is, through requests for the evaluation of a subquery sent to the other database. We define a bottom-up semantics for the marking phase, based on a *parallel* immediate consequence operator. This semantics is based on a composite structure for interpretation in which all the databases in the HU-Datalog system are interpreted simultaneously. For each database, its interpretation is a subset of the set of *constrained* atoms, $\mathcal{B}^{\leftarrow}$, defined as follows:

$$\mathcal{B}^{\leftarrow} = \{H \leftarrow \overline{U} \mid H \text{ is a } (\Pi^i \cup \Pi^e, \Sigma, V)\text{-atom}, \overline{U} \text{ is a set of labeled } (\Pi^u, \Sigma, V)\text{-atoms } \}^{1}$$

The presence of the constrained atom $H \leftarrow \overline{U}$ in the interpretation means that H is true and that its evaluation requires the execution of $\overline{U}$. All updates in $\overline{U}$ are labeled and their labels refer to the database on which they have to be executed. The labels are constants, whereas the atoms are not necessarily ground.

At this point, an interpretation for a set of HU-Datalog databases $DB_1, \ldots, DB_s$, identified respectively by $db_1, \ldots, db_s$, is a tuple of sets $\langle I(db_1), \ldots, I(db_s)\rangle$ where each $I(db_i)$ is a subset of $\mathcal{B}^{\leftarrow}$that interprets the associated database db_i. The domain $\mathcal{I} = \wp(\mathcal{B}^{\leftarrow})^s$ of all interpretations[2] endowed with the usual order on tuples, induced by the subset order, i.e.

$$\langle I_1, \ldots, I_s\rangle \sqsubseteq \langle I'_1, \ldots, I'_s\rangle \quad \textit{if and only if} \quad I_i \subseteq I'_i \quad \textit{for all } i = 1, \ldots, s$$

is a lattice. Notice that since $\mathcal{B}^{\leftarrow}$ is finite, $\mathcal{I}$ is finite.

The next definition formalizes the intuitive meaning of a rule from the deductive part of the system presented in Section 3. It differs from the consequence operator of Obj-U-Datalog because our resulting interpretation can include constrained atoms with inconsistent updates.

Definition 8 (Immediate consequence operator)
Given a HU-Datalog system $\Xi = \langle\{db_1 :: DB_1, \ldots, db_s :: DB_s\}, AR\rangle$*, the* immediate consequence operator $T_\Xi : \mathcal{I} \to \mathcal{I}$ *is defined as follows:*

$$T_\Xi(I) = \langle T_{db_1}(I), \ldots, T_{db_s}(I)\rangle$$

[1] As shorthand, a constrained atom $H \leftarrow$ is simply denoted by H itself.
[2] With D^s we denote the product $\underbrace{D \times \cdots \times D}_{s \text{ times}}$.

where $I \in \mathcal{I}$ and for each i, $i = 1, ..., s$, we have

$$
\begin{aligned}
T_{db_i}(I) = \{ A \leftarrow \overline{U} \mid\ & H \leftarrow U_1, \ldots, U_m, B_1, \ldots, B_n, db_{k_1} : B_{n+1}, \ldots, db_{k_w} : B_{n+w}, \\
& X_1 : B_{n+w+1}, \ldots, X_p : B_{n+w+p} \\
& \textit{is a renamed apart rule of } db_i, \\
& \forall r = 1, \ldots, n\ B'_r \leftarrow \overline{U}_r \in I(db_i), \\
& \forall q = 1, \ldots, w\ B'_{n+q} \leftarrow \overline{U}_{n+q} \in I(db_{k_q}), \\
& \theta = mgu((B_1, \ldots, B_{n+w}), (B'_1, \ldots, B'_{n+w})), \\
& \forall t = 1, \ldots, p\ B'_{n+w+t} \leftarrow \overline{U}_{n+w+t} \in I(X_t\theta), \\
& \theta' = mgu((B_{n+w+1}\theta, \ldots, B_{n+w+p}\theta), (B'_{n+w+1}, \ldots, B'_{n+w+p})), \\
& A = H\theta\theta', \\
& \overline{U} = \textstyle\bigcup_{j=1,m} \{ db_i : U_j\theta\theta' \} \cup \overline{U}_1\theta\theta' \cup \ldots \cup \overline{U}_{n+w+p}\theta\theta' \ \}^3 .
\end{aligned}
$$

□

To build the set of updates, we collect the labeled updates deriving from the resolution of the atoms $B_1, \ldots, B_{n+w+p}$ in the appropriate databases, and the local updates which are labeled by the identifier db_i of the database itself.

The condition requiring that each variable appearing as a label in the body of a rule must appear as argument of an extensional atom of the unlabeled or constant labeled part of the rule, guarantees that the substitution θ is grounding for all such variable labels. Therefore the defined semantics correctly models dynamic channels. Since dynamic labels are no longer visible in the resulting interpretation $T_\Xi(I)$, in that they are instantiated on the extensional database, we do not consider them any more in the semantics of the deductive part of HU-Datalog.

It is easy to prove that the operator defined above is monotonic on the lattice $(\mathcal{I}, \sqsubseteq)$ because T_{db_i} is monotonic, $i = 1, \ldots, s$. Since the domain is finite, the monotonicity of a function is a sufficient condition for its continuity. Therefore T_Ξ is continuous and this allows us to define the fixpoint semantics for a HU-Datalog system as the least upper bound of the chain of the iterated applications of T_Ξ starting from the tuple of the extensional databases.

Definition 9 (Deductive Model)
Let $\Xi = \langle \{ db_1 :: EDB_1 \cup IDB_1 \cup AR_1, \ldots, db_s :: EDB_s \cup IDB_s \cup AR_s \}, AR \rangle$ be a HU-Datalog system. The fixpoint semantics $\mathcal{F}(\Xi)$ of Ξ is defined as $\mathcal{F}(\Xi) = T^\omega_\Xi(\langle EDB_1, \ldots, EDB_s \rangle)$, where as usual, $T^\omega_\Xi(\langle EDB_1, \ldots, EDB_s \rangle)$ represents $\bigsqcup_n T^n_\Xi(\langle EDB_1, \ldots, EDB_s \rangle)$. □

Notice that the above fixpoint is reached in a finite number of steps due to the finiteness of the domain [9].

[3] $\overline{U}\theta$ denotes the set of updates obtained by applying the substitution θ to each update in $\overline{U}$.

Example 10 *Consider the following HU-Datalog system with empty sets of local and global active rules.*

```
m:: p(a). p(b). q(a).                    n:: z(a). z(b).
    r(X) ← p(X), n:s(X,Y), +q(X).        s(X,Y) ← z(X), m:t(Y), +k(X).
    t(X) ← q(X), p(X),-q(X).
```

The least fixpoint computation proceeds as follows:

$$
\begin{aligned}
T^0_\Xi(\langle EDB_{\mathtt{m}}, EDB_{\mathtt{n}}\rangle) &= \langle\{p(a), p(b), q(a)\}, \{z(a), z(b)\}\rangle \\
T^1_\Xi(\langle EDB_{\mathtt{m}}, EDB_{\mathtt{n}}\rangle) &= \langle\{p(a), p(b), q(a),\ t(a) \leftarrow m\!:\!-q(a)\}, \{z(a), z(b)\}\rangle \\
T^2_\Xi(\langle EDB_{\mathtt{m}}, EDB_{\mathtt{n}}\rangle) &= \langle\{p(a), p(b), q(a),\ t(a) \leftarrow m\!:\!-q(a)\}, \{z(a), z(b),\\
&\qquad s(a,a) \leftarrow m\!:\!-q(a), n\!:\!+k(a),\\
&\qquad s(b,a) \leftarrow m\!:\!-q(a), n\!:\!+k(b)\}\rangle \\
T^3_\Xi(\langle EDB_{\mathtt{m}}, EDB_{\mathtt{n}}\rangle) &= \langle\{p(a), p(b), q(a),\ t(a) \leftarrow m\!:\!-q(a),\\
&\qquad r(a) \leftarrow m\!:\!-q(a), n\!:\!+k(a), m\!:\!+q(a),\\
&\qquad r(b) \leftarrow m\!:\!-q(a), n\!:\!+k(b), m\!:\!+q(b)\},\\
&\qquad \{z(a), z(b),\ s(a,a) \leftarrow m\!:\!-q(a), n\!:\!+k(a),\\
&\qquad s(b,a) \leftarrow m\!:\!-q(a), n\!:\!+k(b)\}\rangle \\
T^4_\Xi(\langle EDB_{\mathtt{m}}, EDB_{\mathtt{n}}\rangle) &= T^3_\Xi(\langle EDB_{\mathtt{m}}, EDB_{\mathtt{n}}\rangle)
\end{aligned}
$$

Hence the least fixpoint is $\mathcal{F}(\Xi) = T^3_\Xi(\langle EDB_{\mathtt{m}}, EDB_{\mathtt{n}}\rangle)$. *It is worth remarking that the set of updates in the constrained atom* $r(a) \leftarrow m\!:\!-q(a), n\!:\!+k(a), m\!:\!+q(a)$ *is inconsistent.* ◇

Before introducing the set-oriented semantics we give two auxiliary definitions:

Definition 11 *Given a set of bindings* b *and a transaction* T, *we define*

$$b_{|T} = \{(X = t) \in b \mid X \text{ occurs in } T\}$$

□

Definition 12 *Given a substitution* $\theta = \{V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n\}$ *we define*

$$eqn(\theta) = \{V_1 = t_1, \ldots, V_n = t_n\}.$$

□

Now we define the semantics of a simple transaction T with respect to a HU-Datalog system Ξ. As usual in database systems, we give a default set-oriented semantics, that is, the query-answering process computes a set of answers. We denote with $\mathsf{Set}(T, \Xi)$ the set of pairs $\langle bindings, labeled\ updates\rangle$ computed as answers to the transaction T.

Definition 13 (Simple transaction answer)
Given a HU-Datalog system $\Xi = \langle \{db_1 :: DB_1, \ldots, db_s :: DB_s\}, AR \rangle$ *and a simple transaction* $T{=}B_1, \ldots, B_w, db_{k_1} : B_{w+1}, \ldots, db_{k_p} : B_{w+p}, X_1 : B_{w+p+1}, \ldots, X_q : B_{w+p+q}$, *then*

$$\begin{aligned}\mathsf{Set}(T, \Xi) = \{\langle b, \overline{U}\rangle \mid\ & \forall i = 1, \ldots, w\ \exists\ 1 \leq j \leq s.\ A_i \leftarrow \overline{U}_i \in \mathcal{F}(\Xi)(db_j),\\ & \forall i = 1, \ldots, p\ A_{w+i} \leftarrow \overline{U}_{w+i} \in \mathcal{F}(\Xi)(db_{k_i}),\\ & \theta = mgu((B_1, \ldots, B_{w+p}), (A_1, \ldots, A_{w+p})),\\ & \forall i = 1, \ldots, q\ A_{w+p+i} \leftarrow \overline{U}_{w+p+i} \in \mathcal{F}(\Xi)(X_i\theta),\\ & \theta' = mgu((B_{w+p+1}\theta, \ldots, B_{w+p+q}\theta), (A_{w+p+1}, \ldots, A_{w+p+q})),\\ & b = eqn(\theta\theta')_{|T},\\ & \overline{U} = \textstyle\bigcup_{i=1,w+p+q} \overline{U}_i\theta\theta'\ \}.\end{aligned}$$

□

Note that, if an atom is not labeled, we look for a database in which this atom can be solved, that is, for a database which includes an instance of the atom in its model. We collect all possible solutions in Set. On the other hand, if the atom is labeled, we restrict our search to the database specified by the label. Such a database is always known since the substitution θ is grounding for variable labels, due to the condition that these variables occur at least in one extensional atom of the unlabeled or constant labeled part of the transaction.

Notice that in Obj-U-Datalog, only pairs with consistent updates are inserted in Set. Instead, we release such restriction, deferring the conflict resolution at the next step (Section 4.2). Moreover, atoms labeled by variables are permitted in transactions, whereas Obj-U-Datalog forbids them.

Example 14 *Let* `r(X),n:z(X)` *be a simple transaction on the system* Ξ *described in Example 10. To compute* Set, *the deductive model of* Ξ *is exploited, obtaining:*

$$\begin{aligned}\mathsf{Set}((r(X), n{:}z(X)), \Xi) = \{\ & \langle \{X = a\}, \{m{:}{-}q(a),\ n{:}{+}k(a),\ m{:}{+}q(a)\}\rangle,\\ & \langle \{X = b\}, \{m{:}{-}q(a),\ n{:}{+}k(b),\ m{:}{+}q(b)\}\rangle\ \ \}\end{aligned}$$

Notice that in the first tuple the set of updates is inconsistent. ◇

4.2 Active part semantics

The active part semantics is given following the line of the PARK semantics proposed in [23]. We extend it to deal with labeled atoms and multi-theories. This semantics is well suited to a deferred-update approach, like the one we used in the previous step, and adds much flexibility in that it uses a parametric policy to solve conflicts.

This semantics builds an auxiliary model containing, in particular, the update atoms needed to trigger active rules and to obtain the new system state. To this end, we define a bottom-up operator whose domain is

$$\mathcal{B}^{\pm} = \{db{:}p(\tilde{t}) \mid db \in \Sigma_{db}, p(\tilde{t}) \in \mathcal{B}\} \cup \{db{:}+p(\tilde{t}), db{:}-p(\tilde{t}) \mid db \in \Sigma_{db}, p(\tilde{t}) \in \mathcal{B}, p \in \Pi^e\}$$

where $\mathcal{B}$ is the standard Herbrand Base [4] . A subset of $\mathcal{B}^{\pm}$ is an *i-interpretation* (where the "i" stands for intermediate). An i-interpretation is *consistent* if it does not contain any pair of opposite updates labeled by the same database identifier, i.e. $db{:}+a$ and $db{:}-a$. This is exactly the consistency definition for sets of labeled updates.

In the following, we denote with d, d' a constant or variable database identifier ($d, d' \in Term_{db}$), with db a constant database identifier ($db \in \Sigma_{db}$), with B, B' a $(\Pi^i \cup \Pi^e, \Sigma, V)$-atom, with U a (Π^u, Σ, V)-atom, with G a (Π, Σ, V)-atom, and with L a positive or negative (Π, Σ, V)-atom. We sometimes add subscripts to these symbols.

To establish when an active rule can trigger, that is when its event occurs and its condition holds, we introduce the valid function on labeled atoms and i-interpretations.

Definition 15 (Validity) *The validity of a ground labeled literal* db:a *in an i-interpretation I is defined as follows:*

$$\mathsf{valid}(db{:}a, I) = \begin{cases} I \cap \{db{:}p(\tilde{t}), db{:}+p(\tilde{t})\} \neq \emptyset & \text{if } a = p(\tilde{t}); \\ I \cap \{db{:}p(\tilde{t}), db{:}+p(\tilde{t})\} = \emptyset \text{ or } db{:}-p(\tilde{t}) \in I & \text{if } a = \neg p(\tilde{t}); \\ db{:}a \in I & \text{otherwise.} \end{cases}$$

□

A labeled positive $(\Pi^e \cup \Pi^i, \Sigma, V)$-atom is valid in I if it belongs to I or if it is inserted by an update in I. A labeled negative $(\Pi^e \cup \Pi^i, \Sigma, V)$-atom is valid in I if it is deleted by an update in I, or if its positive atom is not valid. A labeled (Π^u, Σ, V)-atom is valid in I if it belongs to I. Notice that both $db{:}p(\tilde{t})$ and $db{:}\neg p(\tilde{t})$ can be valid according to the above definition. The intuition behind Definition 15 is that since a labeled positive or negative atom belongs to the condition part of the active rule, its validity must be checked with respect to the derived atoms and also to the inserted and deleted atoms. On the other hand, to represent the occurrence of an event, we require that just the labeled update modeling such an event must belong to the i-interpretation.

[4] We recall that the standard Herbrand Base is the set of ground positive atoms consisting of all predicate symbols in $\Pi^i \cup \Pi^e$ and constant symbols in Σ.

To solve the condition part of an active rule, we want to use the knowledge contained in the deductive part. However, in exploiting such knowledge, we work under different assumptions with respect to those presented in Section 4.1:

- conditions should be checked by taking into account the requested updates;
- the resolution of a condition should not affect the state of the system.

While the first condition is assured by Definition 15, to fulfill the second condition we remove the update part from the rules of the intensional databases by using the purification operation defined below.

Definition 16 (Purification) *Let IDB be the intensional database of a HU-Datalog database; we define its* purified *version* $\widehat{IDB}$ *as the set of rules*

$$B_{i+1}, \ldots, B_w, db_{k_1} : B_{w+1}, \ldots, db_{k_p} : B_z, X_1 : B_{z+1}, \ldots, X_q : B_r \rightarrow H.$$

such that there exists in IDB a rule

$$H \leftarrow U_1, \ldots, U_i, B_{i+1}, \ldots, B_w, db_{k_1} : B_{w+1}, \ldots, db_{k_p} : B_z, X_1 : B_{z+1}, \ldots, X_q : B_r.$$

□

Example 17 *The purified form of the intensional databases presented in Example 10 is the following:*

$\widehat{IDB}_{\mathtt{m}} =$ `p(X), n:s(X,Y) → r(x).`
`q(X), p(X) → t(x).`

$\widehat{IDB}_{\mathtt{n}} =$
`z(X), m:t(Y) → s(X,Y).`

◇

It is worth noting that a query is provable in $\widehat{IDB} \cup EDB$ if and only if it is provable in $IDB \cup EDB$, with the same computed answers. Purification only avoids the side effects of the query evaluation. Also notice that we reversed the direction of the arrow in order to have a uniform notation with active rules.

Both purified and local active rules are transformed in order to have a set of rules having only labeled atoms. This operation is called *labeling* and simply adds to each unlabeled atom of a rule the label *db* of the database where it is contained.

Definition 18 (Labeling) *Given a set of rules R and a database identifier* $db \in \Sigma_{db}$*, we define the set of* labeled rules $\mathsf{lab}(R, db)$ *as the set of rules*

$$db : L_1, \ldots, db : L_k, d_1 : L_{k+1}, \ldots, d_r : L_{k+r} \rightarrow db : G_1, \ldots, db : G_m, d'_1 : G_{m+1}, \ldots, d'_r : G_{m+s}.$$

such that there exists in R a rule

$$L_1, \ldots, L_k, d_1 : L_{k+1}, \ldots, d_r : L_{k+r} \rightarrow G_1, \ldots, G_m, d'_1 : G_{m+1}, \ldots, d'_r : G_{m+s}.$$

□

We denote with $\mathcal{L}$ the set of labeled rules. In the sequel, we generically use the term "rules" to refer to both labeled active and labeled purified rules. Notice that Definitions 19, 20, and 21, given a set of rules, only take into account labeled active rules, while subsequent ones work on both kinds of rules.

Now, suppose that given an i-interpretation, several rules are fireable. It may happen that the actions requested by those rules are in conflict. For example, some rules add a certain labeled atom and others remove it from the i-interpretation. In order to obtain a new consistent i-interpretation we prevent one set of rules from firing: if we choose to insert the labeled atom, only the rules adding it are triggered, otherwise only the rules removing it are triggered. Formally, we first define the notion of *conflict*, which consists of a labeled atom and the sets of rules inserting and removing it.

Definition 19 (Conflicts) *A pair (r, θ), where r is a rule and θ is a ground substitution for r is called a* rule grounding.

Let P be a set of rules and let I be an i-interpretation for P. Then conflicts*(P,I) is a set of maximal triples of the form $(db\!:\!a, \mathsf{ins}, \mathsf{del})$ such that a is a ground atom, db is a database identifier and* ins *and* del *are sets of rule groundings. For each such triple the following conditions must hold:*

(1) $\exists\ r = d_1\!:\!L_1, \ldots, d_n\!:\!L_n \rightarrow d_{n+1}\!:\!U_1, \ldots, d_{n+k}\!:\!U_k$, *and*
$r' = d'_1\!:\!L'_1, \ldots, d'_m\!:\!L'_m \rightarrow d'_{m+1}\!:\!U'_1, \ldots, d'_{m+s}\!:\!U'_s$, $\quad r, r' \in P$,
and $\exists\ \theta, \theta'$ *ground substitutions such that*
- $\mathsf{valid}((d_i\!:\!L_i)\theta, I), i = 1, ..., n$,
- $\mathsf{valid}((d'_i\!:\!L'_i)\theta', I), i = 1, ..., m$,
- $\exists\ i, j.\ 1 \leq i \leq k.\ 1 \leq j \leq s.\ U_i = +B,\ U'_j = -B'$ *and*
$(db\!:\!a) = (d_{n+i}\!:\!B)\theta = (d'_{m+j}\!:\!B')\theta'$.

(2) For all possible r, r' and θ, θ', satisfying condition 1 above, $(r, \theta) \in$ ins *and $(r', \theta') \in$* del. □

A triple $(db\!:\!a, \mathsf{ins}, \mathsf{del}) \in \mathsf{conflicts}(P, I)$ is called a *conflict*. To solve conflicts, a parametric conflict resolution policy is introduced. In the following we call labeled extensional database a set of labeled ground extensional atoms.

Definition 20 (Conflict resolution policy) *Let Dom be a domain, recording information about rules (i.e., the priority of the rule, the database the rule belongs to, and anything else which is useful to implement a certain policy). Given a labeled extensional database EDB, a set of rules P, a mapping $f : \mathcal{L} \rightarrow Dom$, an i-interpretation I and a conflict c, we define* $\mathsf{sel}(EDB, P, f, I, c)$ *as a total function with codomain* $\{\mathsf{insert}, \mathsf{delete}\}$. □

The intended meaning of $\mathsf{sel}(EDB, P, f, I, (db\!:\!a, \mathsf{ins}, \mathsf{del}))$ is to choose whether the labeled atom $db\!:\!a$, object of the conflict, should be inserted in or deleted from I, thus effectively choosing which of the conflicting update requests

should prevail. The function f provides information about the rules, which may be useful to implement a certain policy.

Gottlob et al. [23] present a number of commonly adopted policies, and discuss their advantages and disadvantages. We briefly recall here some of them. The *principle of inertia* states that *both* the conflicting updates should be discarded, thus leaving EDB in the same state as before with respect to $db:a$ (in our framework, this can be obtained by returning insert if $db:a$ was already in EDB, delete otherwise). The *source priority* policy determines which update should prevail according to which database the rules requesting such updates come from (in our framework, this can be obtained by using the mapping f which establishes the relation between rules and databases of our system). The *rule priority* policy, found in systems such as Ariel [27], Postgres [42] and Starburst [47], assumes that each rule has a (static or dynamic) priority associated with it; sel returns insert or delete as needed to preserve the update requested by the highest-priority rule. Other policies, like voting schemes or user queries, are also reasonable, but the final choice is left to the particular application.

Based on the result of the sel policy, we prevent the rule instances in one of the two sets of a conflict from firing, by blocking them according to the following definition.

Definition 21 (Blocked rule instances) *Given a labeled extensional database EDB, a set of rules P, a mapping $f : \mathcal{L} \to Dom$, a conflict resolution policy* sel*, and an i-interpretation I, let*

$$X = \{\mathsf{del} \mid (db{:}a, \mathsf{ins}, \mathsf{del}) \in \mathsf{conflicts}(P, I), \\ \mathsf{sel}(EDB, P, f, I, (db{:}a, \mathsf{ins}, \mathsf{del})) = \mathsf{insert}\}$$
$$Y = \{\mathsf{ins} \mid (db{:}a, \mathsf{ins}, \mathsf{del}) \in \mathsf{conflicts}(P, I), \\ \mathsf{sel}(EDB, P, f, I, (db{:}a, \mathsf{ins}, \mathsf{del})) = \mathsf{delete}\}$$

We define $\mathsf{blocked}(EDB, P, f, I, \mathsf{sel}) = (\bigcup_{x \in X} x) \cup \left(\bigcup_{y \in Y} y\right)$. □

An entire rule instance is blocked, instead of the single update responsible for the conflict. This choice has the nice side effect that the set of updates requested by a rule instance exhibits an atomic behavior: either all the updates in the set are executed, or no update at all. Such atomic behavior prevents the risk of an inconsistent database state because of partially-executed actions.

Example 22 *Consider the labeled extensional database EDB and the set of rules P given below:*

$EDB =$ `m:t(a,a).` `m:t(a,b).` `n:r(a).`

$P =$
r_1 `m:t(a,X)` $\rightarrow$ `m:p(X).`
r_2 `n:s(X), n:r(X)` $\rightarrow$ `m:p(X).`
r_3 `n:+s(X),m:t(X,Y)` $\rightarrow$ `o:+q(X).`
r_4 `n:+s(X),m:t(X,Y)` $\rightarrow$ `o:−q(Y).`
r_5 `n:+s(X)` $\rightarrow$ `o:+q(b).`
r_6 `o:+q(X),m:p(X)` $\rightarrow$ `m:−t(X,X).`

Given an i-interpretation $I = \{m : t(a, a), m : t(a, b), n : r(a), n : +s(a)\}$*, the conflict set is*

$$\mathsf{conflicts}(P, I) = \{c_1,\ c_2\}$$

where

$c_1 = (o{:}q(a),\ \{(r_3, \{X \leftarrow a, Y \leftarrow a\}), (r_3, \{X \leftarrow a, Y \leftarrow b\})\},$
$\{(r_4, \{X \leftarrow a, Y \leftarrow a\})\})$
$c_2 = (o{:}q(b), \{(r_5, \{X \leftarrow a\})\}, \{(r_4, \{X \leftarrow a, Y \leftarrow b\})\}).$

Now, consider a function f *associating each rule with the database in which the rule is contained. Suppose that the conflict resolution policy* sel *is such that*

$$\mathsf{sel}(EDB, P, f, I, c_1) = \mathsf{insert} \qquad \mathsf{sel}(EDB, P, f, I, c_2) = \mathsf{delete}$$

Then, we have that

$$\mathsf{blocked}(EDB, P, f, I, \mathsf{sel}) = \{(r_4, \{X \leftarrow a, Y \leftarrow a\}), (r_5, \{X \leftarrow a\})\}. \qquad \diamond$$

Using the above concepts, the immediate consequence operator on i-interpretations can be defined as follows:

Definition 23 (Immediate consequence operator) *Given a set of rules* P*, a set of blocked rule instances* B*, and an i-interpretation* I*, we define* $\Gamma_{P,B}(I)$ *as the smallest set* S *satisfying the following conditions:*

- $I \subseteq S$*;*
- *If* $r = d_1{:}L_1, \ldots, d_n{:}L_n \rightarrow d'_1{:}G_1, \ldots, d'_m{:}G_m \in P$ *and* θ *is a ground substitution for* r *such that*
  - $(r, \theta) \notin B$
  - $\mathsf{valid}((d_i{:}L_i)\theta, I), i = 1, ..., n$

  then $\{(d'_1{:}G_1)\theta, \ldots, (d'_m{:}G_m)\theta\} \subseteq S$. $\square$

The operator $\Gamma_{P,B}$ is monotonic on the lattice $(\wp(\mathcal{B}^{\pm}), \subseteq)$, hence continuous because $\wp(\mathcal{B}^{\pm})$ is finite.

The main difference of the above operator with respect to the traditional immediate consequence operator of logic programming is that it may happen that some rules are not fired even if their body is valid.

Example 24 *Let I be the i-interpretation, P the set of rules, EDB the labeled extensional database, and $B = \mathsf{blocked}(EDB, P, f, I, \mathsf{sel})$ presented in Example 22, then we obtain*

$$\Gamma_{P,B}(I) = \{\ m{:}t(a,a),\ m{:}t(a,b),\ n{:}r(a),\ n{:}{+}s(a),\ m{:}p(a),\ m{:}p(b),\ o{:}{+}q(a),\ o{:}{-}q(b)\ \ \}$$

The use of blocked *has prevented the insertion of any conflict in $\Gamma_{P,B}(I)$.* $\diamond$

In general, the application of the function $\Gamma_{P,B}$ to a consistent i-interpretation does not return a consistent i-interpretation. We must appropriately select rules, that is we must build a set of blocked rules B such that the least fixpoint of $\Gamma_{P,B}$ is consistent. Thus, instead of dealing with i-interpretations, the notion of bi-structures is introduced, as in [23], in order to take into account blocked rules.

Definition 25 (Bi-structures) *A bi-structure $\langle B, I\rangle$ consists of a set B of rule groundings and of an i-interpretation I. We define an order relation on bi-structures as follows:*

$$\langle B, I\rangle \prec \langle B', I'\rangle \quad \Leftrightarrow \quad \begin{cases} B \subset B' & \text{or} \\ B = B' \text{ and } I \subset I' \end{cases}$$

Moreover, given $\mathcal{A}$ and $\mathcal{B}$ bi-structures, $\mathcal{A} \preceq \mathcal{B} \quad \equiv \quad (\mathcal{A} = \mathcal{B} \vee \mathcal{A} \prec \mathcal{B})$. $\square$

Bi-structures ordered by $\preceq$ form a complete partial order because there are only finitely many pairs $\langle B, I\rangle$. On this domain, we can define an operator having a fixpoint, that is used as a model of the active part.

Definition 26 (Δ operator) *Given a set of rules P, a mapping $f : \mathcal{L} \to Dom$, a bi-structure $\langle B, I\rangle$, and a conflict resolution policy* sel*, we define*

$$\Delta_{P,f,\mathsf{sel}}(\langle B, I\rangle) = \begin{cases} \langle B, \Gamma_{P,B}(I)\rangle & \text{if } \Gamma_{P,B}(I) \text{ is consistent} \\ \langle B \cup \mathsf{blocked}(I^{\perp}, P, f, I, \mathsf{sel}), I^{\perp}\rangle & \text{otherwise} \end{cases}$$

where $I^{\perp} = \{db{:}p(\tilde{t}) \in I \mid p \in \Pi^e\}$, i.e. the set of labeled extensional atoms contained in I. $\square$

The definition of Δ we give here differs from the original in [23] in two respects:

- the i-interpretation consists of labeled ground atoms, because it represents the derived atoms and the updates concerning a set of databases, since we are in a multi-theory context;
- the set of rules P contains not only rules with updates in the right hand side (properly active rules) but also purified rules that derive intensional

knowledge rather than new updates. Notice that this extension does not affect the consistency of i-interpretations, since the purified rules can only add labeled (Π^i, Σ, V)-atoms to the i-interpretation.

The intuitive idea of the Δ operator is that, if no conflict arises, Δ does not change the blocked rules set B, and only the i-interpretation of the bi-structure is changed by adding the immediate consequences of the non-blocked rules. On the other hand, as soon as a conflict arises, the conflict is solved via the resolution policy sel and all the blocked rule instances are collected. Then the computation of Δ starts again from the i-interpretation $I^\perp$ and the augmented set of blocked rules. The i-interpretation $I^\perp$ represents the set of labeled extensional atoms of the system, and we have to resort to it to be sure that the starting point of the new computation does not contain atoms whose validity depends on actions of rule instances that are now blocked. As remarked in [23] for the PARK semantics, this semantics may be viewed as a smooth cycle integrating inflationary fixpoint computation [30] and conflict resolution policies.

The next proposition states that the Δ operator is *growing* (statement (1)), therefore, for the finiteness of the domain, a fixpoint is reached by iterating Δ a finite number of steps (statement (2)). Furthermore, the i-interpretation of the fixpoint is consistent (statement (3)). The proof is an easy reformulation of the ones presented in [4] and [23].

Proposition 27 *Given a set of rules P, a mapping $f : \mathcal{L} \to Dom$, a conflict resolution policy* sel*, a bi-structure $\mathcal{A} = \langle B, I \rangle$ with I a set of ground labeled extensional atoms, the following statements hold:*

(1) $\mathcal{A} \preceq \Delta_{P,f,\mathsf{sel}}(\mathcal{A})$*;*
(2) there exists k such that $\Delta^k_{P,f,\mathsf{sel}}(\mathcal{A})$ is a fixpoint of $\Delta_{P,f,\mathsf{sel}}$;
(3) if $\Delta^k_{P,f,\mathsf{sel}}(\mathcal{A}) = \langle B', I' \rangle$, then $I' = \mathsf{lfp}_I(\Gamma_{P,B'})$ [5] *and I' is consistent.* □

4.3 Integrating deductive and active semantics

To assign a semantics to a HU-Datalog system, we must compose the results of the deductive part and active part semantics. We first compute the set of answers in the marking phase (Definition 13). Then we build a set of labeled active and purified rules (Definition 28), and define a mapping $f : \mathcal{L} \to Dom$ and a conflict resolution policy sel in order to apply the Δ operator (Definition 26). From the fixpoint of Δ, we obtain a consistent i-interpretation (Proposition 27), from which the new state of the system is computed (Definition 29).

[5] $\mathsf{lfp}_I(f)$ denotes the least fixpoint of f which is greater or equal to I.

Before presenting the bottom-up semantics of a HU-Datalog system, we need to define the set of rules used to compute the active part semantics, how to incorporate updates in the state and finally, which are the observable properties of a transaction.

The set of rules and the labeled extensional database, used to compute the active part semantics, are defined as follows.

Definition 28 (ρ-set, ε-set)
Let $\Xi = \langle \{db_1 :: EDB_1 \cup IDB_1 \cup AR_1, \ldots, db_s :: EDB_s \cup IDB_s \cup AR_s\}, AR\rangle$ *be a HU-Datalog system and* $\overline{U}$ *be a set of labeled updates. We define*

$$\rho(\Xi, \overline{U}) = \left(\bigcup_{i=1,s} \{r \mid r \in \mathsf{lab}(\widehat{IDB}_i \cup AR_i, db_i)\} \right) \cup AR \cup (\{\rightarrow db\!:\!+a \mid db\!:\!+a \in \overline{U}\} \cup \{\rightarrow db\!:\!-a \mid db\!:\!-a \in \overline{U}\})$$

$$\varepsilon(\Xi) = \bigcup_{i=1,s} \{db_i\!:\!p(\tilde{t}) \mid p(\tilde{t}) \in EDB_i\}.$$

□

The set $\rho(\Xi, \overline{U})$ contains: the purified rules of the intensional databases and the local active rules, appropriately labeled; the global active rules; and the updates requested from the deductive part represented as rules with neither event nor condition. Therefore this set keeps the intensional knowledge of the system needed to verify the condition part of the active rules, and both local and global active rules needed to maintain local consistency and to propagate updates. On the other hand, $\varepsilon(\Xi)$ contains the extensional knowledge of the system, i.e., the set of all labeled atoms belonging to the state of the system.

The semantics of the previous steps (see Sections 4.1 and 4.2) includes neither the execution of the collected updates nor considers the transactional behavior. We now define a function which given an i-interpretation and the current state of the system, returns the next state obtained by executing the updates in the i-interpretation.

Definition 29 (Updates incorporation) *Given a consistent i-interpretation* I *and a tuple* $EDB = \langle EDB_1, \ldots, EDB_s\rangle$*, where* EDB_i *is the extensional database of* db_i*, we define*

$$\mathsf{incorp}(I, EDB) = \langle EDB'_1, \ldots, EDB'_s\rangle$$

where $EDB'_i = (EDB_i \setminus \{a \mid db_i\!:\!-a \in I\}) \cup \{a \mid db_i\!:\!+a \in I\}$. □

Finally, the observable properties of a transaction we are interested in are the

set of answers, the next system state, and the result of the transaction itself (i.e. Commit or Abort).

Definition 30 (Observables)
An observable *Obs is a triple* $\langle \mathsf{Ans}, EDB, \mathsf{Res}\rangle$ *where* Ans *is a set of bindings, EDB is a tuple of extensional databases and* $\mathsf{Res} \in \{\mathsf{Commit}, \mathsf{Abort}\}$. *The set of all observables is denoted by* Obs. □

Now, we give the semantics of a HU-Datalog system distinguishing two cases, namely, the case of simple transaction and the case of complex transaction.

Definition 31 (HU-Datalog system semantics)
Let $\Xi = \langle \{db_1 :: EDB_1 \cup IDB_1 \cup AR_1, \ldots, db_s :: EDB_s \cup IDB_s \cup AR_s\}, AR\rangle$ *be a HU-Datalog system, T be a transaction, f be a mapping from the set of rules* $\mathcal{L}$ *to a certain domain Dom, and* sel *be a conflict resolution policy. The semantics of a transaction T is denoted by the function* $Sem_{\Xi,f,\mathsf{sel}}$ *defined as*

$$Sem_{\Xi,f,\mathsf{sel}}(T) = \mathcal{S}_{\Xi,f,\mathsf{sel}}(T)(\langle \emptyset, \langle EDB_1, \ldots, EDB_s\rangle, \mathsf{Commit}\rangle)$$

where the function $\mathcal{S}_{\Xi,f,\mathsf{sel}}(T) : \mathsf{Obs} \to \mathsf{Obs}$ *is defined as follows:*

If T is a simple transaction, then

$$\mathcal{S}_{\Xi,f,\mathsf{sel}}(T)(\langle \alpha, \xi, \gamma\rangle) = \begin{cases} \langle \emptyset, \xi, \mathsf{Abort}\rangle & \text{if } \gamma = \mathsf{Abort} \\ \langle \mathsf{Ans}, \mathsf{incorp}(I, \xi), \mathsf{Commit}\rangle & \text{if } \overline{U} \text{ ground, } \gamma = \mathsf{Commit} \\ \langle \emptyset, \xi, \mathsf{Commit}\rangle & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \mathsf{Ans} &= \{b_j \mid \langle b_j, \overline{u}_j\rangle \in \mathsf{Set}(T, \Xi)\} \\ \overline{U} &= \bigcup_j \{\overline{u}_j \mid \langle b_j, \overline{u}_j\rangle \in \mathsf{Set}(T, \Xi)\} \\ P &= \rho(\Xi, \overline{U}) \\ \langle B, I\rangle &= \Delta^{\omega}_{P,f,\mathsf{sel}}(\langle \emptyset, \varepsilon(\Xi)\rangle). \end{aligned}$$

If T is a complex transaction $T_1; \ldots; T_k$ $(k \geq 2)$, *then*

$$\mathcal{S}_{\Xi,f,\mathsf{sel}}(T_1; \ldots; T_k)(Obs) = \mathcal{S}_{\Xi',f,\mathsf{sel}}(T_2; \ldots; T_k)(\mathcal{S}_{\Xi,f,\mathsf{sel}}(T_1)(Obs))$$

where $\Xi' = \langle \{db_1 :: EDB_1 \cup IDB'_1 \cup AR_1 \ldots, db_s :: EDB_s \cup IDB'_s \cup AR_s\}, AR\rangle$ *and* $\langle EDB'_1, \ldots, EDB'_s\rangle$ *is the new state of the system, that is, the second component of the observable* $\mathcal{S}_{\Xi,f,\mathsf{sel}}(T_1)(Obs)$. □

It is worth remarking that our semantics, unlike the semantics developed in [5], never generates aborts because of conflicts, thus augmenting the successful computations. This is made possible by the active component of our language, and especially by the conflict resolution policy. However, as already discussed, we elect to keep the indication of success (commit/abort) as observable property. Therefore the first case of the above definition, propagating Abort, is formally necessary, although it will be never used in computing the result of a transaction.

Moreover, note that the "otherwise" case discards all changes to the state of the system if a non-ground update is requested. The semantics of a complex transaction is simply given by the sequential composition of the updates generated by its component transactions, since none of them can abort. The state of the system is updated after each simple transaction. Besides, this semantics discards the answer to all but the last transaction, to stay close to the approach in [5].

As already noted, the answer set Set and the Δ fixpoint are computed in a finite number of steps, hence $Sem_{\Xi,f,\mathsf{sel}}$ is computed in a finite number of steps. Actually, it is computable in polynomial time in the size of the state of the system when the mapping f and the conflict resolution policy sel are computable in polynomial time.

Example 32 *Consider again the two databases* `school` *and* `lib` *introduced in Example 7. We want to integrate the school–library system with the database of another school, called* `sch2`*. Being in a heterogeneous environment,* `sch2`*'s structure is quite different from that of* `school`*. In particular, the second school admits two types of students, namely undergraduate (*`undergr` *predicate) and PhD (*`phd` *predicate), and some of its courses are split into two units, with an examination at the end of each unit. A two-units course is considered passed when both the first and the second unit examinations are passed; units must be taken in their natural order.*

```
sch2:: undergr(pat).                phd(annie).
       exam(math,2).                exam(cs,1).
       units_passed(pat,math,2).    units_passed(annie,math,1).
       units_passed(annie,cs,1).

       student(S) ← undergr(S).
       student(S) ← phd(S).
       passed(S,E) ← student(S),exam(E,N),units_passed(S,E,N).
       pass_unit(S,E,1) ← student(S),exam(E,N),+units_passed(S,E,1).
       pass_unit(S,E,2) ← student(S),exam(E,2),units_passed(S,E,1),
                          +units_passed(S,E,2),−units_passed(S,E,1).
```

In the database `sch2`*, predicate* `exam` *is used to maintain the name of a course*

and the number of its units. Predicate `units_passed` *stores, instead, the number of units, of a given course, a student has already passed. Information about courses passed by a student can be obtained from predicate* `passed`*, whereas predicate* `pass_unit` *updates information about the number of units, of a given course, a student has already passed. For the sake of the example, we do not consider active rules in this database. Notice how* `sch2` *is fully local (no atom is labeled) and totally unaware of the existence of* `school` *and* `lib`*.*

We recall that to integrate the `school` *and* `lib` *databases, we introduced the following rules in our global active rules set:*

```
school:+student(S) → lib:+user(S).
school:−student(S) → lib:−user(S).
school:+passed(S,E),lib:loan(S,B),lib:sect(E,B) → lib:+request(B,S).
```

These rules express our integration policy*: every student of* `school` *is a user of* `lib`*, and when a student passes an exam, the library requests back any book the student has on loan for the course he/she has just passed. Our integration policy regarding* `school` *and* `sch2` *is easily stated: we want that students can move from* `school` *to* `sch2`*, and in doing so any exam they have passed in the first school is preserved (if applicable) in the new school. The following global active rules state exactly the intended behavior:*

```
school:+move(S,sch2) → school:−student(S), sch2:+undergr(S).
school:+move(S,sch2),school:passed(S,E),sch2:exam(E,N)→
                                        sch2:+units_passed(S,E,N).
```

These rules are more specialized than the similar ones presented in Example 7 since we cannot simply assume that the other school database has the same structure of the first one. This is a common phenomenon: it is possible to write rules for generic databases if it is possible to assume something about their structure, otherwise specialized rules must be used to perform integration.

We decide also that undergraduate students of `sch2` *should have access to the library. Such behavior obtained by using the following global active rules:*

```
sch2:+undergr(S) → lib:+user(S).
sch2:−undergr(S) → lib:−user(S).
```

`sch2` *is more liberal than* `school` *regarding loans, so it will not claim back books on loan when a student from* `sch2` *passes an exam.*

We still need to introduce the `move` *predicate in* `school` *to allow the active rules to fire. This can be obtained by adding*

```
transfer(S,T) ← student(S), +move(S,T).
```

to the IDB of `school`*.*

This addition is appropriately placed in `school`*, because a transfer to another school is an event semantically relevant for a student of* `school`*, and only the* `school` *database can verify whether the student to transfer is actually a student in* `school`*. Although the verification is specialized to a given database, as it should be, the actual transfer is not: both* `transfer` *and* `move` *handle the destination school as a totally generic database. Only in the global active rules the name of the destination is used to perform the appropriate integration.*

Now, we can show how the system $\Xi = \langle \{\texttt{school}, \texttt{lib}, \texttt{sch2}\}, AR \rangle$ *reacts to the transaction*

$$T = \texttt{school:transfer(john,sch2)}$$

The first step computes the fixpoint semantics of the deductive part, according to Definition 9. The result is the following:

$$\begin{aligned}
\mathcal{F}(\Xi) = \langle\ & \{\ student(\mathcal{S})^{6},\ exam(\mathcal{E}),\ passed(john,engl),\ passed(john,math),\\
& passed(mary,phys),\ passed(frank,engl),\\
& pass(\mathcal{S},\mathcal{E}) \leftarrow school{:}+passed(\mathcal{S},\mathcal{E}),\ leave(\mathcal{S}) \leftarrow school{:}-student(\mathcal{S}),\\
& transfer(\mathcal{S},\mathcal{D}) \leftarrow school{:}+move(\mathcal{S},\mathcal{D})\ \},\\
& \{\ user(\mathcal{S}),\ user(pat),\ book(\mathcal{K}),\ sect(engl,hamlet),\\
& sect(phys,principia),\ loan(hamlet,john),\ loan(principia,frank),\\
& deny_loan(hamlet,\mathcal{S}),\ deny_loan(hamlet,pat),\\
& deny_loan(principia,\mathcal{S}),\ deny_loan(principia,pat),\\
& return(hamlet,john) \leftarrow lib{:}-loan(hamlet,john),\\
& return(principia,frank) \leftarrow lib{:}-loan(principia,frank)\ \},\\
& \{\ undergr(pat),\ phd(annie),\ exam(math,2),\ exam(cs,1),\\
& units_passed(pat,math,2),\ units_passed(annie,math,1),\\
& units_passed(annie,cs,1),\ student(\mathcal{T}),\ passed(pat,math),\\
& passed(annie,cs),\ pass_unit(\mathcal{T},\mathcal{G},1) \leftarrow sch2{:}+units_passed(\mathcal{T},\mathcal{G},1),\\
& pass_unit(annie,math,2) \leftarrow sch2{:}+units_passed(annie,math,2),\\
& \qquad\qquad sch2{:}-units_passed(annie,math,1)\ \}\quad \rangle
\end{aligned}$$

The transaction answer, obtained according to Definition 13, is

$$\mathsf{Set}(T, \Xi) = \{\langle \emptyset, \{school{:}+move(john,sch2)\}\rangle\}.$$

Next, the active part semantics must be computed. The purified intensional databases are

[6] As a shorthand, in the following we write $\mathcal{S}$ to stand for all elements in {john, mary, frank}, $\mathcal{E}$ for all elements in {engl, math, phys}, $\mathcal{G}$ for all elements in {math, cs}, $\mathcal{K}$ for all elements in {hamlet, principia}, $\mathcal{T}$ for all elements in {pat, annie}, and $\mathcal{D}$ for all elements in {school, lib, sch2}.

$\widehat{IDB}_{\texttt{school}}$= `student(S), exam(E)→pass(S,E).`
`student(S)→leave(S).`
`student(S)→transfer(S,T).`

$\widehat{IDB}_{\texttt{lib}}$= `request(X,U),book(B)→deny_loan(B,U).`
`loan(B,Y),user(U)→deny_loan(B,U).`
`loan(B,U)→return(B,U).`

$\widehat{IDB}_{\texttt{sch2}}$= `undergr(S)→student(S).`
`phd(S)→student(S).`
`student(S),exam(E,N),units_passed(S,E,N)→passed(S,E).`
`student(S),exam(E,N)→pass_unit(S,E,1).`
`student(S),exam(E,2),units_passed(S,E,1)→pass_unit(S,E,2).`

Therefore, the ρ-set for Ξ and the updates $\overline{U} = \{school{:}{+}move(john, sch2)\}$, obtained from $\mathsf{Set}(T, \Xi)$*, is*

$$
\begin{aligned}
\rho(\Xi, \overline{U}) = \{\ & school{:}student(S), school{:}exam(E) \rightarrow school{:}pass(S,E),\\
& school{:}student(S) \rightarrow school{:}leave(S),\\
& school{:}student(S) \rightarrow school{:}transfer(S,T),\\
& school{:}{-}student(S), school{:}passed(S,E) \rightarrow school{:}{-}passed(S,E),\\
& lib{:}request(X,U), lib{:}book(B) \rightarrow lib{:}deny\_loan(B,U),\\
& lib{:}loan(B,Y), lib{:}user(U) \rightarrow lib{:}deny\_loan(B,U),\\
& lib{:}loan(B,U) \rightarrow lib{:}return(B,U),\\
& lib{:}{-}user(U), lib{:}loan(B,U) \rightarrow lib{:}{+}request(B,U),\\
& lib{:}{-}loan(B,U), lib{:}request(B,U) \rightarrow lib{:}{-}request(B,U),\\
& sch2{:}undergr(S) \rightarrow sch2{:}student(S),\\
& sch2{:}phd(S) \rightarrow sch2{:}student(S),\\
& sch2{:}student(S), sch2{:}exam(E,N), sch2{:}units\_passed(S,E,N) \rightarrow\\
& \qquad\qquad sch2{:}passed(S,E),\\
& sch2{:}student(S), sch2{:}exam(E,N) \rightarrow sch2{:}pass\_unit(S,E,1),\\
& sch2{:}student(S), sch2{:}exam(E,2), sch2{:}units\_passed(S,E,1) \rightarrow\\
& \qquad\qquad sch2{:}pass\_unit(S,E,2),\\
& school{:}{+}student(S) \rightarrow lib{:}{+}user(S),\\
& school{:}{-}student(S) \rightarrow lib{:}{-}user(S),\\
& school{:}{+}passed(S,E), lib{:}loan(S,B), lib{:}sect(E,B) \rightarrow\\
& \qquad\qquad lib{:}{+}request(B,S),\\
& school{:}{+}move(S, sch2) \rightarrow school{:}{-}student(S), sch2{:}{+}undergr(S),\\
& school{:}{+}move(S, sch2), school{:}passed(S,E), sch2 :: exam(E,N) \rightarrow\\
& \qquad\qquad sch2{:}{+}units\_passed(S,E,N),\\
& sch2{:}{+}undergr(S) \rightarrow lib{:}{+}user(S),\\
& sch2{:}{-}undergr(S) \rightarrow lib{:}{-}user(S),\\
& \rightarrow school{:}{+}move(john, sch2) \quad \}
\end{aligned}
$$

The corresponding ε-set is

$$
\begin{aligned}
\varepsilon(\Xi) = \{\ & school{:}student(john),\ school{:}student(mary),\ school{:}student(frank),\\
& school{:}exam(engl),\ school{:}exam(math),\ school{:}exam(phys),
\end{aligned}
$$

$school{:}passed(john, engl)$, $school{:}passed(john, math)$,
$school{:}passed(mary, phys)$, $school{:}passed(frank, engl)$,
$lib{:}user(john)$, $lib{:}user(mary)$, $lib{:}user(frank)$, $lib{:}user(pat)$,
$lib{:}book(hamlet)$, $lib{:}book(principia)$, $lib{:}sect(engl, hamlet)$,
$lib{:}loan(hamlet, john)$, $lib{:}loan(principia, frank)$,
$sch2{:}undergr(pat)$, $sch2{:}phd(annie)$, $sch2{:}exam(math, 2)$,
$sch2{:}exam(cs, 1)$, $sch2{:}units\_passed(pat, math, 2)$,
$sch2{:}units\_passed(annie, math, 1)$, $sch2{:}units\_passed(annie, cs, 1)$
}

We assume an inertial conflict resolution policy (see Section 4.2) that does not use the mapping $f : \mathcal{L} \to Dom$. Therefore we leave f undefined. The computation of the system semantics through the Δ operator proceeds as follows:

$$\Delta^0_{\rho(\Xi,\overline{U}),f,\mathsf{sel}}(\langle\emptyset, \varepsilon(\Xi)\rangle) = \langle\emptyset, I_1 = \varepsilon(\Xi)\rangle$$

$$\begin{aligned}\Delta^1_{\rho(\Xi,\overline{U}),f,\mathsf{sel}}(\langle\emptyset, \varepsilon(\Xi)\rangle) = \langle\emptyset, I_2 = I_1 \cup \{\ & school{:}pass(\mathcal{S}, \mathcal{E}),\\ & school{:}leave(\mathcal{S}),\ school{:}transfer(\mathcal{S}, \mathcal{D}),\\ & lib{:}deny\_loan(hamlet, \mathcal{S}),\\ & lib{:}deny\_loan(hamlet, pat),\\ & lib{:}deny\_loan(principia, \mathcal{S}),\\ & lib{:}return(principia, frank), sch2{:}student(\mathcal{T}),\\ & school{:}{+}move(john, sch2)\ \}\rangle\end{aligned}$$

$$\begin{aligned}\Delta^2_{\rho(\Xi,\overline{U}),f,\mathsf{sel}}(\langle\emptyset, \varepsilon(\Xi)\rangle) = \langle\emptyset, I_3 = I_2 \cup \{\ & sch2{:}pass\_unit(\mathcal{T}, \mathcal{G}, 1),\\ & sch2{:}pass\_unit(annie, math, 2),\\ & school{:}{-}student(john),\ sch2{:}{+}undergr(john),\\ & sch2{:}{+}units\_passed(john, math, 2)\ \}\rangle\end{aligned}$$

Now, a conflict arises. Indeed:

$$\begin{aligned}I' = \Gamma_{\rho(\Xi,\overline{U}),\emptyset}(I_3) = I_3 \cup \{\ & school{:}{-}passed(john, engl),\\ & school{:}{-}passed(john, math),\ lib{:}{-}user(john),\\ & lib{:}{+}user(john)\ \}.\end{aligned}$$

The labeled atom object of the conflict is lib:user(john). *The rule that tries to insert it is* $r_1 =$`sch2:+undergr(S)`$\rightarrow$ `lib:+user(S)`, *with grounding substitution* $\{S{\leftarrow}john\}$; *the rule that tries to remove it is* $r_2 =$`school:-student(S)`$\rightarrow$ `lib:-user(S)`, *with the same grounding substitution. Both rules belong to the AR set. Since we are using the inertial conflict resolution policy,* $\mathsf{sel}(\varepsilon(\Xi), \rho(\Xi, \overline{U}), f, I', (lib{:}user(john), \{(r_1, \{S \leftarrow john\})\}, \{(r_2, \{S \leftarrow john\})\}) = \mathsf{insert}$, *thus blocking* $(r_2, \{S{\leftarrow}john\})$. *The computation proceeds without further conflicts, restarting from the bi-structure* $\langle B = \{(r_2, \{S{\leftarrow}john\})\}, \varepsilon(\Xi)\rangle$.

$$\Delta^{3}_{\rho(\Xi,\overline{U}),f,\mathsf{sel}}(\langle\emptyset,\varepsilon(\Xi)\rangle) = \langle B, I_1\rangle$$

$$\Delta^{4}_{\rho(\Xi,\overline{U}),f,\mathsf{sel}}(\langle\emptyset,\varepsilon(\Xi)\rangle) = \langle B, I_2\rangle$$

$$\Delta^{5}_{\rho(\Xi,\overline{U}),f,\mathsf{sel}}(\langle\emptyset,\varepsilon(\Xi)\rangle) = \langle B, I_3\rangle$$

$$\Delta^{6}_{\rho(\Xi,\overline{U}),f,\mathsf{sel}}(\langle\emptyset,\varepsilon(\Xi)\rangle) = \langle B, I_4 = I_3 \cup \{\ school\!:\!-passed(john, engl),\ school\!:\!-passed(john, math),\ lib\!:\!+user(john)\ \}\rangle$$

$$\Delta^{7}_{\rho(\Xi,\overline{U}),f,\mathsf{sel}}(\langle\emptyset,\varepsilon(\Xi)\rangle) = \Delta^{6}_{\rho(\Xi,\overline{U}),f,\mathsf{sel}}(\langle\emptyset,\varepsilon(\Xi)\rangle)$$

Hence the fixpoint is reached.

After performing update incorporation (Definition 29), the new state EDB' of the system is

$EDB'_{\texttt{school}} =$
```
student(mary).   student(frank).
exam(engl).      exam(math).      exam(phys).
passed(mary,math). passed(frank,engl).
move(john,sch2).
```

$EDB'_{\texttt{lib}} =$
```
user(mary).      user(john).
user(frank).     user(pat).
book(hamlet).    book(principia).
sect(engl,hamlet). sect(phys,principia).
loan(john,hamlet). loan(principia,frank).
```

$EDB'_{\texttt{sch2}} =$
```
undergr(pat).    undergr(john).    phd(annie).
exam(math,2).    exam(cs,1).
units_passed(pat,math,2).   units_passed(annie,math,1).
units_passed(annie,cs,1).   units_passed(john,math,2).
```

and the final observable is $Obs' = \langle\emptyset, EDB', \mathsf{Commit}\rangle$.

By looking at the new state of the system, we can see that now John is an undergraduate student of school sch2 *and he is still a user of the library. However, he lost his English exam because such a course is not taught in the school* sch2. ◇

5 Conclusions and future work

In this paper, we defined a logical language supporting cooperative queries, updates, and update propagation. We model the sources of information as deductive databases, sharing the same logical language to express queries and up-

dates, but containing independent, even if possibly related, data. The language used to model data sources has been obtained by extending the Obj-U-Datalog language [5] to deal with active rules in the style of Active-U-Datalog [4,22], whose semantics has been defined according to the PARK semantics proposed in [23]. The use of Obj-U-Datalog enables cooperative query answering and distributed transaction execution among different data sources, whereas active rules perform update propagation and consistency maintenance. The proposed framework results in a uniform integration of active and deductive rules to model cooperation.

This work can be extended in several ways. A first important question is related to the definition and analysis of properties concerning execution of distributed queries, transactions and active rules. A second direction concerns various extensions to the proposed language, such as the introduction of negation in the deductive part of a HU-Datalog system and constructs for modeling more complex events. Finally, since a HU-Datalog database, when ignoring updates and active rules, represents a particular amalgamated knowledge base, as defined in [44], another interesting topic is the extension of the general amalgamated knowledge base framework to deal with updates and actions.

As concluding remark we would like to comment on the relations between our work and recent work on agent technology. Even though it is difficult to precisely define what an agent is, we can consider an agent as "a self-contained program capable of controlling its own decision-making and acting, based on its perception of its environment, in pursuit of one or more objectives" [28]. More specifically, an agent should be characterized by a number of key properties including: *social ability* – referring to the ability to interact with other agents and/or humans; *autonomy* – referring to the control an agent should have over its own actions and internal state; *reactivity* – referring to the ability of an agent to react to changes that occur in the agent state and in the environment. The object model we have developed in this work satisfies all these key properties. In particular, in our model, an object is characterized by a state and a mechanism to perform queries and modifications to the object state. Moreover, objects can communicate with other objects, also according to unanticipated modalities, through the use of labeled atoms. Since labels can also be variables, both static and dynamic communication channels among objects are supported. Finally, objects are able to react to events through the use of active rules. We plan to investigate the use of our object model as a foundation for agent technology dealing with intelligent information management.

References

[1] S. Abiteboul and V. Vianu. A Transaction Language Complete for Database Updates and Specification. In *Proc. of the Seventh ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 260–268, 1987.

[2] S. Abiteboul and V. Vianu. Procedural and Declarative Database Update Languages. In *Proc. of the Eighth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 240–250, 1988.

[3] ANSI TC X3H2 and ISO/IEC JTC 1/SC 21/WG 3. Master Index for SQL and all its Parts, March 1966. Document X3H2-96-066 DBL:MCI-011.

[4] E. Bertino, B. Catania, V. Gervasi, and A. Raffaetà. Active-U-Datalog: Integrating Active Rules in a Logical Update Language. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors, *LNCS 1472: Transactions and Change in Logic Databases*, pages 106–132, 1998.

[5] E. Bertino, G. Guerrini, and D. Montesi. Toward Deductive Object Databases. *Theory and Practice of Object Systems*, 1(1):19–39, 1995.

[6] E. Bertino, M. Martelli, and D. Montesi. Transactions and Updates in Deductive Databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(5):784-797, 1997.

[7] A.J. Bonner and M. Kifer. An Overview of Transaction Logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[8] O.A. Bukhres and A. Elmagarmid, editors. *Object-Oriented Multidatabase Systems*. Prentice-Hall, 1996.

[9] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, Berlin, 1990.

[10] S. Ceri and J. Widom. Deriving Incremental Production Rules for Deductive Data. *Information Systems*, 19(6):467–490, 1994.

[11] S. Ceri and J. Widom. Managing Semantic Heterogeneity with Production Rules and Persistent Queues. In R. Agrawal, S. Baker, and D.A. Bell, editors, *Proc. of the Int. Conf. on Very Large Data Bases*, pages 108–119, 1993.

[12] S.S. Chawathe, H. Garcia-Molina, and J. Widom. A Toolkit for Constraint Management in Heterogeneous Information Systems. In S.Y.W. Su, editor, *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 56-65, 1996.

[13] W. Chen. Declarative Specification and Evaluation of Database Updates. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *LNCS 566: Proc. of the Int. Conf. on Deductive and Object Oriented Databases*, pages 147–166, 1991.

[14] W. Chen. Programming with Logical Queries, Bulk Updates, and Hypothetical Reasoning. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):587–599, 1997.

[15] U. Dayal, A.P. Buchmann, and S. Chakravarthy. The HiPAC Project. In S. Ceri and J. Widom, editors, *Active Database Systems*, pages 177-206. Morgan Kauffman, 1996.

[16] L.M.L. Delcambre and J.N. Etheredge. The Relational Procedural Language: A Production Language for Relational Databases. In L. Kerschberg, editor, *Proc. of the Second Int. Conf. on Expert Database Systems*, pages 333-351, 1988.

[17] Digital Equipment Corporation. *Rdb/VMS - SQL Reference Manual*, November 1991.

[18] L. Do and P. Drew. Active Database Management of Global Data Integrity Constraints in Heterogeneous Database Environments. In P.S. Yu and A.L.P. Chen, editors, *Proc. of the Int. Conf. on Data Engineering*, pages 99-108, 1995.

[19] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1993.

[20] A.A.A. Fernandes, M.H. Williams, and N.W. Paton. A Logic-Based Integration of Active and Deductive Databases. *New Generation Computing*, 15(2):205-244, 1997.

[21] N. Gehani and H.V. Jagadish. Ode as an Active Database: Constraints and Triggers. In G.M. Lohman, A.Sernadas, and R. Camps, editors, *Proc. of the Seventeenth Int. Conf. on Very Large Data Bases*, pages 327-336, 1991.

[22] V. Gervasi and A. Raffaetà. Active-U-Datalog: Integrating Active Rules in a Deductive Database. Technical Report 97-19, Dipartimento di Informatica, Pisa, Italy, 1997.

[23] G. Gottlob, G. Moerkotte, and V.S. Subrahmanian. The PARK Semantics for Active Rules. In P.M.G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *LNCS 1057: Proc. of the Fifth Int. Conf. on Extending Database Technology*, pages 35–55, 1996.

[24] J. Gray and A. Reuter. *Transaction Processing Concepts and Techniques*. Morgan-Kaufmann, 1993.

[25] G. Guerrini and D. Montesi. Design and Implementation of Chimera Active Rules. *Int. Journal on Data and Knowledge Engineering*, 24(1):39-67, 1997.

[26] A. Gupta, M. P. Reddy, and M. Siegel. Towards an Active Schema Integration Architecture for Heterogeneous Database Systems. In H.J. Schek, A.P. Sheth, and B.D. Czejdo, editors, *Proc. of the IEEE Int. Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS93)*, pages 178–183, 1993.

[27] E. Hanson. The Design and Implementation of the Ariel Active Database Rule System. *IEEE Transactions on Knowledge and Data Engineering*, 8(1): 157-172, 1996.

[28] N. R. Jennings and M. Woldridge. Software Agents. *IEEE Review*, pages 17–20, January 1996.

[29] Kirkwood. *Sybase Architecture and Administration.* Prentice-Hall, 1993.

[30] P. Kolaitis and C. Papadimitriou. Why Not Negation by Fixpoint? *Journal of Computer and System Sciences*, 43(1):125–144, 1991.

[31] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[32] B. Ludäscher, U. Hamann, and G. Lausen. A Logical Framework for Active Rules. In *Proc. of the Seventh Int. Conf. on Management of Data (COMAD)*, 1995.

[33] B. Ludäscher, W. May, and G. Lausen. Nested Transactions in a Logical Languages for Active Rules. In D. Pedreschi and C. Zaniolo, editors, *LNCS 1154: Proc. of the Int. Workshop on Logic in Databases*, pages 197–222, 1996.

[34] S. Manchanda. Declarative Expression of Deductive Database Updates. In *Proc. of the SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 93–100, 1989.

[35] S. Manchanda and D.S. Warren. A Logic-Based Language for Database Updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 363–394. Morgan-Kaufmann, 1988.

[36] D. R. McCarhy and U. Dayal. The Architecture of an Active Database Management System. In J. Cliffors, B.G. Lindsay, and D. Maier, editors, *Proc. of the Int. Conf. on Extending Data Base Technology*, pages 215-224, 1989.

[37] D. Montesi and R. Torlone. A Transaction Transformation Approach to Active Rule Processing. In P.S. Yu and A.L.P. Chen, editors, *Proc. of the Eleventh Int. Conf. on Data Engineering*, pages 109–116, 1995.

[38] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, 1989.

[39] Oracle Corp. *Oracle 7 Server Concepts Manual, 7.3.* Oracle Corp., February 1996.

[40] H.J. Schek, A. Sheth, and B. Czejdo, editors, *Proc. of the IEEE Int. Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS93)*, 1993.

[41] M. Stonebraker. *The Ingres Papers.* Addison-Wesley, Reading MA, 1986.

[42] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views in Data Base Systems. In H.G. Molina and H.V. Jagadish, editors, *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 281–290, 1990.

[43] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–92, October 1991.

[44] V. S. Subrahmanian Amalgamating Knowledge Bases. *ACM Transactions on Database Systems*, 19(2):291–331, 1994.

[45] C.A. Wichert and B. Freitag. Capturing Database Dynamics by Deferred Updates. In *Proc. of the Int. Conf. on Logic Programming*, pages 226–240, 1997.

[46] J. Widom, R. J. Cochrane, and B. G. Lindsay. Implementing Set-oriented Production Rules as an Extension to Starburst. In G.M Lohman, A. Sernadas, and R. Camps, editors, *Proc. of the Seventeenth Int. Conf. on Very Large Data Bases*, pages 275-285, 1991.

[47] J. Widom and S. Finkelstein. Set-oriented Production Rules in Relational Databases. In H.G. Molina and H.V. Jadadish, editors, *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 259–270, 1990.

[48] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25:38–49, 1992.

[49] C. Zaniolo. A Unified Semantics for Active and Deductive Databases. In N. W. Paton and M. H. Williams, editors, *Proc. of the First Int. Workshop on Rules in Database Systems*, pages 271–287, 1993.