# Transactions and Updates in Deductive Databases

Danilo Montesi Elisa Bertino Maurizio Martelli

Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, UK
D.Montesi@doc.ic.ac.uk

Dipartimento di Scienze dell'Informazione
Università di Milano
Via Comelico 39, 20135 Milano, Italy
bertino@hermes.mc.dsi.unimi.it

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Via Benedetto XV, 3 16132 Genova, Italy
martelli@disi.unige.it

**Technical Report DOC 95/2**

**Abstract**

In this paper we develop a new approach providing a smooth integration of extensional updates and declarative query language for deductive databases. The approach is based on a declarative specification of updates in rule bodies. Updates are not executed as soon are evaluated. Instead, they are collected and then applied to the database when the query evaluation is completed. We call this approach non-immediate update semantics. We provide a top-down and equivalent bottom-up semantics which reflect the corresponding computation models. We also package set of updates into transactions and we provide a formal semantics for transactions. Then, in order to handle complex transactions, we extend the transaction language with control constructors still preserving formal semantics and semantics equivalence.

# 1 Introduction

In recent years, deductive databases have been the focus of intense research, which has brought strong advances in theory, systems and applications. The major advantages of deductive databases are their formal setting based on logic paradigm [31] and their computational models for query evaluation, such as top-down or bottom-up evaluation [5]. Generally speaking, a database language should express both structural and behavioral properties [16]. Structure deals with static properties while behavior concerns dynamic properties. Structural properties have been investigated in relational and deductive databases with particular emphasis on the expressiveness and efficiency of query languages.

Several proposals for the inclusion of update capabilities in declarative languages have been presented. In general all these proposals are based on including in rules, in addition to usual predicates, special atoms denoting updates[1]. However, all these proposals are based on the immediate update semantics. Under that semantics, an update is executed as soon as it is encountered during the evaluation of a rule, containing the update atom. Thus the declarativeness of the language is lost due to the introduction of a sequential order among updates inside the rules of the programs. Indeed, because of the updates performed during a goal evaluation, the answers returned by the goal may differ depending on the rule evaluation order choosen by the query processor. This is a first drawback of such update approach.

Existing proposals can be classified depending on whether updates appear in the head or the body of a rule. The presence of updates in rule body leads to top-down computations, whereas the presence of updates in rule heads leads to bottom-up computations. That is, bottom-up and top-down reasonings are no more equivalent. This is a second drawback. Indeed, it is desirable to answer queries by computing forward from the base relations, thus obtaining a set-oriented behavior. However, bottom-up evaluation does not take advantage of constants, that may occur in queries, for restricting the search space. Such a restriction is a by-product of top-down reasoning. Thus computational models, where top-down and bottom-up are combined, provide the appropriate approach to efficient evaluation strategies [5].

Another major drawback is that updates are often forced to be serially executed rather than in parallel. However, many operations in real applications are inherently parallel. This is the case of a bank transfer where insertion and deletion can be performed in parallel instead than is sequence. Similarly, two independent deposit transactions can be performed in parallel. Finally, immediate updates semantics leads to rule computations which go through sequences of states. This implies a complex rules semantics [1]. Despite these drawbacks, immediate updates semantics is the default in many current deductive database systems where updates are performed in sequence.

In order to avoid the above drawbacks, we consider a different approach to the integration of updates and a declarative query language. Our approach can be sum-

[1] We will elaborate more on this later on.

marized as follows. The intensional database consists of a set of rules, which can have update atoms in their bodies. Whenever, a rule with updates, is evaluated as part of a goal evaluation, the non-immediate semantics is used for these updates. Thus, these updates are applied to the extensional database only when the goal evaluation is completed. In addition to the rule language, used to write the rules, we provide a transaction language. Such language is used for writing the deductive database applications. In its simplest form, a transaction is a goal. Goals can be combined, by using sequence and iteration constructors, to form complex transactions. Transactions, either simple or complex, are atomic execution units, according to standard transaction models. Within a complex transaction, an immediate update semantics is used among goals. Thus, the *update visibility model* of our transaction language is as follows:

- each goal does not "see" the effects of its own updates, since these are installed only after the goal evaluation is completed;
- a goal in a transaction sees the updates performed by previously executed goals within the same complex transaction.

Note that this model is fully in accord with database languages, like SQL, and current transaction models. For example, in a transaction written in SQL, if an update statement is peformed before a query statement, the query will see the effects of the update statement. Indeed, a transaction always "sees" its own updates.

Our approach avoids the drawbacks of other deductive database languages with updates, and allows real applications to be modeled. In particular, the rule language models the dynamic behavior of the database system, by preserving at the same time the equivalence of top-down and bottom-up computations. Thus top-down and bottom-up evaluation are reconciled.

Updates, in our approach, are computed according to two phases. During the first phase updates are collected and their consistency is checked[2]. In the second phase, the collected updates are executed altogether. Therefore, rule computations do not go through sequences of states, performing updates in sequence, but updates are executed in parallel, that is, they do not form a sequence but a set. Such execution models a simple transaction where all-or-none of the updates are executed. In order to model complex transactions, i.e. transactions with control constructors, the transaction language is extended with explicit constructors external to rule language. In this way, whenever required, updates can form a sequence. Thus, the computation of a complex transaction goes through a sequence of states, without affecting the nice feature of our approach for the rule language.

Once the language (with a four steps) semantics is provided, we have a two interesting equivalence notions: equivalence between transactions, and equivalence between databases. In our approach a transaction may contain queries, updates and control constructors. Thus we have a unified framework to model static and dynamic features while keeping the nice feature of the declarative query language.

[2] A set of updates is said to be consistent if does not contain complementary updates.

This unified framework allows to investigate equivalence properties, not only between sequences of updates (see for instance [4]), but also to take into account queries and control constructors, thus improving the generality of equivalence towards real transactions.

The formal setting proposed here has several interesting extensions. First of all, equivalence among transactions is used to optimize transaction executions by means of transaction and database transformations. Transaction optimization through static analysis has been developed in [7, 8]. A second extension is towards a more general data model such as object oriented in the deductive context [9, 11]. Finally, other relevant developments include semantic integrity constraints [42]. Even if important, we do not consider in this paper algorithmic, complexity and implementation issues; we focus here on the language and its semantics and compare with existing approaches. Algorithms to execute and optimize transactions are described in the above papers. The implementation issues are described in [6].

In order to provide a formal foundation for our language and semantics, we model the language in terms of constraint logic programming (CLP) [36], one of the most relevant research directions in the logic programming area. CLP integrates two of the most interesting programming paradigms, namely constraint and logic programming. Updates in rule bodies are specified through constraints [12]. Therefore, our language inherits the formal setting of CLP, in a similar way as Datalog inherits the formal setting of logic programming.

The remainder of this paper is organized as follows. Section 2 introduces the basic notions of deductive databases and reviews related works. Section 3 informally introduces our approach to updates and queries integration. Section 4 presents the design of the U-Datalog language to express updates and (complex) transactions. Section 5 presents the four steps semantics for the above language and the two equivalence relations: the first for transactions; the second for databases. In Section 6 the results are summarized and some directions for further research are discussed. Finally, Appendix A introduces the CLP language and the relationship with U-Datalog. The required extensions to CLP, and the proofs are given in Appendix B.

# 2 Related works

In this section we introduce the basic notions of deductive databases that are relevant to the subsequent discussion. Then we critically review some approaches to model updates in deductive databases.

## 2.1 Deductive databases

The motivations for using rule languages in the database systems are twofold. On one side they are easy to learn and understand. They are also more human oriented and higher level than other formalisms specifically developed for computers. On the other side they extend the relational domain calculus into a rule language. Rule

languages have promised to maintain the nice properties of relational languages, such as simplicity and theoretical foundation, while adding inferential features. Rule languages can be based on logical paradigm such as Datalog-like language [22, 18] or on production one such as OPS5 [17]. Researches on rule languages started with logic languages and have resulted in a comprehensive theoretical framework to study declarative query languages. Production languages for databases still do not have a theoretical framework even if some languages and systems have been developed, for instance, RDL1 [20], Heraclitus [35] and Starburst [26]. In the following we introduce databases based on logic languages and we provide examples of both the families of rule languages.

A deductive database (DB) can be seen as a set of sentences defining data (facts) or views (deductive rules). Facts are stored in an *extensional database* ($EDB$) and the set of deductive laws is called the *intensional database* ($IDB$). The Datalog query language is a good representative of logic query languages. Often we refer to a database simply as $DB$ that is $EDB \cup IDB$. Both extensional and intensional databases are built starting from atoms. The *atoms* are expressions of the form $p(t_1, ..., t_n)$, where the $t_i, 1 \leq i \leq n$, are terms and $p$ is a predicate name. Atoms which do not contain variables are called *grounds*. A *term* is either a constant or a variable. Predicate symbols are partitioned in two disjoint sets: the extensional predicates and the intensional predicates. The former appear in the extensional database and the latter occur in the intensional database but not in the extensional one. The extensional atoms are those built starting from the extensional predicates.

An *intensional* database (IDB) is a finite set of rules. A *rule* is an expression of the form $H \leftarrow B_1, \ldots, B_n, n \geq 0$, where the *body* $B_1, \ldots, B_n$ is a conjunction of extensional and intensional atoms and the head $H$ is an intensional atom. In the following, $\tilde{t}$ denotes the tuple $t_1, ..., t_n$, while $\tilde{A}$ denotes a (possibly empty) conjunction of atoms $A_1, \ldots, A_k$. A *substitution* is a function $\theta : V \rightarrow Term$ associating with each variable a term in $Term$. It extends to apply to any syntactic object (e.g. term, atom, rules, etc.) in the usual way. We require rules to be *safe*. A rule is safe if each variable in the head appears in an atom in the body [48]. A *query* is a rule with no head.

EXAMPLE 2.1. *Consider the Datalog program $EDB \cup IDB$, where* `father`, `mother` *and* `grandfather` *express the relations father, mother and grandfather respectively.*

$$
\begin{aligned}
EDB = \quad & \texttt{father}(\texttt{tom}, \texttt{sue}). \\
& \texttt{mother}(\texttt{sue}, \texttt{bob}). \\
& \texttt{father}(\texttt{kim}, \texttt{alan}).
\end{aligned}
$$

$$
\begin{aligned}
IDB = \quad & \texttt{grandfather}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{father}(\texttt{X}, \texttt{Z}), \texttt{father}(\texttt{Z}, \texttt{Y}). \\
& \texttt{grandfather}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{father}(\texttt{X}, \texttt{Z}), \texttt{mother}(\texttt{Z}, \texttt{Y}).
\end{aligned}
$$

*The intensional database derives the relation* `grandfather(tom, bob)`. *Usually the $EDB$ is considered as a time varying collection of information, while the $IDB$ is considered as a time invariant set of rules.*

$\diamond$

## 2.2 Updates in rule languages

Databases should change over time to follow the evolution of stored information. Unfortunately, Datalog language does not support updates. Dynamic aspects such as elementary updates to the extensional database are fundamental in database systems. Research on database updates can be classified into *inference-based* and *declaration-based* approaches. In the former, a database is viewed as a logic theory, and an update is intended to make an arbitrary sentence true in a new database. The focus in the inference-based approach is to derive a new database that satisfies certain criteria; the exact modification to the current database are not explicit, and have thus to be inferred [25, 49]. In a declaration-based approach, a database is viewed as a finite model or a finite set of ground facts. Changes to the current database, such as insertions and deletions, are explicitly declared by users. Recent works in this area have been focusing on introducing updates into logic programming or studying the expressive power of update languages. In the rest of this section we compare some of these works from the point of view of extending logic languages with updates.

Updates can be expressed in a declarative query language outside the query language or within. For instance, Glue-Nail [21] separates the query language from the update language. However, the most interesting languages and systems are those that integrate the query and update languages, for example, Dynamic logic programming (DLP) [41], $\mathcal{LDL}$ [43], transaction logic programming ($\mathcal{T}_R$) [14], Declarative Languages (DL) [3], and RDL1 [20].

Languages, providing integrated query and update facilities, can in turn be classified depending on whether updates are defined in the rule bodies (DLP, $\mathcal{LDL}$ and $\mathcal{T}_R$) or in the rule heads (DL and RDL1). These two approaches were developed with different goals. The first family was developed to model updates in logic programming, while the second was developed to investigate the expressive power of rule languages. In the following we consider these two families of approaches by mean of examples. It is important to note that the two ways of introducing updates induce two different ways to introduce control in rule languages. We start with DLP as a representative of the first family. Updates in DLP rule bodies are viewed as modal operators. The formal semantics proposed for DLP is based on Kripke's semantics for dynamic logic [34].

EXAMPLE 2.2. *Consider the following DLP program where* `es` *and* `ed` *are base relation while* `hire` *and* `avgsal` *are derived ones.*

$$\begin{aligned} \texttt{hire}(\texttt{ename}, \texttt{sal}, \texttt{dept}) \leftarrow\ & +\texttt{es}(\texttt{ename}, \texttt{sal}), \\ & +\texttt{ed}(\texttt{ename}, \texttt{dept}), \\ & \texttt{avgsal}(\texttt{dept}, \texttt{avg}), \\ & \texttt{avg} \leq \texttt{50k}) \end{aligned}$$

*when called by the query ?* $\texttt{hire}(\texttt{joe}, \texttt{60k}, \texttt{weapons})$ *the above program will add the base relation* $\texttt{es}(\texttt{joe}, \texttt{60k})$ *and then* $\texttt{ed}(\texttt{joe}, \texttt{weapons})$. *The average salary of*

*the* `weapons` *department is computed and the updated database is returned if this average is less than* `50k`*. If this average is greater than* `50K`*, the procedure will fail and no updates will be performed.*

◇

In the above example the (update) atoms in rule body are not evaluated together but are evaluated in sequence.

EXAMPLE 2.3. *Consider the following DLP query.*

$$? \ \texttt{es(joe, 20k)}, +\texttt{ed(joe, weapons)}, \texttt{edp(joe, weapons, proj)}.$$

*The query checks whether* `es(joe, 20k)` *is true in the current database, then adds the base relation* `ed(joe, weapons)`*, and then finds the value of* `proj` *for which* `edp(joe, weapons, proj)` *is true in the updated database.*

◇

The atoms `es(joe, 20k)` and `edp(joe, weapons, proj)` above are not evaluated in parallel. Indeed, between the two atoms there is an update which changes the database state. Thus, rule computation goes through a sequence of database states. This is strictly related to the intended interpretation of DLP rules reflecting the computational model which is only top-down.

Similarly to DLP, $\mathcal{T}_R$ allow updates in rule bodies. In addition, transaction logic provides a clear way to combine sequences of updates through a special operator called serial conjunction and to specify nondeterministic transactions. Moreover, the *all-or-nothing* behavior of a transaction is central to this approach and has a mathematical foundation lacking in other proposals. The semantics is the model and proof theory. Thus the only computational model available is, as in DLP, top-down.

By contrast, Chen in [19] adopts a different approach considering update specifications to describe a set of updates. This approach avoids introducing control in the rule language and thus it is closer to declarative languages. However, it does not allow updates execution but only update specifications, that is they do not permanently change the database. Lindholm and O'Keefe in [39] instead, were looking for efficient implementation of updates in Prolog. They argued that although the update problem in logic languages is not logical, its behavior needs not be incoherent with the logic paradigm. Their proposal relies on a logical view which avoids the immediate updates semantics due to the fact that in Prolog one might need to backtrack. Because of this, the immediate updates view is not considered appropriate for high performance Prolog systems. Thus Lindholm and O'Keefe approach to efficient Prolog system considers updates in parallel and not in sequence, but the formal framework is lacking.

As we have said, there is another way to introduce updates in a logic language. It is related to definition of updates in rule heads. This is the case of DL where negative literals in heads are interpreted as deletions and positive literals are interpreted as insertions. Note that, this interpretation is not universally recognized [38]. A

deterministic behavior is obtained in DL by firing in parallel all applicable rules. The nondeterministic behavior results from firing (nondeterministically) one rule at a time. The formal semantics proposed for DL extensions is based on fixpoint extensions of first-order logic [33].

EXAMPLE 2.4. *Consider the DL program where* s *is a base relation, and* r, t *are derived ones.*

$$+\mathtt{r(X)} \leftarrow \mathtt{s(X)}, \neg \mathtt{t(X)}$$
$$+\mathtt{t(X)} \leftarrow \mathtt{s(X)}, \neg \mathtt{r(X)}.$$

*Suppose we apply this program to the extensional database* $\mathtt{s(1)}, \mathtt{s(2)}, \mathtt{s(3)}$. *Under the nondeterministic computation, we can derive several fixpoints, for instance* $\mathtt{s(1)}, \mathtt{s(2)}, \mathtt{s(3)}, \mathtt{r(1)}, \mathtt{t(2)}, \mathtt{t(3)}$, *and* $\mathtt{s(1)}, \mathtt{s(2)}, \mathtt{s(3)}, \mathtt{t(3)}, \mathtt{r(1)}, \mathtt{r(2)}$. *Each of the fixpoints is a model for the program. Other models can be obtained as well. Under the deterministic computation the above program applied to* $\mathtt{s(1)}, \mathtt{s(2)}, \mathtt{s(3)}$ *yields the unique result* $\mathtt{s(1)}, \mathtt{s(2)}, \mathtt{s(3)}, \mathtt{r(1)}, \mathtt{r(2)}, \mathtt{r(3)}, \mathtt{t(1)}, \mathtt{t(2)}, \mathtt{t(3)}$ *in a single step.* ◇

Note that the updates computing the first fixpoint of the Example 2.4, that is +r(1) and then +t(2),+t(3), are performed in sequence due to the nondeterministic computation. Thus this approach introduces control under the nondeterministic semantics. By contrast under the deterministic semantics, the updates are performed in parallel. Unfortunately, both the approaches allow only a bottom-up computational model.

Summarizing, the introduction of updates in rule heads or bodies induces a form of control which does not allow the use of a top-down and equivalent bottom-up computational models. This is a strong drawback because many efficient query evaluation techniques rely on combined computational models, that take advantage of the constants in a query by means of top-down computations and provide set oriented answers by means of bottom-up computations [5].

Moreover, the first type of approach does support run-time parameter passing, while the second one does not. In addition, the molteplicity of DL fixpoints is not a desiderable feature for databases, since each computed model can be seen as a database state. Finally, we note that even though the introduction of control, through updates or other constructors, improves the expressive power of languages, it results in abandoning the nice declarative setting with the well-known drawbacks.

# 3 Informal overview

As we have discussed, existing approaches to the introduction of updates in declarative query languages induces a form of control. We believe that this is related to the classic semantics of updates, namely the immediate updates semantics. This is the case of DLP, $\mathcal{LDL}$, DL, RDL1 and $\mathcal{T}_R$. Other approaches, instead, consider the specification of updates without execution.

Our approach could be considered as complementary to those based on immediate updates. We define a logic language with updates as much as possible close to a declarative language avoiding the introduction of control within the rule language. To avoid induced control due to updates, we consider a non-immediate update semantics. Under this semantics updates are not executed as soon are evaluated. They are collected during a phase called the "marking phase". During this phase the consistency of updates is checked. If they are consistent, e.g. there are no insertion/deletion of the same fact, then they are executed altogether in parallel during the "update phase" which follow the marking phase.

There are several motivations for supporting updates based on a non-immediate semantics. The first is that such semantics is close to the semantics underlying updates in languages of relational DBMS, namely the SQL language. Indeed, in SQL it is possible to perform updates on a relation by identifying the tuples to be updated via a query. For example, one of the formats for the update command in SQL[3] is the following:

```
UPDATE relation-name SET set-rule
WHERE qualification-rule
```

where set-rule is a set of attribute assignments. An assignment has, in turn, the form attribute-name = expression. The qualification rule is a Boolean combination of predicates selecting the tuples to be updated. An update command is executed by first selecting the tuples to be updated, and then performing the update on these tuples. Note that the evaluation of the qualification rule is performed on the database state before applying any update. Therefore, the evaluation of the qualification rule in an update command is independent on any update performed as part of the same command. In our approach, we follow the same pattern in that: (i) the marking phase can be seen as the equivalent of the evaluation of the qualification rule in an SQL update command; (ii) the update phase can be seen as the equivalent of the execution of attribute assignments in an SQL update command.

A second important motivation is that a non-immediate update semantics ensures that the set of performed updates is independent on the evaluation order of updates. Therefore, we regain the declarative setting. Static analysis of rules is thus possible in order to detect rules that may perform inconsistent updates [8]. To our knowledge, no static analysis techniques nor tools have been developed with respect to updates for other logic languages with updates. This seems related to the fact that immediate semantics, used in other approaches, makes very complex such static analysis. We believe, however, that for large rule databases static analysis is important for improving user productivity and supporting explanation tools.

Another important choice in the design of our language is the position of updates in rule language. The main motivations for considering updates in rule bodies is:

1. The approach with updates in the heads does not allow the transmission of runtime parameters.

[3]Note that the DELETE command of SQL also allows to identify the tuples to be removed via a qualification rule.

2. We are interested in inheriting query processing strategies developed for Datalog programs.

3. Updates with non-immediate semantics in rule bodies result in a language which nicely fits into the formal setting of constraint logic programming.

We extend Datalog with a set of atoms that can occur in the body of a Datalog rule and thus also of a query by allowing updates atoms to specify updates to base relations. By contrast, the Datalog atoms are called query atoms. The intuitive meaning of $+p(t_1, \ldots, t_n)$ is to insert the tuple $t_1, \ldots, t_n$ into the base relation $p$; correspondingly, the intuitive meaning of $-p(t_1, \ldots, t_n)$ is to delete the tuple $t_1, \ldots, t_n$ from the base relation $p$.

EXAMPLE 3.1. *Assume that* `balance` *is a base relation giving the account number and the balance. To modify this relation we have two operations:* $-$`balance(Acnt, Amt)` *to delete tuples from the relation and* $+$`balance(Acnt, Amt)` *to insert tuples into the relation. Then we define four transactions:*

- `changebalance(Acnt, Bal, Bal'`) *to change the balance of an account;*
- `withdraw(Atm, Acnt)` *to withdraw an amount from an account;*
- `deposit(Amt, Acnt)` *to deposit an amount into an account;*
- `transfer(Amt, Acnt, Acnt'`) *to transfer an amount from an account to another.*

*These transactions are defined by the following four rules:*

```
changebalance(Acnt, B, B') ← −balance(Acnt, B),
                              +balance(Acnt, B').
withdraw(Amt, Acnt) ← changebalance(Acnt, B, B − Amt),
                      balance(Acnt, B).
deposit(Amt, Acnt) ← changebalance(Acnt, B, B + Amt),
                     balance(Acnt, B).
transfer(Amt, Acnt, Acnt') ← withdraw(Amt, Acnt),
                             deposit(Amt, Acnt').
```

*A change operation can be specified by means of deletion and insertion. The marking phase marks all the tuples to be inserted and deleted. The update phase collects the updates and then inserts and deletes such tuples from the relation* `balance`. ◇

Deletion and insertion do not form a sequence, they are collected as a set of updates and can be executed in parallel. Similarly the two deposit transactions ?`deposit(Amt, Acnt)`, `deposit(Amt', Acnt'`) are performed in parallel. An important difference of our approach with respect to DLP, $\mathcal{LDL}$ and $\mathcal{T}_R$ is that updates are *collected* as side effects of the query-answering process, while in the other approaches they are *executed* as side effects of the query-answering process. To highlight this difference compare the execution of Example 3.1 with that of Example 2.4 in [14].

## 3.1 Computational Model

Non-immediate update semantics is based on marking and update phases. The marking phase models the query-answering process. Query-answering process can be performed either top-down or bottom- up. In the case of updates in the bodies, the top-down interpretation of a rule is intuitive and thus in the remainder of the discussion we consider the bottom-up interpretation.

EXAMPLE 3.2. *Consider the following database.*

$$
\begin{aligned}
DB = \quad & \mathtt{balance}(\mathtt{102}, \mathtt{1500}). \\
& \mathtt{balance}(\mathtt{105}, \mathtt{28000}). \\
& \mathtt{potentialsaver}(\mathtt{Acnt}, \mathtt{B}) \leftarrow \mathtt{balance}(\mathtt{Acnt}, \mathtt{B}), \\
& \qquad\qquad\qquad\qquad\qquad \mathtt{B} > \mathtt{10000}. \\
& \mathtt{changebalance}(\mathtt{Acnt}, \mathtt{B}, \mathtt{B}') \leftarrow -\mathtt{balance}(\mathtt{Acnt}, \mathtt{B}), \\
& \qquad\qquad\qquad\qquad\qquad +\mathtt{balance}(\mathtt{Acnt}, \mathtt{B}'). \\
& \mathtt{withdraw}(\mathtt{Amt}, \mathtt{Acnt}) \leftarrow \mathtt{changebalance}(\mathtt{Acnt}, \mathtt{B}, \mathtt{B} - \mathtt{Amt}), \\
& \qquad\qquad\qquad\qquad\qquad \mathtt{balance}(\mathtt{Acnt}, \mathtt{B}), \mathtt{Atm} < \mathtt{B}. \\
& \mathtt{deposit}(\mathtt{Amt}, \mathtt{Acnt}) \leftarrow \mathtt{changebalance}(\mathtt{Acnt}, \mathtt{B}, \mathtt{B} + \mathtt{Amt}), \\
& \qquad\qquad\qquad\qquad\qquad \mathtt{balance}(\mathtt{Acnt}, \mathtt{B}). \\
& \mathtt{transfer}(\mathtt{Amt}, \mathtt{Acnt}, \mathtt{Acnt}') \leftarrow \mathtt{withdraw}(\mathtt{Amt}, \mathtt{Acnt}), \\
& \qquad\qquad\qquad\qquad\qquad \mathtt{deposit}(\mathtt{Amt}, \mathtt{Acnt}').
\end{aligned}
$$

*Informally, a bottom-up interpretation of the database computes*

$$
\begin{aligned}
\mathcal{S}(DB) = \{\ & balance(102, 1500), \\
& balance(105, 28000), \\
& potentialsaver(105, 28000), \\
& changebalance(Acnt, B, B') \leftarrow -balance(Acnt, B), \\
& \qquad\qquad +balance(Acnt, B'), \\
& withdraw(Amt, 102) \leftarrow -balance(102, 1500), \\
& \qquad\qquad +balance(102, 1500 - Amt), \\
& \qquad\qquad Amt < 1500, \\
& withdraw(Amt, 105) \leftarrow -balance(105, 28000), \\
& \qquad\qquad +balance(105, 28000 - Amt), \\
& \qquad\qquad Amt < 28000, \\
& deposit(Amt, 102) \leftarrow -balance(102, 1500), \\
& \qquad\qquad +balance(102, 1500 + Amt), \\
& deposit(Amt, 105) \leftarrow -balance(105, 28000), \\
& \qquad\qquad +balance(105, 28000 + Amt), \\
& transfer(Amt, 102, 105) \leftarrow -balance(102, 1500), \\
& \qquad\qquad +balance(102, 1500 - Amt), \\
& \qquad\qquad -balance(105, 28000), \\
& \qquad\qquad +balance(105, 28000 + Amt), \\
& \qquad\qquad Amt < 1500, \\
& transfer(Amt, 105, 102) \leftarrow -balance(105, 28000), \\
& \qquad\qquad +balance(105, 28000 - Amt), \\
& \qquad\qquad -balance(102, 1500), \\
& \qquad\qquad +balance(102, 1500 + Amt), \\
& \qquad\qquad Amt < 28000 \qquad\qquad \}
\end{aligned}
$$

*Because of updates, $\mathcal{S}(DB)$[4] consists of a set of annotated atoms. Annotated atoms are atoms which can have "attached" update specifications. The informal reading of the atom $balance(102, 1500)$ is that it is true. Similarly the annotated atom $changebalance(Acnt, B, B') \leftarrow -balance(Acnt, B), +balance(Acnt, B')$ means that $changebalance(Acnt, B, B'))$ is true if the updates $-balance(Acnt, B)$ and $+balance(Acnt, B')$ are consistent.*

◇

The semantics of a traditional deductive database is a set of atoms. We would like to express the semantics of a logic database with updates as a set of (possibly) annotated atoms. Update specifications are collected during the marking phase and are executed in parallel during the update phase in a *all-or-nothing* style. Thus, our framework nicely fits with the expected behavior of a transaction. This results in the fact that our approach can be very efficient and simple to implement. Indeed, due to the two phases computation there is no transaction rollback, and the updates can be executed in parallel. There is no transaction rollback at logical level, because there is no need to undo updates or apply other equivalent techniques

[4]We denote the syntactic domain in "`typewriter`" style and the semantic domain in "roman" style.

if the transaction aborts. For instance the transaction ? `transfer(2000, 102, 105)` aborts because the amount is over `1500`. This abort is detected as a query failure due to the update specification. In other words, we can check if a transaction aborts during the marking phase avoiding useless updates executions. Note that these features are related to the semantics of the language according to [19, 39]. However, due to failures at physical level, which are not addressed in this paper, log information are required to ensure data integrity [32]. Nevertheless, the fact that only for errors at the physical level this import and inefficient utility is required, means that there is less overhead to handle the type of transaction so far introduced. Finally, notice that the transactions ? `changebalance(102, 1500, 1800)` and ? + `balance(102, 1800), changebalance(102, 1500, 1800)` are equivalent. Indeed, the approach of collecting updates considers sets of updates. Therefore, if there are duplications of updates triggered by the same query, they are "filtered" within the marking phase. In this way logical optimization to the collected updates is a by-product of this approach. Thus multiple executions of the same update inside a transaction are avoided. If instead, multiple executions are required, this is possible by introducing control at transaction level.

## 3.2 Explicit control

The proposed language up to now provides a smooth integration between the logic and update languages by avoiding control in the rule language. However, control is important in many cases. For instance to improve the expressive power of the language or simply as a way to perform sequential computations on data. Therefore, we allow explicit control constructors, but outside the rule language. These control constructors provide a simple way to define complex transactions starting from simple ones. Thus, we are able to maintain the nice declarative style while providing, at the same time, control capabilities. We consider two constructors: sequence and iteration. We consider only these constructors because they form the minimum extensions (to rule language with updates) to achieve the required expressive power for database languages according to [2]. The first constructor is sequential transaction composition (denoted by ";"), it performs a sequence of transactions as a single transaction.

EXAMPLE 3.3. *Consider a transaction that makes a deposit, then transfers half of the resulting amount into another account. This transaction is modeled as a sequence of simple transactions as follows.*

? `deposit(600, 102)`; ? `balance(102, Amt), transfer(Amt/2, 102, 105)`

*The transaction first performs first* ? `deposit(600, 102)` *and only after the execution of this transaction,* ? `balance(102, Amt), transfer(Amt/2, 102, 105)` *is performed. Thus the second transaction sees the updates performed from the first transaction. The sequence is executed as a transaction itself, that is if any of the components of the sequence, in this case deposit or transfer aborts, then the large transaction aborts.* ◇

The second constructor (denoted by "while" ) performs iteration. It is useful to express iterations where at each iteraction step one must perform a sequence of updates. Indeed, due to the set-oriented computation the rule language already provides a facility to delete a set of tuples which satisfy a given condition. However, the operations over each tuple form a set, therefore, there is no way to execute them in sequence. This approach can be inadequate for expressing iterations where the update operations within the same iteraction step must be performed in sequence. The next example clarifies this point.

EXAMPLE 3.4. *Consider the following transaction which for each account over* `1000` *increases by* `5`% *the balance, then takes a commission of* `1` *and finally transfers the exceeding money over* `1000` *into a saving account.*

```
? while balance(Acnt,Amt),Amt>1000 do
        ? deposit(Acnt,1.05*Amt); ? withdraw(1,Acnt);
        ? balance(Acnt,Amt),transfer(Amt-1000,Acnt,SaveAcnt)
                                    endo
```

◇

For each account with an amount greater then `1000`, three operations are performed in sequence: deposit, withdraw and transfer. This kind of computation is not possible without the iteration constructor.

# 4 U-Datalog language

In this section we provide the formal definition of the language introduced by means of examples in the previous section. The language is defined as an extension of Datalog language, called U-Datalog to express updates with non-immediate semantics. We then extend this language to accommodate explicit constructors outside the rule language. U-Datalog has the following characteristics:

1. The marking phase, that is the query-answering process, computes the bindings for the variables of the transaction and the set of updates to be performed. This phase can be executed both top-down or bottom-up.

2. The update phase performs these updates collected during the marking phase, if they are consistent, otherwise the updates are not executed.

To model the marking phase we need a formal framework to collect update specification and to check their consistency. We choose the constraint logic programming, (CLP(X)) [23] which is the result of the integration between constraint and logic programming. U-Datalog is just a special instance of CLP with some syntactic restrictions. We extend the Datalog rules by allowing *updates atoms* to extensional relations of the form $\pm p(t_1, \ldots, t_n)$. By contrast,the Datalog atoms are called *query atoms.*

A state or extensional database $EDB$ is a (possibly empty) set of base relations. We denote with $EDB_i, i = 1, \ldots, n$ the possible extensional databases.

DEFINITION 4.1. *The intensional database $IDB$ is a set of rules of the form*

$$H \leftarrow U_1, \ldots, U_s, B_1, \ldots, B_t.$$

*where $B_1, \ldots, B_t$ (as in Datalog) are base or derived atoms, $H$ is a derived atom and $U_1, \ldots, U_s$ are update atoms.*

The above definition needs some comments. The order of rule atoms is irrelevant. However, to avoid complex index notation we group them into update and query atoms. This is just a convention. The intuitive meaning of one of this rule is: "if $B_1, \ldots, B_t$ is true and the updates $U_1, \ldots, U_s$ are consistent (i.e. not complementary), then $H$ is true". The updates $+\mathtt{p(X)}, -\mathtt{p(X)}$ are complementary updates. The updates $+\mathtt{p(Y)}, -\mathtt{p(X)}$ could become complementary if for instance $\mathtt{Y/tom}, \mathtt{X/tom}$.

DEFINITION 4.2. *A transaction is a rule with no head of the form*

$$U_1, \ldots, U_s, B_1, \ldots, B_t$$

*where the atoms are as in Definition 4.1.*

In the examples we always prefix with the symbol '?' a transaction which also has a query facility. In the following we use a notion of database safety which needs some comments. A database is safe if its rules are range restricted, that is every variable in the head also occurs in query atoms. However top-down evaluation allows a less restrictive definition of safety. For instance the rule $\mathtt{love(X,Y)} \leftarrow \mathtt{nice(X)}$ is not safe, but the query $?\,\mathtt{love(X,bob)}$ renders it safe. In our case a transaction subsumes a query. Thus we consider databases which are safe through transaction invocation, that is, if every rule in the database is either range restricted or rendered safe from transaction invocation.

DEFINITION 4.3. *An U-Datalog database $DB$ consists of the extensional database $EDB$ and of the intensional database $IDB$.*

EXAMPLE 4.1. *Consider the following database.*

$\mathtt{eds(tom, shoe, 15k)}.$
$\mathtt{eds(bob, shoe, 18k)}.$
$\mathtt{transfer(X)} \leftarrow -\mathtt{eds(X, shoe, S)}, +\mathtt{eds(X, toy, S)}, \mathtt{eds(X, shoe, S)}.$

$?\ \mathtt{transfer(X)}$ *moves all the employees from* `shoe` *to* `toy` *departments. The marking phase collects:*

- *the bindings* $\mathtt{X} = \mathtt{tom}$ *and* $\mathtt{X} = \mathtt{bob}$*;*
- *the updates* $-\mathtt{eds(tom, shoe, 15k)}, +\mathtt{eds(tom, toy, 15k)}$*,* $-\mathtt{eds(bob, shoe, 18k)}$ *and* $+\mathtt{eds(bob, toy, 18k)}$*.*

*Thus the computation is performed in a set oriented style. Similarly does the transaction* $?\ -\mathtt{eds(tom, shoe, 15k)}, \mathtt{transfer(X)}.$ *fails due to the complementary updates* $-\mathtt{eds(tom, shoe, 15k)}, +\mathtt{eds(tom, shoe, 15k)}.$ $\diamond$

## 4.1 Extended U-Datalog

Transactions defined so far can be seen as building blocks used to construct complex transactions by means of control constructors. The sequence constructor is required to model sequences and iteration is required to express iteration where we want to model a sequence of operations inside a loop. The introduction of control at transaction level can be seen as a way to allow (only in the transaction language) immediate updates semantics. Thus immediate and non-immediate update semantics can be combined taking advantage of both the approaches. For instance, transactions can be expressed with an immediate update rule language such as transaction logic [14]. This would lead to a further integration between different rule languages which is behind the scope of this paper.

DEFINITION 4.4. *A transaction is either $T$ as in Definition 4.2 or a sequence of transaction, or an iterative constructor of the form while $Q$ do $T$ done, that is*

$$T ::= T_1 \mid T_1; T_2 \mid \textit{ while } Q \textit{ do } T \textit{ endo}$$

# 5 The Semantics of U-Datalog

The semantics of the language is tailored to model the properties that are relevant for database systems. For example, in a query rule language one can be interested in modeling the set of answers that are computed as result of a query (as in Datalog), or just one of these answers (as in logic programming). This is called an *observable property*. Thus observable properties must be chosen and the semantics must faithfully express them. For example, the semantics of a sequence of updates forming a transaction is a mapping from state to state. An observable property of interest for a transaction is the new state. Similarly, the semantics of a Datalog query can be expressed as a mapping from a state to a set of answers [45]. In our approach, due to the query-update feature of the language, we are interested in modeling as transaction properties: *the set of answers*, *the database state* and *the results of a transaction*, that is abort or commit.

## 5.1 Updates Interpretation

Before defining the semantics of U-Datalog we need to give an interpretation for updates. We have already discussed non-immediate update semantics as a complement to the immediate update semantics. However, another classification of the update semantics is possible, that is weak update semantics and strong one. Weak updates are those for which there is no precondition. Consider, for example, the database state $\mathtt{p(a)}, \mathtt{p(b)}, \mathtt{q(a)}$. The insertion of $\mathtt{p(b)}$ (respectively deletion of $\mathtt{q(b)}$) does not change the database, nevertheless they are allowed.

Strong updates are those which allow to delete (insert) atoms only if they are (not) in the current database state. Consider, for instance, the above database state. The insertion of $\mathtt{q(b)}$ is allowed, whereas the deletion of $\mathtt{p(c)}$ is not. Strong updates are available in most commercial database systems, such as INGRES [24].

Since our approach is parametric with respect to strong or weak updates, we need to define the informal notion of inconsistent or complementary updates to take into account also the feature of strong/weak updates. This results in the notion of solvability. The structure to interpret strong updates is simple and reflects the above informal discussion.

DEFINITION 5.1. *A structure $\Re^s$ for strong updates consists of*

- *EDB*
- *an assignment to each update atoms $\pm p(\tilde{t})$ with $\tilde{t}$ ground term such that:*
  1. $+p(\tilde{t}) = True$ *iff* $p(\tilde{t}) \notin EDB$
  2. $-p(\tilde{t}) = True$ *iff* $p(\tilde{t}) \in EDB$

*Assignments to constant symbols and syntactic equality are as usual. An $\Re^s$-valuation for update atoms is a mapping $\theta$ from variables to terms.*

DEFINITION 5.2. *Updates atoms $U_1, U_2$ are solvable iff there exists a valuation $\theta$ such that $\Re \models (U_1, U_2)\theta$. $\theta$ is called the $\Re$-solution of $U_1, U_2$.*

The above solvability notion is referred to $\Re$, thus to any structure for strong or weak updates. For strong updates it checks whether two updates are complementary, and whether the inserted information already are in the database or not, according to strong update semantics. For weak updates (see below the related structure) it checks only if there are complementary updates.

EXAMPLE 5.1. *Consider the database with strong update semantics*

```
p(a).
q(b).
r(X) ← +p(X), q(X).
s(X) ← +q(X), q(X).
```

*The query* ? `s(X)` *fails due to the unsolvability of* `+q(b)`, *while* ? `r(X)` *succeeds due to the solvability of* `+p(b)`.

◇

By contrast, no preconditions are required for weak updates. Therefore, the domain is the Herbrand universe.

DEFINITION 5.3. *A structure $\Re^w$ for weak updates consists of an assignment to each predicate atoms $\oplus p(\tilde{t}_i)$ and $\ominus p(\tilde{t}_i)$ with $\tilde{t}_i$ ground term such that:*

1. $\oplus p(\tilde{t}_1) = True$
2. $\ominus p(\tilde{t}_1) = True$

3. $\vdots$

4. $\ominus p(\tilde{t}_1), \oplus p(\tilde{t}_2) = True$

5. $\ominus p(\tilde{t}_1), \oplus q(\tilde{t}_2) = True$

6. $\ominus p(\tilde{t}_1), \oplus q(\tilde{t}_1) = True$

7. $\vdots$

8. $\ominus p(\tilde{t}_1), \oplus q(\tilde{t}_1), \ldots, \ominus t(\tilde{t}_1) = True$

*False in case of existence of* $\ominus p(\tilde{t})$ *and* $\oplus p(\tilde{t})$ *An* $\Re^w$*-valuation for updates atoms is a mapping* $\theta$ *from variables to term.*

The above solvability notion checks whether two updates are complementary. Here there is no need to check whether the two updates are already in the database due to the weak update semantics. Note that instead of the explicit interpretation of all non-atomic formulae of the update language (items from $(d)$ to $(j)$ of Definition 5.3), we can synthesize these items as

$$\delta A_1, \ldots, \gamma A_s = \begin{cases} false & \text{if } \exists i, j, 0 \leq i, j \leq s \text{ such that} \\ & A_i = A_j \wedge \delta \neq \gamma \\ true & otherwise \end{cases}$$

where $\delta, \gamma \in \{\oplus, \ominus\}$ and $A_i, A_j$ are atoms. The notion of solvability is the same of Definition 5.2[5]. Note that the structure for weak updates does not change over time whereas that for strong updates changes in order to consider state evolution.

EXAMPLE 5.2. *Consider the database of Example 5.1 with weak update semantics*

```
p(a).
q(b).
r(X) ← ⊕p(X), q(X).
s(X) ← ⊕q(X), q(X).
```

*The query* ? `s(X)` *succeeds due to the solvability of the* $\oplus$`q(b)`. *The new state is equal to the current database state.* ? `r(X)` *succeeds due to the solvability of* $\oplus$`p(b)`. $\diamond$

Finally, note that Definition 5.2 reflects the algorithmic computation that checks whether a set of updates is solvable and its complexity is linear in the number of updates. In addition the strong updates implementation should verify the precondition.

[5]To highlight the difference between the strong and weak updates we use the symbols $-, +$ and $\ominus, \oplus$ respectively.

## 5.2 The Four Steps Semantics

The semantics of U-Datalog is defined in four steps. Each step relies on the previous one and is denoted with $\mathcal{S}_i, i = 1...4$. As we have said, the database consists of two components. Therefore we would like to give the semantics of the database in term of that of the components. This is the first step semantics called *compositional semantics*. Compositionality is well known in logic programming [15]. Since we have the marking and update phases we need to model these phases. The second step semantics, called *marking phase semantics*, models the former. It is defined through operational and equivalent fixpoint semantics. Many of the already existing algorithms for efficient query evaluation strategies rely on the equivalence between top-down and bottom-up computations, thus on the equivalence between operational and fixpoint semantics [5]. Informally, they take advantage of the constants in the query to cut the search space (through top-down evaluation) and compute the set of answers (through bottom-up evaluation). From here the choice of providing operational and fixpoint semantics for U-Datalog. We do not consider the model theoretic semantics because it does not provide a guideline for the computational model. The first and second steps together model the specification and collection of updates through operational and fixpoint semantics. They are a special case of a general one, that is CLP semantics. Thus in this section we provide the necessary results for U-Datalog which are instances of the CLP schema[6]. The necessary extensions to CLP for handling the first step semantics are provided in Appendix B. The third step semantics, called *update phase semantics*, instead, receives as input the collected updates computed by the marking phase and executes them computing the new database state. In addition, the result of the transaction says whether the transaction commits or aborts and provides the set of answers for the query part. In order to model complex transactions we consider a fourth step semantics called *complex transaction semantics*. The semantics of complex transactions is defined in terms of the semantics of the simple transactions and the control constructors. In addition to the four step semantics, we provide equivalence relations for transactions, and for databases.

## 5.3 Compositional Semantics

We are interested in modeling the semantics of $IDB$ modulo the possible $EDB's$. In the following we consider $IDB$ and $EDB$ as the components of the database and consider only the necessary results instantiated on U-Datalog. To help the reader, we provide into brackets the reference of the general case (i.e. CLP) given in Appendix B. The composition we consider is union, denoted by $\cup$. Since in our case we are interested in importing base relations in the intensional database, the $\cup$ operator is applied to the extensional database and to the intensional database. In order to define the compositional semantics we need to introduce some notions. $U_1, \ldots, U_s, G_1, \ldots, G_n \rightsquigarrow_{DB} U'_1, \ldots, U'_s, B_1, \ldots, B_t$ denotes the partial derivation, that is a number of derivation step of $U'_1, \ldots, U'_s, B_1, \ldots, B_t$ from $U_1, \ldots, U_s, G_1, \ldots, G_n$

[6] An introduction to CLP and its results are in Appendix A.

by means of the rules of $DB$. $Id$ denotes a set of rules of the form $A \leftarrow A$ where $A$ is a base atom. Thus the denotation of a database is a set of rules where the update and query parts are the result of a partial derivation performed with respect to $DB$ in the former part and then respect to $DB^+ = DB \cup Id$.

DEFINITION 5.4. *[B.10] Let $DB$ be an U-Datalog database, $\tilde{U} = U_1, \ldots, U_m$ be a set of updates, $\tilde{B} = B_1, \ldots, B_n$ be a set of base atoms and $\tilde{A} = A_1, \ldots, A_p$ be a set of atoms. Moreover, let $DB^+ = DB \cup Id$. Then the compositional semantics is*

$$\begin{aligned}\mathcal{S}_1(DB) = \{ \quad & p(\tilde{X}) \leftarrow \tilde{U}', \tilde{B} \mid \\ & p(\tilde{X}) \rightsquigarrow_{DB} \tilde{U}, \tilde{A} \rightsquigarrow_{DB^+} \tilde{U}', \tilde{B}, \\ & \textit{and the predicate of atoms in } \tilde{B} \\ & \textit{are a subset of base predicates} \quad \}.\end{aligned}$$

Note that the above definition computes the semantics domain as a set of rules. Informally, this is a mapping stating that the truth of something is conditionated to the truth of something else. This is remarkably different from the traditional semantics where the domain is a set of ground atoms which are always true. Not surprisingly the semantics domain turns to be a program and therefore we can use the same composition operator over semantic domains.

The following theorem states the necessary result about programs composition on one side and semantics composition on the other. This due to the fact that the semantics of a program is a program itself.

THEOREM 5.1. *[B.3] Let $EDB$ and $IDB$ be extensional and intensional databases. Then $\mathcal{S}_1(EDB \cup IDB) = \mathcal{S}_1(\mathcal{S}_1(EDB) \cup \mathcal{S}_1(IDB))$.*

The following example illustrate the above notion.

EXAMPLE 5.3. *Consider the database components:*

$$\begin{aligned}EDB_1 = \quad & \texttt{q(b).} \\ & \texttt{t(a).}\end{aligned} \qquad \begin{aligned}IDB = \quad & \texttt{p(X)} \leftarrow -\texttt{q(X)}, \texttt{q(X)}. \\ & \texttt{r(X)} \leftarrow +\texttt{t(X)}, \texttt{p(X)}. \\ & \texttt{k(X)} \leftarrow -\texttt{t(X)}. \\ & \texttt{s(X)} \leftarrow \texttt{t(X)}.\end{aligned}$$

*Their semantics are:*

$$\begin{aligned}\mathcal{S}_1(EDB_1) = \quad & \{q(b), \\ & t(a) \}\end{aligned} \qquad \begin{aligned}\mathcal{S}_1(IDB) = \{ \quad & p(X) \leftarrow -q(X), q(X), \\ & r(X) \leftarrow +t(X), -q(X), q(X), \\ & k(X) \leftarrow -t(X), \\ & s(X) \leftarrow t(X) \quad \}\end{aligned}$$

*Considering $DB = EDB_1 \cup IDB$ we have $\mathcal{S}_1(EDB_1 \cup IDB) = \mathcal{S}_1(\mathcal{S}_1(EDB_1) \cup \mathcal{S}_1(IDB))$. Therefore*

$$\mathcal{S}_1(EDB_1 \cup IDB) = \{ \quad p(X) \leftarrow -q(X), q(X),$$
$$r(X) \leftarrow +t(X), -q(X), q(X),$$
$$k(X) \leftarrow -t(X),$$
$$s(X) \leftarrow t(X),$$
$$p(b) \leftarrow -q(b),$$
$$r(b) \leftarrow +t(b), -q(b),$$
$$s(a),$$
$$t(a),$$
$$q(b) \quad \}$$

$\diamond$

## 5.4 Marking Phase Semantics

Example 5.3 shows the compositional semantics of intensional and extensional databases. Such semantics contains rules in the semantics domain. To provide the marking phase semantics, we are no more interested to keep the distinction between the intensional and extensional components. The semantic domain does not need anymore to be a set of rules. It is a set of atoms with update specifications. For instance $p(b) \leftarrow -q(b)$ says that the atom $p(b)$ is true and the update $-q(b)$ can be the result of the marking phase if $p(b)$ is involved in the query-answering process. Thus the semantics of marking phase has as semantics domain a set of atoms annotated with update specifications (both possibly non-ground). Since the marking phase executes the query-answering process and this was shown to be efficiently computable, if there is an operational and equivalent fixpoint semantics we provide such semantics denoting them with $\mathcal{S}_2^o$ and $\mathcal{S}_2^f$ respectively. Thus the marking phase can be computed both in top-down or equivalently bottom-up style. We denote a successful derivation of a transaction $T$ in a database $DB$ which collects the update specifications $U'$, by $T \longrightarrow^*_{DB} U'$.

DEFINITION 5.5. *[A.14] Let $DB$ be a database. The operational semantics $\mathcal{S}_2^o(DB)$ is defined as follows*

$$\mathcal{S}_2^o(DB) = \quad \{p(\tilde{X}) \leftarrow \tilde{U} \quad \mid p(\tilde{X}) \longrightarrow^*_{DB} \tilde{U} \quad \}.$$

In order to provide the fixpoint semantics we introduce the immediate consequence operator. Such operator computes a step of forward computation collecting updates and passing bindings.

DEFINITION 5.6. *[A.15] Let $DB$ be a database and let $J$ be an interpretation, then the immediate consequence operator is*

$$
\begin{aligned}
T_{DB}(J) = \{ \quad & p(\tilde{X}) \leftarrow \tilde{U} \mid \\
& \exists \text{ a renamed rule} \\
& p(\tilde{t}) \leftarrow U_0, p_1(\tilde{t}_1), \ldots, p_n(\tilde{t}_n) \text{ in } DB \\
& \forall i = 1...n,\ \exists \theta (p_i(\tilde{X}_i) \leftarrow U_i) \in J, 1 \leq i \leq n \\
& \text{which share no variables such that} \\
& p(\tilde{X}) = \theta p(\tilde{t}) \text{ and} \\
& U = U_0 \cup \{U_1, \ldots, U_n\} \\
& U \text{ is } \Re\text{-solvable} \quad \}.
\end{aligned}
$$

The above operator is continuous and thus there exists a least fixpoint (unique by definition) which is $T_{DB} \uparrow n$ according to the following definition.

DEFINITION 5.7. *[A.17] Let $DB$ be a database. The fixpoint semantics $Fix_2(DB)$ of $DB$ is defined as $\mathcal{S}_2^f(DB) = T_{DB} \uparrow n$*

Note that by assuming a finite domain, the fixpoint is reached in a finite number of steps. Theorem 5.2 states the expected result on the relationship between the operational and the fixpoint semantics.

THEOREM 5.2. *[A.2] (Equivalence of the operational and the fixpoint semantics) Let $DB$ be a database. Then $\mathcal{S}_2^o(DB) = \mathcal{S}_2^f(DB)$.*

The above result was first given in [36] for CLP and allow us, in the context of U-Datalog, to have top-down or bottom-up computations regardless of updates. Let us now show how the fixpoint is computed.

EXAMPLE 5.4. *Consider the database of Example 5.3 then*

$$
\begin{aligned}
& T_0(IDB \cup EDB_1) = \{q(b), t(a), k(X) \leftarrow -t(X)\} \\
& T_1(IDB \cup EDB_1) = \{s(a), p(b) \leftarrow -q(b)\} \cup T_0(IDB \cup EDB_1) \\
& T_2(IDB \cup EDB_1) = \{r(b) \leftarrow +t(b), -q(b)\} \cup T_1(IDB \cup EDB_1) \\
& T_3(IDB \cup EDB_1) = T_i(IDB \cup EDB_1)(\forall i > 2) = T_2
\end{aligned}
$$

*and $\mathcal{S}_2^o(IDB \cup EDB_1) = \mathcal{S}_2^f(IDB \cup EDB_1)$.*

$\diamond$

In the following we use just $\mathcal{S}_2$ to denote the second step operational or equivalent fixpoint semantics. Marking phase and compositional semantics differ because the former has no more rules in the semantics domain. This relationship is stated by Theorem 5.3. The auxiliary function $Proj$ takes a set of rules and returns only those with empty query in the body i.e. a set of annotated atoms. Informally, this means that once that we are no more interested in having information about the modular structure of the database we can forget it, and consider only the annotated atoms.

THEOREM 5.3. *[B.4] Let $DB$ be a database. Then $\mathcal{S}_2(DB) = Proj(\mathcal{S}_1(DB))$.*

The next example illustrates this relationship.

EXAMPLE 5.5. *Consider*

$$\begin{aligned}\mathcal{S}_1(EDB_1 \cup IDB) = \{\ & p(X) \leftarrow -q(X), q(X),\\ & r(X) \leftarrow +t(X), -q(X), q(X),\\ & k(X) \leftarrow -t(X),\\ & s(X) \leftarrow t(X),\\ & p(b) \leftarrow -q(b),\\ & r(b) \leftarrow +t(b), -q(b),\\ & s(a),\\ & t(a),\\ & q(b)\ \ \}\end{aligned}$$

*of the Example 5.3 and*

$$\begin{aligned}\mathcal{S}_2(EDB_1 \cup IDB) = \{\ & p(b) \leftarrow -q(b),\\ & r(b) \leftarrow +t(b), -q(b),\\ & k(X) \leftarrow -t(X),\\ & s(a),\\ & t(a),\\ & q(b)\ \ \}\end{aligned}$$

*of the Example 5.4. Then, according to Theorem 5.3, we have* $\mathcal{S}_2(EDB_1 \cup IDB) = Proj(\mathcal{S}_1(EDB_1 \cup IDB))$.

◇

Note that the semantic domain of the above example is still more general than the traditional one, indeed it is a set of non-ground (annotated) atoms. For instance $k(X) \leftarrow -t(X)$. This allow us to insert new values in the active domain of the database, that is in the set of constants stored in the current database state. Note, however, that the updates $-t(X)$ can become ground due to transaction transmission of the parameter.

## 5.5 Update Phase Semantics

The semantics of the marking phase does not include the execution of the collected updates neither provide the transaction semantics. This is done in the update phase semantics. Before providing such semantics we need some preliminar notions. First we note that database systems use as default a set-oriented query-answering process denoted as

$$Set(T, DB) = \{\langle Binding_j, U_j\rangle \mid T \rightarrow^*_{DB} \langle Binding_j, U_j\rangle\ \ \}$$

The set of pairs (bindings and updates) is computed as answers to $T$ by the marking phase. Second, we define a function which takes a set of ground updates, the current database state and returns the new state.

DEFINITION 5.8. *Let $EDB_i \in States$ be the current database and $U \in Upd$ is a consistent set of ground updates. Then the new database $EDB_{i+1}$ is computed by means of the function $\Delta : States \times Upd \rightarrow States$ as follows:*

$$\Delta(EDB_i, u) = (EDB_i \setminus \{p(\tilde{t}) \mid -p(\tilde{t}) \in u\}) \cup \{p(\tilde{t'}) \mid +p(\tilde{t'}) \in u\}$$

*where $States$ is the set of possible database states and $Upd$ is the set of possible updates.*

According to the definition of the structure for strong updates, each time a new state is computed we change the structure itself which gives us the interpretation of the updates. Therefore, given a sequence of states $EDB_1; EDB_2; \ldots; EDB_n$, there exists a corresponding sequence of structures $\Re^s_1; \Re^s_2; \ldots; \Re^s_n$. This is not the case for weak updates where the structure does not change. Third, we denote the observable property of a transaction as $Oss_i = \langle Ans, State, Res \rangle$ where $Ans$ is a set of answers. $State$ is the database state, and $Res$ is the transaction result, that is either Commit or Abort. The set of possible observables is $OSS$. Finally we are ready for the update phase semantics.

DEFINITION 5.9. *Let $DB_i = EDB_i \cup IDB$ be the database. The semantics of a transaction is denoted by the function $\mathcal{S}_{3,IDB}(T)$ ($\mathcal{S}_3(T)$ for short) from $OSS$ to $OSS$. If $T$ is a transaction, then*

$$\mathcal{S}_3(T)(EDB^i) = \begin{cases} Oss^{i+1} & \text{if } OK \\ \langle \emptyset, EDB^i, Commit \rangle & ungroundness \\ \langle \emptyset, EDB^i, Abort \rangle & inconsistency \end{cases}$$

*where $Oss_{i+1} = \langle \{Binding_j \mid \langle Binding_j, U_j \rangle \in Set(T, DB_i)\}, EDB_{i+1}, Commit \rangle$, $EDB_{i+1}$ is $\Delta(EDB_i, \bar{U})$. The condition $OK$ expresses the fact that the set $\bar{U} = \bigcup_j U_j Binding_j$ has no complementary ground updates. $U_j Binding_j$ denotes the ground updates obtained by substituting the variables of $U_j$ with the ground terms associated with the variables in $Binding_j$. The condition ungroundness of non-ground updates turns the behavior of a transaction into a "no operation", while that of inconsistency into an abort.*

Note that according to the above definition, in U-Datalog the abort of a transaction may be caused by a transaction that generate an update set with complementary updates on the same atom. In this case, the resulting state would depend on the execution order of updates, so we disallow this situation by aborting the transaction. The second situation (which me may call *no operation for ungroundness*) is related to a transaction that generates a non-ground set of update. In this case we are not able to decide what update to execute, and therefore we do not execute any update.

EXAMPLE 5.6. *Consider the database* $DB_1 = EDB_1 \cup IDB$, *where*

$$EDB_1 = \begin{array}{l} \texttt{q(b).} \\ \texttt{t(a).} \end{array} \quad IDB = \begin{array}{l} \texttt{p(X)} \leftarrow -\texttt{q(X)}, \texttt{q(X)}. \\ \texttt{r(X)} \leftarrow +\texttt{t(X)}, \texttt{p(X)}. \\ \texttt{k(X)} \leftarrow +\texttt{t(X)}. \\ \texttt{s(X)} \leftarrow \texttt{t(X)}. \end{array}$$

$\mathcal{S}_2(DB_1) = Proj(\mathcal{S}_1(IDB) \cup \mathcal{S}_1(EDB_1)))$

$$\mathcal{S}_2(DB_1) = \{ \begin{array}{l} k(X) \leftarrow +t(X), \\ p(b) \leftarrow -q(b), \\ r(b) \leftarrow +t(b), -q(b), \\ s(a), \\ t(a), \\ q(b) \quad \} \end{array}$$

- *Let* $Oss_1 = \langle \emptyset, EDB_1, Commit \rangle$. *The semantics of* $T_1 = ?$ $\texttt{r(X)}$ *is*

  $$\mathcal{S}_3(T_1)(Oss_1) = \langle \{X = b\}, EDB_2, Commit \rangle$$

  *where* $EDB_2 = \{t(a), t(b)\}$ *and* $\mathcal{S}_2(DB_2) = Proj(\mathcal{S}_1(\mathcal{S}_1(IDB) \cup \mathcal{S}_1(EDB_2)))$

  $$\mathcal{S}_2(DB_2) = \{ \begin{array}{l} k(X) \leftarrow +t(X), \\ s(a), \\ s(b), \\ t(a), \\ t(b) \quad \} \end{array}$$

- *The semantics of* $T_2 = ?$ $\texttt{s(X)}$ *is*

  $$\mathcal{S}_3(T_2)(Oss_2) = \langle \{\{X = a\}, \{X = b\}\}, EDB_2, Commit \rangle$$

- *The semantics of* $T_3 = ?$ $\texttt{k(c)}$ *is*

  $$\mathcal{S}_3(T_3)(Oss_2) = \langle \{X = c\}, EDB_3, Commit \rangle$$

  *with* $EDB_3 = \{t(a), t(b), t(c)\}$ *and* $\mathcal{S}_2(DB_3) = Proj(\mathcal{S}_1(\mathcal{S}_1(IDB) \cup \mathcal{S}_1(EDB_3)))$

  $$\mathcal{S}_2(DB_3) = \{ \begin{array}{l} k(X) \leftarrow +t(X), \\ s(c), \\ s(b), \\ s(a), \\ t(c), \\ t(b), \\ t(a) \quad \} \end{array}$$

- *The semantics of* $T_4 =$ ? $+\mathtt{t(a)}, \mathtt{s(a)}$ *is*

  $$\mathcal{S}_3(T_4)(Oss_3) = \langle \emptyset, EDB_3, Abort \rangle$$

  *due to the fact that* $t(a)$ *is already in the database.*

◇

## 5.6 The Semantics of Complex Transactions

Complex transactions built by means of sequence and iteration constructors are not modeled by the semantics so far introduced. Thus we define their semantics semantics over that of simple transactions.

DEFINITION 5.10. *Let* $DB_i$ *be the database and* $T_1; T_2$ *be a complex transaction. The semantics of* $T_1; T_2$ *is denoted by the function* $\mathcal{S}_4(T_1; T_2)$ *from* $OSS$ *to* $OSS$.

$$\mathcal{S}_4(T_1; T_2)(Oss_i) = \begin{cases} Oss_{i+2} \text{ if } OK \\ \\ \langle \emptyset, Oss_i.2, Abort \rangle \quad otherwise \end{cases}$$

*where* $Oss_{i+2} = \mathcal{S}_3(T_2)(Oss_{i+1})$. $Oss_{i+1} = \mathcal{S}_3(T_1)(Oss_i)$ *represents the observable of the database after the transaction* $T_1$ *and* $OK$ *expresses the condition that* $\mathcal{S}_3(T_2)(Oss_{i+1}).3 = Commit$ [7] *and* $\mathcal{S}_3(T_1)(Oss_i).3 = Commit$.

Note that the above semantics does not model the answers of the first transaction. We choose this approach to avoid keeping the histories of the transactions. However, the semantics can be easily changed to embody the histories of the transaction.

EXAMPLE 5.7. *Consider the database*

$$EDB_1 = \begin{array}{l} \mathtt{q(b).} \\ \mathtt{t(a).} \end{array} \quad IDB = \begin{array}{l} \mathtt{p(X)} \leftarrow -\mathtt{q(X)}, \mathtt{q(X)}. \\ \mathtt{r(X)} \leftarrow +\mathtt{t(X)}, \mathtt{p(X)}. \\ \mathtt{k(X)} \leftarrow -\mathtt{t(X)}. \\ \mathtt{s(X)} \leftarrow \mathtt{t(X)}. \end{array}$$

- *Let* $Oss_1 = \langle \emptyset, EDB_1, Commit \rangle$. *The semantics of* $T =$ ? $\mathtt{r(X)}$; ? $\mathtt{s(X)}$ *is*

  $$\mathcal{S}_4(T)(Oss_1) = \langle \{\{X = a\}, \{X = b\}\}, EDB_2, Commit \rangle$$

  *where* $EDB_2 = \{t(a), t(b)\}$.

[7] $Oss.n$ denotes the $n-th$ component of the tuple $Oss$.

- *The semantics of* $T' =$ ? `r(X);? s(X);? k(a)`
  *is*

  $$\mathcal{S}_4(T')(Oss_1) = \langle \{X = a\}, EDB_3, Commit \rangle$$

  *with* $EDB_3 = \{t(b)\}$.

- *The semantics of* $T'' =$ ? `r(X);? s(X);? k(a);? p(a)` *is*

  $$\mathcal{S}_4(T'')(Oss_1) = \langle \emptyset, EDB_1, Abort \rangle$$

  *due to strong update semantics.*

◇

The semantics for the iteration constructor is defined below.

DEFINITION 5.11. *Let* $DB_i$ *be a database,* $Q$ *a query and* $T$ *a transaction. The semantics of a complex transaction* $T' =$ `while` $Q$ `do` $T$ `endo` *is denoted by the function* $\mathcal{S}_4(T') : OSS \longrightarrow OSS$, *where*

$$\mathcal{S}_4(T')(Oss_i) = \begin{cases} Oss_{i+n} & \text{if } OK \\ \\ \langle \emptyset, Oss_i.2, Abort \rangle & otherwise \end{cases}$$

*OK expresses the condition that there exists* $n \geq 0$ *such that for each* $j$ $(i \leq j \leq i+n)$ $\mathcal{S}_4(T')(Oss_j).3 = Commit$ *and such that*

1. *for each* $j$, $(i \leq j < i+n)$ $\mathcal{S}_2(Q)(Oss_j).1 \neq Binding$
2. *for each* $j$ $(i \leq j < i+n)$ $\mathcal{S}_4(T)(Oss_j) = Oss_{j+1}$ *and*
3. $\mathcal{S}_2(Q)(Oss_{i+n}).1 = Binding$.

Note that the query $Q$ expresses just a condition. If $Q$ has updates or triggers updates in the database they are not executed.

EXAMPLE 5.8. *Consider the following database*

$$EDB_1 = \begin{array}{l} \texttt{q(c).} \\ \texttt{q(b).} \\ \texttt{q(a).} \end{array} \quad IDB = \begin{array}{l} \texttt{p(X)} \leftarrow \ominus\texttt{q(X)}, \texttt{q(X)}. \\ \texttt{r(X)} \leftarrow \oplus\texttt{t(X)}, \texttt{p(X)}. \\ \texttt{k(X)} \leftarrow \oplus\texttt{q(X)}. \\ \texttt{s(X)} \leftarrow \texttt{t(X)}. \end{array}$$

*The semantics of* $T =$ `while ? q(X) endo ? r(X);? k(d)` *done is*

$$\mathcal{S}_4(T)(Oss_1) = \langle \{X = d\}, EDB_3, Commit \rangle$$

*At the end of the first iteration* $EDB_2$ *is* `t(a),t(b),t(c),q(d)`. *After the second and last iteration* $EDB_3 =$ `t(a),t(b),t(c),t(d)`.

◇

## 5.7 Equivalence Notions

We have defined the semantics of U-Datalog modeling the expected behavior. The first motivation to define a semantics is to formally specify the behavior of a database system. However, it should also provide a theoretical framework to study practical issues. This leads to the second motivation for defining a formal semantics, that is to study equivalences for optimization issues. We recall that equivalence for database languages have only been investigated into the separate contexts of query language for Datalog language and transactions for relational systems [4, 45]. By contrast, our investigation can be regarded as the analog of transactions for deductive databases with updates, transaction and control structures. Indeed we can accommodate, inside transactions, queries with updates and control constructs such as sequence and iteration. To this purpose we provide two equivalence notions. The first states the equivalence between databases while the second one is between transactions.

DEFINITION 5.12. *(Database equivalence) Let $IDB, IDB'$ be intensional databases, $\mathcal{S}_4, IDB$ and $\mathcal{S}_4, IDB'$ their semantics and $T$ be any transaction. Then $IDB \approx IDB'$ (are equivalent) iff $\forall Oss_i \in OSS$, $\mathcal{S}_4, IDB(T)(Oss_i) = \mathcal{S}_4, IDB'(T)(Oss_i)$.*

According to the above definition, two intensional databases are equivalent with respect to a given transaction $T$ if the result is the same for any possible database state.

DEFINITION 5.13. *(Transaction equivalence) Let $IDB$ be an intensional database and $T_1, T_2$ be transactions. Then $T_1 \approx T_2$ (are equivalent) iff $\forall Oss_i \in OSS$, $\mathcal{S}_4, IDB(T_1)(Oss_i) = \mathcal{S}_4, IDB(T_2)(Oss_i)$*

According to the above definition, two transactions are equivalent with respect to a given intensional database if their results are the same for any possible database state. Talking of equivalence in database systems one is interested in checking such property. However, a fundamental negative result has shown that in general the equivalence between two queries is undecidable in an unrestricted framework [46]; similarly the equivalence for the while construct is undecidable. Thus we consider the yet interesting context of finite domains according to [29] without the while construct. In this restricted framework the equivalence notions are decidable.

EXAMPLE 5.9. *Consider the following databases*

$$
\begin{aligned}
IDB_1 = \quad & \mathtt{p(X)} \leftarrow -\mathtt{q(X)}, \mathtt{q(X)}. \\
& \mathtt{r(X)} \leftarrow +\mathtt{t(X)}, \mathtt{p(X)}. \\
& \mathtt{s(X)} \leftarrow \mathtt{t(X)}.
\end{aligned}
$$

$$
\begin{aligned}
IDB_2 = \quad & \mathtt{p(X)} \leftarrow -\mathtt{q(X)}, \mathtt{q(X)}. \\
& \mathtt{r(X)} \leftarrow +\mathtt{t(X)}, -\mathtt{q(X)}, \mathtt{q(X)}. \\
& \mathtt{s(X)} \leftarrow \mathtt{t(X)}.
\end{aligned}
$$

*$IDB_1$ and $IDB_2$ are equivalent. Consider the following transactions:*

$T_1 = ?\ \mathtt{s(X)}; ?\ + \mathtt{t(X)}, \mathtt{p(X)}$

$T_2 = ?\ \mathtt{r(X)}$

*$T_1$ and $T_2$ are equivalent.*

◇

The generality introduced by our approach is related to the language and its semantics which model complex transactions, with control construct, queries and updates in a uniform way, while preserving equivalence of top-down and bottom-up computation models. In addition, our approach to equivalence is not only uniform with respect to queries or transactions, but it is also uniform with respect to transactions or databases. A prototype for U-Datalog with a bottom-up computational model has been developed in [6]. The above equivalence notions are used to develop algorithms to check equivalences and to perform optimization through transaction transformation [7, 8].

# 6 Conclusions

In this paper we have presented a new approach to define update in declarative query languages based on non-immediate update semantics. Such language is control free and this allow us to use top-down and equivalent bottom-up computational models. Thus we can reuse the already developed techniques for Datalog query evaluation. Updates can be performed in parallel under our approach. Thus a potentially efficient system can take full advantage of this feature. Control constructs are defined at transactional level preserving the nice declarative property of rule language. We provide a formal semantics that was designed from the very beginning to take advantage of already developed concepts and to provide a formal framework into which investigate equivalence and thus optimization. The resulting approach is suitable for object-oriented [9, 10, 11] and semantics integrity constraints extensions [42].

**Acknowledgment** We would like to thank the anonymous referees who provided very helpful comments and suggestions.

# References

[1] S. Abiteboul. Updates, a New Frontier. In M. Gyssens, J.Paredaens, and D. Van Gucht, editors, *Proc. Second Int'l Conf. on Database Theory*, volume 326 of *Lecture Notes in Computer Science*, pp. 1–18. Springer-Verlag, Berlin, 1988.

[2] S. Abiteboul and V. Vianu. A Transaction Language Complete for Database Update and Specification. In *Proc. of the ACM Symposium on Principles of Database Systems*, pp. 260–268. ACM, New York, USA, 1987.

[3] S. Abiteboul and V. Vianu. Procedural and Declarative Database Update Languages. In *Proc. of the ACM Symposium on Principles of Database Systems*, pp. 240–251. ACM, New York, USA, 1988.

[4] S. Abiteboul and V. Vianu. Equivalence and Optimization of Relational Transactions. *Journal of the ACM*, 35(1):70–120, January 1988.

[5] F. Bancilhon and R. Ramakrishnan. Performance Evaluation of Data Intensive Logic Programs. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pp. 439–519, Morgan-Kaufmann, 1987.

[6] E. Bertino, B. Catania, G. Guerrini, M. Martelli, and D. Montesi. A Bottom-Up Interpreter for a Database Language with Updates and Transactions. In *Joint Conference on Declarative Programming GULP- PRODE*, Peniscola (Spain), pp. 206–220, 1994.

[7] E. Bertino, B. Catania, G. Guerrini, and D. Montesi. Static Analysis of Transactional Intensional Database. In *Proc. Second Workshop on Deductive Databases of International Conference on Logic Programming*, pp. 57–73, Genova, 1994.

[8] E. Bertino, B. Catania, G. Guerrini, and D. Montesi. Transaction Optimization in Rule Databases. In *Fourth IEEE Research Issues in Data Engineering: Active Database Systems (RIDE-ADS'94)*, pp. 137–145, Houston, 1994.

[9] E. Bertino, G. Guerrini and D. Montesi. Deductive Object Databases, In *Proc. European Conference on Object Oriented*, Lecture Notes in Computer Science 821, pp. 213–235, Bologna, 1994.

[10] E. Bertino, G. Guerrini and D. Montesi. Generic Methods for Deductive Object Databases, To appear in *Proc. International Conference on Object Oriented Information Systems*, London, 1994.

[11] E. Bertino, G. Guerrini and D. Montesi. Towards Deductive Object Databases, To appear in *Journal of Theory and Pratice of Object Systems*, John Wiley & Sons, 1994.

[12] E. Bertino, M. Martelli, and D. Montesi. Modeling Database Updates with Constraint Logic Programming. In U. W. Lipeck and B. Thalheim, editors, *Proc. Fourth Int'l Work. on Foundations of Models and Languages for Data and Objects*, pp. 120–132, Dagstuhl, 1992.

[13] B. Bertolino, P. A. Bonatti, D. Montesi, and S. Pelagatti. Correctness and Completeness of logic programs under the CLP schema. In P. Asirelli, editor, *Proc Sixth Italian Conference on Logic Programming*, pp. 391–405, Pisa, 1991.

[14] A. J. Bonner and M. Kifer. An Overview of Transaction logic programming. To appear in *Theoretical Computer Science*, 1994. Other papers on the subject are available in *pub/bonner/transaction.logic* by anonymous ftp to *db.toronto.edu*.

[15] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.

[16] M. L. Brodie. On Modelling Behavioural Semantics of Databases. In C. Zaniolo and C. Delobel, editors, *Proc. Seventh Int'l Conf. on Very Large Data Bases*, pp. 32–42, 1981.

[17] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5.* Addison-Wesley, 1985.

[18] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog - And Never Dared to Ask. *IEEE Tran. on Knowledge and Data Eng.*, 1(1):146–164, March 1989.

[19] W. Chen. Declarative Specification and Evaluation of Database Updates. In C. Delobel et al., editor, *Proc. Third Int'l Conf. on Deductive and Object-Oriented Databases*, pp. 147–166, 1991.

[20] C. de Maindreville and E. Simon. A Production Rule based approach to Deductive databases. In *Proc. Fourth Int'l Conf. on Data Engineering*, 1988.

[21] M. A. Derr, G. Phipps, and K. A. Ross. Glue-Nail: A Deductive Database System. In J. Clifford and R. King, editors, *Proc. Int'l Conf. ACM on Management of Data*, pp. 308–317, 1991.

[22] M. Van Emden and R. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, October 1976.

[23] J. Jaffar et al. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[24] J. Woodfill et al. INGRES version 7. Technical Report April, 8, Reference Manual, 1981.

[25] R. Fagin, J. D. Ullman, and M. Vardi. On the Semantics of Updates in Database. In *Proc. of the ACM Symposium on Principles of Database Systems*, pp. 352–365. ACM, New York, USA, 1983.

[26] S. J. Finkelstein and J. Widom. A Syntax and Semantics for Set-Oriented Production Rules in Relational Database Systems. *SIGMOD Record*, 18(3):36–45, September 1989.

[27] B. Freitag. A Deductive Database Language Supporting Modules. In *Proc. Int'l Computer Science Conference*, pp. 210– 216, Hong Kong, 1992.

[28] M. Gabbrielli, N. Dore, and G. Levi. Observable semantics for Constraints Logic Programs. To appear in *Journal of Logic and Computation*, 1994.

[29] M. Gabbrielli, R. Giacobazzi, and D. Montesi. Modular logic programs on finite domain and dataflow analysis. *Technical Report LIX/RR/94/04*, Ecole Polytechnique, Laboratoire d'Informatique, Paris, 1994.

[30] H. Gaifman and E. Shapiro. Fully Abstract Compositional Semantics for Logic Programs. In *Proc. Sixteenth Annual ACM Symp. on Principles of Programming Languages*, pp. 134–142. ACM, 1989.

[31] H. Gallaire, J. Minker, and J. M. Nicolas. Logic and database: A deductive approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.

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

[33] Y. Gurevich. Towards logic taylored for comptutational complexity, Computation and Proof Theory. ed. M. M. Richter et al. Springer Verlag, *Lecture Notes in Math.* 1104, pp. 175–216, 1984.

[34] D. Harel. *First-Order Dynamic Logic*, Lecture Notes in Computer Science 68, Springer-Verlag, Berlin, 1979.

[35] R. Hull and D. Jacobs. Language Constructs for Programming Active Databases. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pp. 455–467, 1991.

[36] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pp. 111–119. ACM, New York, USA, 1987.

[37] D. Karabeg and V. Vianu. Parallel Update Transactions. In M. Gyssens, J.Paredaens, and D. Van Gucht, editors, *Proc. Second Int'l Conf. on Database Theory*, volume 326 of *Lecture Notes in Computer Science*, pp. 307–321, Springer-Verlag, Berlin, 1988.

[38] E. Laenens, D. Saccà, and D. Vermeir. Extending Logic Programming. In H. Garcia-Molina and H.V. Jagadish, editors, *Proc. Int'l Conf. ACM on Management of Data*, pp. 184–193, 1990.

[39] T. G. Lindholm and R. A. O'Keefe. Efficient Implementation of a defensible Semantics for Dynamic Prolog Code. In J.-L. Lassez, editor, *Proc. Fourth Int'l Conf. on Logic Programming*, pp. 21–39, The MIT Press, Cambridge, Mass., 1987.

[40] J.W. Lloyd. *Foundations of logic programming.* Springer-Verlag, Berlin, 1987. Second edition.

[41] S. Manchanda and D. S. Warren. A Logic-based Language for Database Updates. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pp. 363–394. Morgan-Kaufmann, 1987.

[42] D. Montesi and E. Bertino. Queries, constraints, updates and transactions within a logic- based language. In Y. Yesha B. Bhargava, T. Finin, editor, *Second International Conference of Information and Knowledge Management*, pp. 500–506, 1993.

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

[44] R. A. O'Keefe. Towards an Algebra for Constructing Logic Programs. In *Proc. IEEE Symp. on Logic Programming*, pp. 152– 160, 1985.

[45] Y. Sagiv. Optimizing Datalog Program. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pp. 659–698. Morgan-Kaufmann, 1987.

[46] O. Shmueli. Decidability and Expressiveness Aspects of Logic Queries. In *Proc. of the ACM Symposium on Principles of Database Systems*, pp. 237–249. ACM, New York, USA, 1986.

[47] G. L. Steele. *The implementation and definition of computer programming language based on constraints*. PhD thesis, MIT, Department of Eletrical Engineering and computer Science, August 1980.

[48] J. D. Ullman. *Database and Knowledge-Base Systems*. Computer Science Press, 1989.

[49] M. Winslett. *Updating logical Databases*. Cambridge University Press, 1990.

# A Constraint Logic Programming

The idea of programming with constraints is not new. The use of constraints in Artificial Intelligence had been proposed early in [47] and a number of languages based on it have been designed and implemented. Many of those languages deal with arithmetic constraints. The central concept is that constraints are used not only to represent relationships between objects, but also to compute values based on these relationships. Such use of constraints gave rise to languages which have been called "declarative" because constraints, unlike assignments, are nondirectional.

Intuitively a program in a constraint programming language consists of a set of relations between a set of objects. For example to compute the Fahrenheit (F) equivalent of a Celsius (C) temperature (and vice versa) one might write the statement (which is also a program)

$$\mathtt{F} = 1.8 * \mathtt{C} + 32$$

Given either F or C, the other can be computed, so the same program can be used to solve two different problems, without any explicit decision point. In the above example, F and C are the objects, which in this case are numbers, and the equation is the relationship between these two objects. While it is possible to augment any language with constraint facilities, an important issue is how the underlying language interacts with those constraint facilities. In some constraint languages these interactions require, from the user, a great deal of information about how constraints are to be collected and solved. The most important aspect of constraint logic programming (CLP(X)) [36] language, is that it defines a clean interaction between the underlying logic programming framework and the way constraints are used. CLP(X) defines a class of languages based upon the paradigm of logic languages. Each instance of the schema is a programming language and is obtained by specifying a structure of computation X. That is, the domain of discourse, the functions and relations on this domain characterize the language. A key aspect of this class is that all languages are soundly based within a single framework of formal semantics. This framework extends, in a natural way, the logic programming framework for the following reasons.

- *There is no restriction to the Herbrand universe.* For example, the non-Herbrand domain considered in CLP($\mathcal{R}$) is defined on an arithmetic domain $\mathcal{R}$ of real numbers [23].
- *There is no restriction on the unification.* The notion of *unification* is but a special case of constraint solvability: the notion of obtaining (maximally general) *unifiers* is no more than an implementation notion.
- *There is no restriction on equations.* Equations are but one particular kind of constraints.

The original motivation to integrate logic programming and constraint programming can be described through an example. In Prolog the equality:

$$1 + \mathtt{X} = 3$$

results in a failure, since the operation + is considered as an unevaluated function symbol and the unification algorithms fails. In the past, there have been two unsatisfactory ways to approach the problem. The first is to use Peano's axioms (that is, to define predicates using the successor function). The predicate

$$\texttt{plus}(\texttt{X}, \texttt{Y}, \texttt{Z})$$

means that `Z` is the sum of `X` plus `Y`, and the axioms for addition are:

$$\texttt{plus}(\texttt{0}, \texttt{Y}, \texttt{Y}).$$

$$\texttt{plus}(\texttt{s}(\texttt{X}), \texttt{Y}, \texttt{s}(\texttt{Z})) \leftarrow \texttt{plus}(\texttt{X}, \texttt{Y}, \texttt{Z}).$$

The equation $\texttt{1} + \texttt{X} = \texttt{3}$ is expressed by the query:

$$? \ \texttt{plus}(\texttt{s}(\texttt{0}), \texttt{X}, \texttt{s}(\texttt{s}(\texttt{s}(\texttt{0})))).$$

resulting in $\texttt{X} = \texttt{s}(\texttt{s}(\texttt{0}))$. Programming with Peano's axioms is inefficient, anachronistic and incompatible with high level languages. The second way to bypass the problem of expressing the equality of arithmetic expression is to write special predicates using test and assignments which attempt to determine the values of the variables. In the example this results in:

$$\texttt{X is 2}$$

where the predicate `is` represents an assignment, always requiring a variable as its left operand. Obviously, the assignment expressed by the predicate `is` has not a declarative semantics and therefore cannot be captured with the standard semantics defined in [22]. A general and clean solution to this problem is to replace unification of terms by constraints such as the equality of arithmetic expressions. For example, the constraint:

$$\texttt{2} + \texttt{X} = \texttt{Z} + \texttt{3}$$

will be handled as in algebra. The actual values for `X` or `Z` may be computed later when more constraints are added. When the system of constraints is unsatisfiable, then it results in a failure, just as in logic programming.

## A.1 The CLP language

In order to provide a uniform view of CLP we will deviate slightly from the standard definitions [36]. We assume a fixed $(\Sigma, \Pi, V)$- language over predicates, constant and variables. In addition the CLP framework:

- considers functions in the signature $\Sigma$;

- splits the predicates in two sets, $\Pi_C$ and $Pi_B$ such that $\Pi = \Pi_C \cup \Pi_B$ and $\Pi_C \cap \Pi_B = \emptyset$. Predicates in $\Pi_C$ are used for the constraint language and predicates in $\Pi_B$ for the logic language;

- uses a many-sorted first order language, where $SORT = \bigcup_i SORT_i$ denotes a finite set of sorts.

A *signature* of an $n$-ary function, predicate or variable symbol $f$, is a sequence of respectively $n+1$, $n$, 1 elements of $SORT$. By *sort of* $f$ we mean the last element in the signature of the function symbol $f$. By $Var(E)$ ($Pred(E)$) we denote the sets of variables (predicates) occurring in the expression $E$. The Herbrand universe $\mathcal{H}$ (the set of ground terms) is defined for a given language. $\tilde{B}$ denotes a (possibly empty) conjunction of atoms. Atoms are built over a subset of predicates ($\Pi$), functions and constant ($\Sigma$), and variables ($V$). Often we use a subset of the above triples to denote which kind of atoms (i.e. ground, non ground, constraints) we are dealing with by means of the prefix $(\Pi, \Sigma)$, $(\Pi, \Sigma, V)$ and $(\Pi_C, \Sigma, V)$ respectively. The empty set of constraints is denoted by $true$.

DEFINITION A.1. *(CLP program) A $(\Pi, \Sigma, V)$-program is a set of rules of the form*

$$H \leftarrow c_1, \ldots, c_k, B_1, \ldots, B_n.$$

*where $c_1, \ldots, c_k$ are $(\Pi_C, \Sigma, V)$-atoms, $c = c_1, \ldots, c_k$ (the constraint) is a finite $(\Pi_C, \Sigma, V)$-constraint, $H$ (the head) and $B_1, \ldots, B_n$ are $(\Pi_B, \Sigma, V)$-atoms with distinct variables as arguments. The right part of the rule is called body and can be empty.*

DEFINITION A.2. *(Goal) A goal is a program rule with no head of the form*

$$c_1, \ldots, c_k, B_1, \ldots, B_n.$$

*where $c_1, \ldots, c_k$ and $B_1, \ldots, B_n$ are as in Definition A.1.*

Following the tradition in the examples we always prefix a query with the symbol "?". ? can be seen as the prompt of the corresponding system.

DEFINITION A.3. *(Constrained atom) A $(\Pi_B, \Sigma, V)$- constrained atom is an object of the form $p(\tilde{X}) \leftarrow c$, where $c$ is a $(\Pi_C, \Sigma, V)$-constraint, $p(\tilde{X})$ is a $(\Pi_B, \Sigma, V)$-atom and $\tilde{X}$ are distinct variables.*

The notion of *structure* gives the semantic interpretation of such a domain and is the key element of this framework.

DEFINITION A.4. *A* structure $\Re(\Pi_C, \Sigma)$ *is defined over the (sorted) alphabets $\Pi_C$ and $\Sigma$ of predicate and function symbols, where $\Pi_C$ contains the equality symbol (which needs no signature). $\Re(\Pi_C, \Sigma)$ consists of*

1. *a collection $D\Re$ of non-empty sets $D\Re_s$, where $s$ ranges over the sorts in SORT.*

2. *an assignment to each $n$-ary function symbol $f \in \Sigma$ of a function $D\Re_{s_1} X \ldots X D\Re_{s_n} \rightarrow D\Re_s$ where $(s_1, s_2, \ldots, s_n, s)$ is the signature of f,*

3. *an assignment to each $n$-ary predicate symbol $p \in \Pi_C$, apart from $=$ (which is interpreted as syntactic equality) of a function $D\Re_{s_1} X \ldots X D\Re_{s_n} \rightarrow \{True, False\}$, where $(s_1, s_2, \ldots, s_n)$ is the signature of $p$.*

*An $\Re(\Pi_C, \Sigma)$-valuation for a $(\Pi_C, \Sigma, V)$- expression is a mapping $\theta : V \rightarrow D\Re$, where $V = \bigcup_{s \in SORT} V_s$ is the set of all variables, and $\theta(X_s) \in D\Re_s$, where $s$ is the sort of the variable $X_s$.*

$(\Pi, \Sigma, V)$-programs, $(\Pi_C, \Sigma, V)$-constraints and $(\Pi_B, \Sigma, V)$- constrained atoms will be called programs, constraints and constrained atoms. Moreover, $\Re(\Pi_C, \Sigma)$ will be denoted by $\Re$. The notion of $\Re$-valuation is extended in the obvious way to terms and constraints. If $C$ is a possibly infinite set of atomic constraints, $\Re \models C\theta$ iff $\forall c \in C \ \Re \models c\theta$ ($c\theta$ is $\Re$-equivalent to $True$) holds. Given an expression $E$ and an $\Re$-valuation $\theta$, $E\theta$ denotes the result of the usual application operation.

DEFINITION A.5. *(Solvability) A constraint $c$ is $\Re$-solvable iff there exists an $\Re$-valuation $\theta$ such that $\Re \models c\theta$. $\theta$ is called an $\Re$-solution of $c$. A constrained atom $c, p(\tilde{X})$ is $\Re$- solvable iff $c$ is $\Re$-solvable.*

In the following we consider an example of CLP($\mathcal{H}, EDB$) the instance of CLP(X) that allow to express non-immediate updates. Note that we consider a two sorted structure where the first sort is the equalities between terms and the second is the extensional database. In U-Datalog we avoided the equalities to simplify the syntax. U-Datalog language has also other restrictions with respect to $CLP(\mathcal{H}, EDB)$:

- only constant symbols are allowed;
- an U-Datalog program has the form $EDB \cup IDB$. Thus there is a partition of $\Pi_B$ into $\Pi_{IDB}$ and $\Pi_{EDB}$, that is intensional and extensional predicate symbols. Similar partition for $\Pi_C$ with $=$ for equalities interpreted in the usual way and $\pm p$ where $p \in \Pi_{EDB}$ for the updates.
- constraints are used to express update specifications to base relations.

EXAMPLE A.1. *The database below is the same of Example 3.2 written in U-Datalog. The only syntactic difference is the explicit use of equalities for binding variables with constants. The language we consider is:*

- $\Pi_B = \{$`balance`, `potentialsaver`, `changebalance`, `withdraw`, `deposit`, `transfer`, $<, >\}$,
- $\Pi_C = \{=, -$`balance`, $+$`balance`$\}$,

- $\Sigma = \{102, 105, 1500, 28000, 10000\}$.

*The full syntax for CLP($\mathcal{H}, EDB$) is*

$$
\begin{aligned}
DB' = \ & \texttt{balance(Acnt, Amt)} \leftarrow \texttt{Acnt} = 102, \texttt{Amt} = 1500. \\
& \texttt{balance(Acnt, Amt)} \leftarrow \texttt{Acnt} = 105, \texttt{Amt} = 28000. \\
& \texttt{potentialsaver(Acnt, B)} \leftarrow \texttt{true}, \texttt{balance(Acnt, B)}, \\
& \qquad\qquad\qquad\qquad \texttt{B} > 10000. \\
& \texttt{changebalance(Acnt, B, B}') \leftarrow -\texttt{balance(Acnt, B)}, \\
& \qquad\qquad\qquad\qquad +\texttt{balance(Acnt, B}'). \\
& \texttt{withdraw(Amt, Acnt)} \leftarrow \texttt{true}, \texttt{changebalance(Acnt, B, B} - \texttt{Amt}), \\
& \qquad\qquad\qquad\qquad \texttt{balance(Acnt, B)}, \texttt{Atm} < \texttt{B}. \\
& \texttt{deposit(Amt, Acnt)} \leftarrow \texttt{true}, \texttt{changebalance(Acnt, B, B} + \texttt{Amt}), \\
& \qquad\qquad\qquad\qquad \texttt{balance(Acnt, B)}. \\
& \texttt{transfer(Amt, Acnt, Acnt}') \leftarrow \texttt{true}, \texttt{withdraw(Amt, Acnt)}, \\
& \qquad\qquad\qquad\qquad \texttt{deposit(Amt, Acnt}').
\end{aligned}
$$

*The absence of a constraint is denoted by the true value.* ◇

The above syntax is remarkably more difficult to understand of the equivalent U-Datalog syntax. This is the main reason to use in the example the more appropriate U-Datalog language. However, the formal results are carried out in the general CLP($X$) context.

## A.2 Semantics domain

In the following we recall the operational, and fixpoint semantics and their equivalence. The equivalent operational and fixpoint semantics allow to use a top-down or bottom-up computational model to perform the query-answering process.

We introduce the notion of interpretation for CLP programs which is an extension of the classic notion of interpretation in logic programming [40]. An interpretation in logic programming is any subset of the Herbrand base. The Herbrand base is a set of all the ground atoms for a program $P$. The notion of constrained atom introduced in Definition A.3 extends to CLP the classic notion of atom in an interpretation for logic programs. Since the structure $\Re$ is the intended domain of computation for CLP, interpretations for CLP programs are based on $\Re$. All the following definitions are related to a given $\Re$. Let us first define the set of "domain instances" on constrained atoms by means of the operator "[ ]".

DEFINITION A.6. *Let $p(\tilde{X}) \leftarrow c$ be a constrained atom, thus*

$$[p(\tilde{X}) \leftarrow c] = \{(p(\tilde{X}) \leftarrow c)\theta \mid \theta \text{ is an } \Re\text{-solution of } c\}.$$

*This definition can be extended to a set $S$ of constrained atoms in the obvious way by defining $[S] = \bigcup_{A \in S}[A]$.*

Let us now give the new definition of interpretation for CLP programs. We first introduce the equivalence $\simeq$ used in the semantic domain in order to abstract from syntactical differences among constrained atoms.

DEFINITION A.7. *Let $a_1 = p(\tilde{X}) \leftarrow c_1$ and $a_2 = p(\tilde{X}) \leftarrow c_2$ be two constrained atoms. Then*

$$a_1 \simeq a_2 \text{ iff } [a_1] = [a_2].$$

*However, we denote with $a_1$ the constrained atom and its equivalence class $[a_1]_{/\simeq}$.*

Note that $a_1 \simeq a_2$ is equivalent to say that $c_1$ and $c_2$ have the same solutions for the variables $\tilde{X}$.

DEFINITION A.8. *(Base) Let $P$ be a program and let $\mathcal{A}$ be the set of all the $\Re(\Pi_C, \Sigma)$-solvable constrained atoms for $P$. The base of interpretations $\mathcal{B} = \mathcal{A}_{/\simeq}$.*

DEFINITION A.9. *(Interpretation) An interpretation is any subset of $\mathcal{B}$. The set of all interpretations is denoted by $\mathcal{I}$.*

DEFINITION A.10. *(Truth) Let $I$ be an interpretation. A constrained atom $p(\tilde{X}) \leftarrow c$ is true in $I$ iff $[p(\tilde{X}) \leftarrow c] \subseteq [I]$. A rule $H \leftarrow c, B_1, \ldots, B_n.$ is true in $I$ iff for each $\Re$-valuation $\theta$ such that $\theta$ is an $\Re$-solution of $c$ and $\{(H_1 \leftarrow c_1)\theta, \ldots, (H_n \leftarrow c_n)\theta\} \subseteq [I]$ implies $(H \leftarrow c, c_1, \ldots, c_n, H_1 = B_1, \ldots, H_n = B_n)\theta \in [I]$.*

A parallel with the logic programming case can clarify the discussion. The logic programs standard semantics defines models as sets of ground atoms. The base of Definition A.8 contains non-ground constrained atoms in the models, where a non-ground constrained atom stands for an implicit definition of the set of all its ground instances. Each constrained atom $p(\tilde{X}) \leftarrow c$ describing the set of elements $(p(\tilde{X}) \leftarrow c)\theta$ where $\theta$ is any $\Re$-solution for the constraint $c$. In order to abstract from the particular syntactic representation of atoms, in the logic programming case the base is defined modulo variance.

## A.3 The operational semantics

The operational semantics of constraint logic programs can be specified by means of a set of inference rules which specify how derivations are performed. The CLP inference rule takes into account that the derivation is performed over a generic structure $\Re$ rather than the Herbrand domain. Therefore, unification is replaced by checking the $\Re$-solvability of constraints. The following definitions use a *parallel selection rule* where all the atoms of the resolvent are evaluated in a single derivation step. This is exactly what the inference rule of a deductive database does. As far as successful derivations are considered, a parallel selection rule is equivalent to the more usual one which selects a single atom according to [13].

DEFINITION A.11. *($\Re$-derivation step) Let $P$ be a program. An $\Re$-derivation step of a goal $G = c, G_1, \ldots, G_t$ in $P$ results in a goal of the form $c', \tilde{B}_1, \ldots, \tilde{B}_t$, and is denoted by $c, G_1, \ldots, G_t \longrightarrow_P c', \tilde{B}_1, \ldots, \tilde{B}_t$ if there exist $t$ variants of clauses in $P, H_j \leftarrow c_j, \tilde{B}_j$, $j = 1, \ldots, t$ with no variables in common with $G$ and with each other, such that $c'$ is $\Re$- solvable with $c' = c \cup c_1 \cup \ldots \cup c_t \cup \{H_1 = G_1\}, \ldots, \{H_t = G_t\}$.*

DEFINITION A.12. *($\Re$-derivation) Let $P$ be a program. An $\Re$-derivation of a goal $G = c, G_1, \ldots, G_n$ in $P$ is a maximal (finite or infinite) sequence of goals starting from $G$ such that goal every goal apart from $G$ is obtained from the previous one by means of an $\Re$-derivation step.*

DEFINITION A.13. *(successful $\Re$-derivation) Let $P$ be a program. A successful $\Re$-derivation of a goal $G = c, G_1, \ldots, G_n$ is a finite $\Re$-derivation whose last element is a goal of the form $c'$. $c'$ is the answer constraint of the derivation. All other finite $\Re$-derivations are finitely failed. The successful derivation of a goal $G$ which yields the answer constraint $c'$, is denoted by $G \longrightarrow_P^* c'$.*

For the sake of simplicity we omit the program $P$ in $\longrightarrow_P$ and $\longrightarrow_P^*$ whenever the considered program is clear. We are now ready to define the operational semantics of CLP.

DEFINITION A.14. *Let $P$ be a program. The operational semantics $\mathcal{O}(P)$ is defined as follows*

$$\mathcal{O}(P) = \{\ p(\tilde{X}) \leftarrow c \in \mathcal{B} |\ true, p(\tilde{X}) \longrightarrow^* c\ \}.$$

We recall that *true* denotes the empty constraint. The denotation of a program is a set of non-ground constrained atoms. More precisely, a denotation is a (possibly infinite) set of equivalence classes of constrained atoms. The equivalence is needed to abstract from irrelevant syntactic differences and in the above semantics it is simply the variance relation.

EXAMPLE A.2. *Consider the database of Example 3.2 or the equivalent A.1. Then, its operational semantics is*

$$
\begin{aligned}
\mathcal{O}(DB) = \{\ & balance(102, 1500), \\
& balance(102, 28000), \\
& potentialsaver(105, 28000), \\
& changebalance(Acnt, B, B') \leftarrow -balance(Acnt, B), \\
& \qquad\qquad +balance(Acnt, B'), \\
& withdraw(Amt, 102) \leftarrow -balance(102, 1500), \\
& \qquad\qquad +balance(102, 1500 - Amt), \\
& \qquad\qquad Amt < 1500, \\
& withdraw(Amt, 105) \leftarrow -balance(105, 28000), \\
& \qquad\qquad +balance(105, 28000 - Amt), \\
& \qquad\qquad Amt < 28000, \\
& deposit(Amt, 102) \leftarrow -balance(102, 1500), \\
& \qquad\qquad +balance(102, 1500 + Amt), \\
& deposit(Amt, 105) \leftarrow -balance(105, 28000), \\
& \qquad\qquad +balance(105, 28000 + Amt), \\
& transfer(Amt, 102, 105) \leftarrow -balance(102, 1500), \\
& \qquad\qquad +balance(102, 1500 - Amt), \\
& \qquad\qquad -balance(105, 28000), \\
& \qquad\qquad +balance(105, 28000 + Amt), \\
& \qquad\qquad Amt < 1500, \\
& transfer(Amt, 105, 102) \leftarrow -balance(105, 28000), \\
& \qquad\qquad +balance(105, 28000 - Amt), \\
& \qquad\qquad -balance(102, 1500), \\
& \qquad\qquad +balance(102, 1500 + Amt), \\
& \qquad\qquad Amt < 28000 \qquad\qquad \}
\end{aligned}
$$

$\diamond$

In the following we give some results concerning the operational, fixpoint semantics and their equivalence. Further details are in [28]. Theorem A.1 states a soundness and completeness results for $\mathcal{O}(P)$. As a consequence we obtain corollary A.1 which shows that this semantics fully characterizes the operational behavior modeling the answer constraints of programs.

THEOREM A.1. *Let $P$ be a program and $G = (c_0, A_1, \ldots, A_n)$ be a goal. Then $G \longrightarrow^* c$ iff there exist $n$ constrained atoms $B_i \leftarrow c_i \in \mathcal{O}(P)$, $i = 1, \ldots n$, which share no variables with $G$ and with each other, such that $c_0 \cup c_1 \cup \ldots \cup c_n \cup \{A_1 = B_1\} \cup \ldots \cup \{A_n = B_n\}$ and $c$ have the same solutions for the variables in $G$.*

The following corollary shows the relationship between programs and semantics equivalence.

COROLLARY A.1. *Let $P_1, P_2$ be programs. Then $P_1 \approx P_2$ iff $\mathcal{O}(P_1) = \mathcal{O}(P_2)$.*

## A.4 The fixpoint semantics

Let us now introduce an immediate consequence operator, which allows to characterize the CLP programs.

DEFINITION A.15. *Let $P$ be a program and let $J \subseteq \mathcal{B}$.*

$$
\begin{array}{ll}
T_P(J) = \{ & p(\tilde{X}) \leftarrow c \in \mathcal{B} \mid \\
& \exists \text{ a renamed clause} \\
& p(\tilde{t}) \leftarrow c_0, p_1(\tilde{t_1}), \ldots, p_n(\tilde{t_n}) \text{ in } P \\
& \exists\ p_i(\tilde{X_i}) \leftarrow c_i \in J, 1 \leq i \leq n \\
& \text{which share no variables} \\
& c = c_0 \cup \{\tilde{X_1} = \tilde{t_1}\} \cup \ldots \cup \{\tilde{X_n} = \tilde{t_n}\} \cup \\
& \{c_1, \ldots, c_n\} \cup \{\tilde{X} = \tilde{t}\} \\
& c \text{ is } \Re\text{-solvable} \quad \}.
\end{array}
$$

DEFINITION A.16. *Let $T$ be a monotonic operator on the lattice $(\mathcal{I}, \subseteq)$. Then we define*
$T \uparrow 0 = \emptyset$
$T \uparrow \alpha + 1 = T(T \uparrow \alpha)$ *for any ordinal* $\alpha$
$T \uparrow \gamma = \bigcup_{\alpha < \gamma} T \uparrow \alpha$ *for* $\gamma$ *limit ordinal*

The following lemma allows to define a fixpoint semantics using the $T_P$ operator.

LEMMA A.1. *The mapping $T_P$ is continuous on the cpo $(\mathcal{I}, \subseteq)$. There exists a least fixpoint $lfp(T_P)$ of $T_P$ which is $T_P \uparrow \omega$.*

DEFINITION A.17. *Let $P$ be a program. The fixpoint semantics $Fix(P)$ of $P$ is defined as $Fix(P) = T_P \uparrow \omega$.*

Theorem A.2 shows that we have the expected result on the relation between the operational and the fixpoint semantics.

THEOREM A.2. *(Equivalence of the operational and the fixpoint semantics) Let $P$ be a program. Then $\mathcal{O}(P) = Fix(P)$*[8].

EXAMPLE A.3. *Consider the database of Examples A.2. The fixpoint semantics is:*

[8] The operational and fixpoint semantics were denoted as $\mathcal{S}_2^o$ and $\mathcal{S}_2^f$ in Section 4.

$$
\begin{aligned}
T_0(DB) = \{\ & balance(102, 1500), \\
& balance(102, 28000), \\
& changebalance(Acnt, B, B') \leftarrow -balance(Acnt, B), \\
& \qquad\qquad +balance(Acnt, B') \quad \} \\
T_1(DB) = \{\ & potentialsaver(105, 28000), \\
& withdraw(Amt, 102) \leftarrow -balance(102, 1500), \\
& \qquad\qquad +balance(102, 1500 - Amt), \\
& \qquad\qquad Amt < 1500, \\
& withdraw(Amt, 105) \leftarrow -balance(105, 28000), \\
& \qquad\qquad +balance(105, 28000 - Amt), \\
& \qquad\qquad Amt < 28000, \\
& deposit(Amt, 102) \leftarrow -balance(102, 1500), \\
& \qquad\qquad +balance(105, 28000 + Amt), \\
& deposit(Amt, 105) \leftarrow -balance(105, 28000), \\
& \qquad\qquad +balance(Acnt, B + Amt) \quad \} \\
& \cup T_0(DB) \\
T_2(DB) = \{\ & transfer(Amt, 102, 105) \leftarrow -balance(102, 1500), \\
& \qquad\qquad +balance(102, 1500 - Amt), \\
& \qquad\qquad -balance(105, 28000), \\
& \qquad\qquad +balance(105, 28000 + Amt), \\
& \qquad\qquad Amt < 1500, \\
& transfer(Amt, 105, 102) \leftarrow -balance(105, 28000), \\
& \qquad\qquad +balance(105, 28000 - Amt), \\
& \qquad\qquad -balance(102, 1500), \\
& \qquad\qquad +balance(102, 1500 + Amt), \\
& \qquad\qquad Amt < 28000 \quad \} \cup T_1(DB) \\
T_3(DB) = \quad & T_i(DB)(\forall i > 2) = T_2
\end{aligned}
$$

$\diamond$

We conclude this appendix with some observations. CLP was originally introduced to formalize the integration between constraint programming and logic programming. Our motivations to use CLP are quite different. The use of CLP is in formalizing the result of marking phase through that of answer constraints. Such answer can be computed top-down or bottom-up. Therefore, we have now the general framework to solve the problem of modelling the marking phase. However, we have not yet solved the problem of compositionality among extensional and intensional databases. This is the major achievement of the next appendix.

# B Compositional CLP programs

Compositionality is related with a (syntactic) program composition operator *op*, and holds when the semantics of the compound program $P_1$ *op* $P_2$ is defined by (semantically) composing the semantics of the constituents $\mathcal{S}(P_1)$ and $\mathcal{S}(P_2)$. In the case of logic programs, the construct which raises a compositionality problem is

the *union* of rules. The related property is sometimes called *OR-compositionality*. OR-compositional semantics have been investigated for logic programs both for theoretical and for practical purposes in [27, 30, 44]. Unfortunately, the semantics for constraint logic programming is not compositional with respect to program union (see Example B.1). In this appendix we solve such problem. The semantics introduced in the previous Appendix considers a program as a set of facts and rules. Indeed this is the simplest case. Generally speaking a program could be a set of units. Each unit can be a program itself. In this way it is possible to model modular knowledge bases, where each module can cooperate with the others. This is the case of U-Datalog programs, where we are interested to compose intensional and extensional databases.

EXAMPLE B.1. *Let us consider the following database where we can compute the ancestor and remove all the parents of a given ancestor.*

$$
\begin{aligned}
EDB = \;& \texttt{parent(henry, peter).} \\
& \texttt{parent(bob, peter).} \\
& \texttt{parent(peter, john).}
\end{aligned}
$$

$$
\begin{aligned}
IDB = \;& \texttt{anc(X, Y)} \leftarrow \texttt{parent(X, Y).} \\
& \texttt{anc(X, Z)} \leftarrow \texttt{anc(X, Y), parent(Y, Z).} \\
& \texttt{rmanc(X, Z)} \leftarrow -\texttt{parent(X, Y), rmanc(X, Y), parent(Y, Z).} \\
& \texttt{rmanc(X, Y)} \leftarrow -\texttt{parent(X, Y), parent(X, Y).}
\end{aligned}
$$

*According to Definition A.14 the semantics,* $\mathcal{O}(IDB) = \emptyset$ *and*

$$
\begin{aligned}
\mathcal{O}(EDB) = \{ \;& parent(henry, peter), \\
& parent(bob, peter), \\
& parent(peter, john) \;\; \}.
\end{aligned}
$$

*Since*

$$
\begin{aligned}
\mathcal{O}(EDB \cup IDB) = \{ \;& parent(henry, peter), \\
& parent(bob, peter), \\
& parent(peter, john), \\
& anc(henry, peter), \\
& anc(bob, peter), \\
& anc(peter, john), \\
& anc(bob, iohn), \\
& anc(henry, john), \\
& rmanc(henry, peter) \leftarrow -parent(henry, peter), \\
& rmanc(bob, peter) \leftarrow -parent(bob, peter), \\
& rmanc(peter, john) \leftarrow -parent(peter, john) \\
& rmanc(bob, iohn) \leftarrow -parent(bob, peter), \\
& \qquad\qquad -parent(peter, john), \\
& rmanc(henry, john) \leftarrow -parent(henry, peter), \\
& \qquad\qquad -parent(peter, john) \;\; \}.
\end{aligned}
$$

*the semantics of the union of the two programs cannot be obtained from the semantics of the programs.*

◇

In the following we define a compositional semantics for CLP in the style introduced by Bossi et al. ([15]). Such semantics extends that for constraint logic programs that was introduced in the previous Appendix. The program composition operator *op* we will consider is $\cup$. In order to model a composition of programs, we introduce the notion of open program, the semantics domains, the operational semantics. Finally, the relationship with the classic semantics of CLP is provided.

## B.1 Open programs

We extend the approach to compositional semantics presented by Bossi et Al. for pure logic programs to constraint logic programs. Informally, the notion of program composition we consider is $\cup_\Omega$, which is a generalization of program union where the set of predicates $\Omega \subseteq \Pi_B$ specifies which predicates can be shared by different programs. If $\Omega = \Pi_B$, $\cup_\Omega$ is the standard union, while if $\Omega = \emptyset$ the composition is allowed only among programs which do not share predicate symbols. Let us formally give the definition of the program composition we consider.

DEFINITION B.1. *($\Omega$-open CLP program) An $\Omega$-open CLP program ($\Omega$-program for short) is a constraint logic program $P$ together with a set $\Omega$ of predicate symbols such that $\Omega \subseteq \Pi_B$. A predicate symbol occurring in $\Omega$ is considered to be only partially defined in $P$.*

We recall that a CLP program is defined over $(\Pi, \Sigma, V)$, therefore the fact that $\Omega \subseteq \Pi_B$ is obvious.

DEFINITION B.2. *($\Omega$-union) Let $P_1$ be an $\Omega_1$-program and $P_2$ be an $\Omega_2$-program. If*

1. $\Omega \subseteq \Omega_1 \cup \Omega_2$ *and*
2. $(Pred(P_1) \cap Pred(P_2)) \subseteq (\Omega_1 \cap \Omega_2)$

*then $P_1 \cup_\Omega P_2$ is the $\Omega$-open program $P_1 \cup P_2$. Otherwise $P_1 \cup_\Omega P_2$ is not defined.*

Note that when considering an $\Omega$-open program $P$ and an $\Omega'$-open program $Q$, the composition of $P$ and $Q$ is defined only if $(Pred(Q) \cap Pred(P)) \subseteq (\Omega \cap \Omega')$. Moreover, the composition of $P$ and $Q$ is a $\Psi$-open program, where $\Psi = \Omega \cup \Omega'$. The definition of any predicate symbol $p \in \Omega$ in an $\Omega$-open program $P$ can always be extended or refined. Therefore, a deduction dealing with a predicate symbol of an $\Omega$-open program $P$ can be either *complete* (when it takes place completely in the program $P$) or *partial* (when it terminates in $P$ with an atom $p(\tilde{t})$ such that $p \in \Omega$ and $p(\tilde{t})$ does not unify with the head of any rule in $P$. A partial deduction can be completed by the addition of new rules. Thus we have an *hypothetic deduction*, which depends on the extension of the predicate $p$.

## B.2 Interpretations for open CLP programs

In this section we formally define the interpretations which characterize the above informal semantics. Since the semantics of open CLP programs contains rules (whose body predicates are all in $\Omega$), we have to accommodate rules in the interpretations we use. Therefore we will define the notion of interpretation for open CLP programs. Such interpretation extends the notion of interpretation for CLP programs given in Definition A.9 since it contains conditional constrained atoms. The conditional constrained atom extends the notion of constrained atom given in Definition A.3. Informally, this means that the interpretation for rules must contain information in form of a mapping from sets of atoms to sets of atoms. This mapping called conditional constraint atoms is nothing else that the usual concept of rule.

DEFINITION B.3. *(Conditional constrained atom) A conditional constrained atom is a rule of the form $H \leftarrow c, B_1, \ldots, B_n$ where $B_1, \ldots, B_n$ is a multiset of $(\Pi_B, \Sigma, V)$-atoms such that $Pred(B_1, \ldots, B_n) \subseteq \Omega$ and $c$ is a set of $(\Pi_C, \Sigma, V)$-constraints. $H$ is a $(\Pi_B, V)$-atom and $Var(H)$ are distinct variables.*

Before giving the definition of interpretation we need the following definitions.

DEFINITION B.4. *Let $H \leftarrow c, \tilde{B}$ be a conditional constrained atom. Then we define*

$$[H \leftarrow c, \tilde{B}] = \{(H \leftarrow c, \tilde{B})\theta \mid \theta \text{ is an } \Re\text{-solution of } c\}.$$

*This definition can be extended to a set $S$ of conditional constrained atoms in the obvious way by defining $[S] = \bigcup_{A \in S}[A]$.*

We first introduce the equivalence $\simeq$ used in the semantic domain in order to abstract from syntactical differences among conditional constrained atoms.

DEFINITION B.5. *Let $d_1 = H_1 \leftarrow c_1, B_1, \ldots, B_n$ and $d_2 = H_2 \leftarrow c_2, D_1, \ldots, D_m$ be conditional constrained atoms. Then*

$$d_1 \simeq d_2 \text{ iff } [d_1] = [d_2]$$

*Moreover, $d_1$ denotes the conditional constrained atom and its equivalence class $[d_1]_{/\simeq}$.*

DEFINITION B.6. *(Base) Let $P$ be a program and let $\mathcal{A}$ be the set of all the $\Re(\Pi_C, \Sigma)$-solvable conditional constrained atoms for $P$. The base of interpretations $\mathcal{C}^\Omega = \mathcal{A}_{/\simeq}$.*

DEFINITION B.7. *(Interpretation) An interpretation is any subset of $\mathcal{C}^\Omega$. The set of all the interpretations is denoted by $\mathcal{I}$.*

Note that any subset of $\mathcal{C}^\Omega$ will be considered implicitly as an $\Omega$-open program. In the following the semantics for open CLP programs will be formally considered.

## B.3 The semantics of open CLP programs

We want to express the semantics by means of a set of inference rules which specify how derivations are made. In Subsection A.3 we considered inference rules using a parallel selection rule. In the following we consider inference rule using a fair selection rule $R$. Thus we denotes a derivation step with $\longrightarrow_{P,R}$ to make clear that the derivation is performed with the selection rule $R$ in the program $P$. Similarly, for the transitive closure of the derivation relation $\longrightarrow^*_{P,R}$.

DEFINITION B.8. *(linear $\Re$-derivation step) Let $P$ be a program and $R$ a selection rule. A linear $\Re$-derivation step of a goal $G = c, G_1, \ldots, G_t$ in $P$ which uses the selection rule $R$ results in a goal of the form*

$$G' = c', G_1, \ldots, G_{k-1}, \tilde{B}, \ldots, G_t,$$

*and is denoted by $G \longrightarrow_{P,R} G'$ if there exist an atom $G_k$ and a variant of rule in $P, H \leftarrow c'', \tilde{B}$, with no variables in common with $G$ and such that $c' = c \cup c'' \cup \{G_k = H\}$ is $\Re$-solvable.*

We will show the equivalence between the successful $\Re$-derivation using a parallel selection rule and the linear successful $\Re$-derivations using any fair selection rule. The proof of Theorem B.1 makes uses of two lemmas, stating the independence from the selection rule.

LEMMA B.1. *Let $G_0, \ldots, G_n$ be the sequence of goals of a linear $\Re$-derivation, such that*

1. *$G_q = c, A_1, \ldots, A_m$ (where $1 \leq q \leq n-2$);*
2. *$A_i$ is the selected atom in $G_q$ and $C_i = H_i \leftarrow c_i, \tilde{B}_i$ is the corresponding rule;*
3. *$A_j$ is the selected atom in $G_{q+1}$ and $C_j = H_j \leftarrow c_j, \tilde{B}_j$ is the corresponding rule.*

*Then there exists a linear $\Re$-derivation with the sequence of goals $G_0, \ldots, G_q, G', G_{q+2}, \ldots, G_n$, where $A_j$ is selected in $G_q$ and $A_i$ is selected in $G'$.*

**Proof**
Without loss of generality, assume $i < j$. First note that

$$\begin{aligned} G_{q+1} &= c \cup c_i \cup \{A_i = H_i\}, A_1, \ldots, \tilde{B}_i, \ldots, A_m \\ G_{q+2} &= c \cup c_i \cup \{A_i = H_i\} \cup c_j \cup \{A_j = H_j\}, \\ &\quad A_1, \ldots, \tilde{B}_i, \ldots, \tilde{B}_j, \ldots, A_m \end{aligned}$$

If we select $A_j$ before $A_i$ we get:

$$
\begin{aligned}
G' &= c \cup c_j \cup \{A_j = H_j\}, A_1, \ldots, \tilde{B}_j, \ldots, A_m \\
G'' &= c \cup c_j \cup \{A_j = H_j\} \cup c_i \cup \{A_i = H_i\}, \\
&\quad A_1, \ldots, \tilde{B}_i, \ldots, \tilde{B}_j, \ldots, A_m
\end{aligned}
$$

Obviously, $G'' = G_{q+2}$ since the union operator is commutative. ∎

LEMMA B.2. *For every goal $G$, for every selection rules $R$ and $R'$, $G$ has a linear $\Re$-successful derivation via $R$ with answer constraint $c$ iff $G$ has a linear $\Re$-successful derivation via $R'$ with answer constraint $c$.*

**Proof**
The proof is by induction on the length of the derivation.
($l = 1$) Trivial.
($l > 1$) If $R$ and $R'$ agree on the first selection, then the theorem easily follows from the induction hypothesis. Otherwise, let $G, G_1, \ldots, G_n$ be a refutation of $G$ via $R$, and let $A_i$ and $A_j$ be the literals selected in $G$ by $R$ and $R'$, respectively. By repeatedly applying Lemma B.1 to $G, G_1, \ldots, G_n$ we can obtain a linear $\Re$-successful derivation $G, G'_1, \ldots, G'_n$ where $A_j$ is selected at the first step and $G_n = G'_n$. Then the theorem follows by applying the induction hypothesis to $G'_1$.

∎

THEOREM B.1. *Let $P$ be a program and $G = c, A_1, \ldots, A_n$ a goal. $G$ has a successful $\Re$-derivation with answer constraints $c$ iff $G$ has a linear successful $\Re$-derivation with the same answer constraint.*

**Proof**
($\Rightarrow$) We expand each step of the $\Re$-derivation into $n$ steps of a linear $\Re$-derivation where $n$ is the number of atoms in the goal. Let $G_i = c_i, A_1, A_2, \ldots, A_n$ the $i_{th}$ resolvent in the $\Re$-derivation, let the selected rules be

$$
\begin{aligned}
C_1 &= A'_1 \leftarrow \check{c}_1, B_1 \\
C_2 &= A'_2 \leftarrow \check{c}_2, B_2 \\
&\vdots \\
C_n &= A'_n \leftarrow \check{c}_n, B_n
\end{aligned}
$$

and let $G_{i+1}$ be equal to $c_i \cup \check{c}_1 \cup \check{c}_2 \cup \ldots \cup \check{c}_n \cup \{A_1 = A'_1\} \cup \{A_2 = A'_2\} \cup \ldots \cup \{A_n = A'_n\}, \tilde{B}$, where $\tilde{B}$ is the juxtaposition of $B_1$, ..., $B_n$. If we select each $A_i$ in turn and apply rules $C_i$ in $n$ successive steps of a linear $\Re$-derivation, we obtain $G_{i+1}$. Expanding each step of the successful $\Re$-derivation we obtain a linear successful $\Re$- derivation with the same answer constraint. By Lemma B.2 the same answer constraint will be computed using any selection rule.

($\Leftarrow$) If we have a linear $\Re$-refutation for G, by Lemma B.2 we have also a derivation using a "breadth first" selection rule. Let $G_i$ be the $i_{th}$ resolvent in this refutation, with $G_0 = G = c, A_1, A_2, \ldots, A_n$. The breadth first selection rule will select each $A_i$ from left to right. Let

$$C_i = A'_i \leftarrow \tilde{c}_i, B_i$$

the rule used in the $i_{th}$ step. After $n$ steps the resolvent $G_n$ will be $c \cup \tilde{c}_1 \cup \ldots \cup \tilde{c}_n \cup \{A_1 = A'_1\} \cup \ldots \cup \{A_n = A'_n\}, \tilde{B}$, where $\tilde{B}$ is the juxtaposition of $B_1, \ldots, B_n$. This resolvent can be obtained by a single step in a $\Re$-derivation using the same rules. This construction can be iterated to obtain a successful $\Re$-derivation from $G$ with the same answer constraint.

∎

In the following we omit to specify that a derivation is linear.

DEFINITION B.9. *Let $\Omega$ be a set of predicates. We define*

$$Id_\Omega = \{p(\tilde{X}) \leftarrow true, p(\tilde{X}) \mid \tilde{X} \text{ are distinct variables } \}.$$

In the following we denote with $c, G_1, \ldots, G_n \rightsquigarrow_{P,R} c', B_1, \ldots, B_t$ the $\Re$-derivation of the resolvent $c', B_1, \ldots, B_t$ from $c, G_1, \ldots, G_n$ in the program $P$ using the selection rule $R$. If there is no indication of the selection rule $R$, then we assume the parallel selection rule.

DEFINITION B.10. *(Open semantics) Let $P$ be a program, $\Omega$ be a set of predicate symbols, $\tilde{A} = A_1, \ldots, A_m$ and $\tilde{B} = B_1, \ldots, B_n$. Moreover let $P^+ = P \cup Id_\Omega$ and let $R$ be a fair selection rule. Then the $\Omega$-compositional semantics is*

$$\begin{aligned}\mathcal{O}_\Omega(P) = \{ \quad & p(\tilde{X}) \leftarrow c', \tilde{B} \in \mathcal{C}^\Omega \mid \\ & true, p(\tilde{X}) \rightsquigarrow_{P,R} c, \tilde{A} \rightsquigarrow_{P^+,R} c', \tilde{B}, \\ & \text{and } \{Pred(\tilde{B})\} \subseteq \Omega \qquad\qquad \}.\end{aligned}$$

*Moreover if $P$ is an $\Omega$-open program, $\mathcal{O}_\Omega(P)$ is also $\Omega$-open.*

The denotation of a program is a set of conditional constrained atoms, which can be viewed as a possibly infinite program. More precisely, a denotation is a (possibly infinite) set of equivalence classes of conditional constrained atoms. The equivalence is need to abstract from irrelevant syntactic differences and in the above semantics it is simply the variance relation.

Note that $\mathcal{O}_\Omega(P)$ is a set of *resultants* [40] obtained from goals of the form $true, p(\tilde{X})$ in $P$ and is essentially the result of the partial evaluation of $P$, where derivations terminate at open predicates (i.e. predicates in $\Omega$). The set of rules $Id_\Omega$ in the previous definition is used to delay the evaluation of open atoms. As shown by the Proposition B.1, this a trick which allows to obtain, by using a fixed fair

selection rule $R$, all the derivations $true, p(\tilde{X}) \leadsto_{P,R'} c', B_1, \ldots, B_n$ which use any fair selection rule $R'$ for $Pred(B_1, \ldots, B_n) \subseteq \Omega$. Therefore the previous definition is independent from the selection rule considered. Note that in the first step of the derivations we use rules from $P$ (instead than from $P^+$) because we want $\mathcal{O}_\Omega(P)$ to contain a rule $p(\tilde{X}) \leftarrow true, p(\tilde{X})$ if and only if $true, p(\tilde{X}) \leadsto_P true, p(\tilde{X})$.

PROPOSITION B.1. *Let $R$ be a selection rule, let $P^+ = P \cup Id_\Omega$, $\tilde{X}$ a tuple of distinct variables and $\{Pred(B_1, \ldots, B_n)\} \subseteq \Omega$. Then there exists a rule $R'$ such that $true, p(\tilde{X}) \leadsto_{P,R'} c, B_1, \ldots, B_n$ iff $true, p(\tilde{X}) \leadsto_{P,R} c', D_1, \ldots, D_m$ $\leadsto_{P^+,R} c'', B_1, \ldots, B_n$ and $c$, $c''$ have the same solutions for the variables $\tilde{X}$.*

**Proof**
$\Leftarrow$) Straightforward, by considering $R'$ as the selection rule obtained from $R$, by eliminating in the derivation

$$true, p(\tilde{X}) \leadsto_{P,R} c', D_1, \ldots, D_m \leadsto_{P^+,R} c'', B_1, \ldots, B_n$$

all the selections of atoms, which are rewritten by rules in $Id_\Omega \setminus P$.
$\Rightarrow$) By hypothesis there exists a derivation

$$d_1 : \quad true, p(\tilde{X}) \leadsto_{P,R'} c, B_1, \ldots, B_n$$

Due to the fact that $P \subseteq P^+$ we can assume that there exists

$$d_2 : \quad true, p(\tilde{X}) \leadsto_{P^+,R} \bar{c}, B_1, \ldots, B_n$$

The first rule used in $d_2$ is the same rule of program $P$ which is used to rewrite $p(\tilde{X})$ in the first step of the derivation $d_1$.

For each atom $A_j$ that is selected by $R$ in a step of the derivation $d_2$, if $A_j$ is chosen by $R'$ in $d_1$, we use the same input rule used in $d_1$ (recall that $P \subseteq P^+$). Note that if $A_j$ is not selected by $R'$, then $Pred(A_j) \in \Omega$, since $\{Pred(B_1, \ldots, B_n)\} \subseteq \Omega$. Therefore if $A_j$ is not selected by $R'$, we can use, to write $A_j$, the input rule $p_j(\tilde{X}_j) \leftarrow p_j(\tilde{X}_j) \in Id_\Omega$, where $p_j = Pred(A_j)$. Then the derivation $d_2$ only uses rules which are used in the derivation $d_1$ and some rules in $Id_\Omega$. Moreover, since the selection rule $R$ is fair, if an atom $A_i$ is selected in a step of the derivation $d_1$, then in the derivation $d_2$ the atom $A_i$ is selected in a finite number of steps (recall that we can always rewrite an atom $q(\tilde{t})$, where $q \in \Omega$ in the derivation $d_2$ by means the input rule $q(\tilde{Y}) \leftarrow q(\tilde{Y})$. Thus $c = \bar{c} \cup E$ where $E = \{\tilde{t}_1 = \tilde{Y}_1, \ldots, \tilde{t}_s = \tilde{Y}_s\}$ for $i = 1, \ldots, s$ is a set of equations and $\tilde{Y}_i's$ are new distinct variables, which do not occur in $c$. Therefore, $c$ and $\bar{c}$ have the same solutions for the variables $\tilde{X}$.

∎

We define the congruence $\approx_\Omega$ on programs when considering the compositional program operator $\cup$ and the set of predicate symbols $\Omega$. It can formally be defined as follows.

DEFINITION B.11. *Let $P_1, P_2$ be $\Omega$-open programs. Then $P_1 \approx_\Omega P_2$ if for every goal $G$ (with $\tilde{Y} = Var(G)$) and for every $\Omega$-program $Q$ such that $P_i \cup_\Omega Q$, $i = 1, 2$, is defined,*

$$true, G \longrightarrow^*_{P_1 \cup_\Omega Q} c \text{ iff } true, G \longrightarrow^*_{P_2 \cup_\Omega Q} c'$$

*where $c$ and $c'$ have the same solutions for the variables $\tilde{Y}$.*

$\mathcal{O}_\Omega$ allows to characterize a notion of answer constraints which enhances the usual one (see Definition A.14), since also (unresolved) atoms, with predicate symbols in $\Omega$, are considered. Therefore it is able to model answer constraints in an OR-compositional way. As we will see in the following, Theorem B.2 shows that a program $P$ and its operational semantics $\mathcal{O}_\Omega(P)$ are $\approx_\Omega$ equivalent. As a consequence, the semantics $\mathcal{O}_\Omega(P)$ correctly captures the answer constraint observable when considering also programs union, that is, $\mathcal{O}_\Omega(P)$ is correct with respect to the equivalence $\approx_\Omega$ (corollary B.1). Theorem B.3 will show the compositionality of the semantics with respect to the $\cup_\Omega$ operator.

LEMMA B.3. *Let $P$ be a program. Then, $c_0, p(\tilde{t}) \rightsquigarrow_{P,R} c_1, G$ iff*

- *$true, p(\tilde{X}) \rightsquigarrow_{P,R} c_2, G$*
- *$c_1$ and $c_2 \cup c_0 \cup \{\tilde{X} = \tilde{t}\}$ are $\Re$-solvable and have the same solution, for the variables $\tilde{X}$.*

**Proof**
$\Rightarrow$) Let us first note that the goal $c_0, p(\tilde{t})$ and $c_0 \cup \{\tilde{X} = \tilde{t}\}, p(\tilde{X})$ (where $\tilde{X}$ are new distinct variables) are equivalent Moreover if the constraint $c \cup c'$ is $\Re$-solvable, the constraint $c$ is $\Re$-solvable too. Hence in the derivation of $true, p(\tilde{X})$ we can repeat exactly the steps (and use the same rules) of the derivation of $c_0 \cup \{\tilde{X} = \tilde{t}\}, p(\tilde{X})$ and the first part of the thesis holds.
$\Leftarrow$) Assume now $true, p(\tilde{X}) \rightsquigarrow_{P,R} c_2, G$, and let $c^*$ be the constraint in a generic goal of this derivation. By definition of derivation, $c^* \subseteq c_2$. Therefore, since $c_2 \cup c_0 \cup \{\tilde{X} = \tilde{t}\}$ is $\Re$-solvable, $c^* \cup c_0 \cup \{\tilde{X} = \tilde{t}\}$ is $\Re$-solvable too. Then by repeating each step of the derivation $true, p(\tilde{X}) \rightsquigarrow_{P,R} c_2, G$ and by adding $c_0 \cup \{\tilde{X} = \tilde{t}\}$ at each constraint, we get a derivation $c_0 \cup \{\tilde{X} = \tilde{t}\}, p(\tilde{X}) \rightsquigarrow_{P,R} c_0 \cup \{\tilde{X} = \tilde{t}\} \cup c_2, G$. By the same argument, we obtain a derivation $c_0, p(\tilde{t}) \rightsquigarrow_{P,R} c_1, G$ which completes the proof.

∎

THEOREM B.2. *Let $P$ be an $\Omega$-open program. Then $P \approx_\Omega \mathcal{O}_\Omega(P)$.*

**Proof**
We have to show that for every $\Omega$-open $Q$ such that $\mathcal{O}_\Omega(P) \cup_\Omega Q$ and $P \cup_\Omega Q$ are defined, $true, G \longrightarrow^*_{\mathcal{O}_\Omega(P) \cup Q} c$ iff $true, G \longrightarrow^*_{P \cup Q} c'$ where $\tilde{Y} = Var(G)$ and $c$, $c'$ have the same solutions for the variables $\tilde{Y}$ (note that if $P \cup_\Omega Q$ is defined then also $\mathcal{O}_\Omega(P) \cup_\Omega Q$ is defined but the converse is not true). By the independence from the selection rule, considering a CLP-like version of SLD derivation [40], such as in Definition B.8 we can assume that the selection of the atoms is performed according to the following rule denoted by $S$

1. select first the non-open atoms (i.e. the $p(\tilde{t})$'s such that $p \notin \Omega$)

2. among the non-open atoms, select first those which are added to the current resolvent by the last inference step (i.e. those in the body of the last used rule).

We will show that $true, G \rightsquigarrow_{\mathcal{O}_\Omega(P)\cup Q,S} c', R$ in one step iff there exists $n$ such that $true, G \rightsquigarrow_{P\cup Q,S} c'', R$ in $n$ steps and $c'$ , $c''$ have the same solutions for the variables $\tilde{Y}$. The thesis follows from the above result by a straightforward inductive argument and by definition of $\approx_\Omega$. Let $p(\tilde{t})$ be the atom selected in $G$. If $p(\tilde{t})$ is reduced by using a rule in $Q$, the thesis follows with $n = 1$. Otherwise let $R$ be defined as follows

1. $R = c, (G \setminus p(\tilde{t})), B$ is the first resolvent ($\neq G$) in the derivation $true, G \longrightarrow^*_{P\cup Q,S} c$ such that $S(R) = A$ and either $Pred(A) \in \Omega$ or $A \in G$. Note that by definition of $S$ $Pred(B) = \Omega$.

2. If there does not exist an $R$ as specified in 1, then $R$ is the empty resolvent.

Such an $R$ exists since we are considering finite (successful) derivations. For $R$ as specified in 1. we have

| | | |
|---|---|---|
| $true, G \rightsquigarrow_{P\cup Q,S} c, R$ | iff | (by Definition of $R$ and $S$) |
| $c_0, p(\tilde{t}) \rightsquigarrow_{P,S} c, B$ | iff | (by Lemma B.3) |
| $true, p(\tilde{X}) \rightsquigarrow_{P,S} c', B$ | iff | by Definition B.10 and Proposition B.1 |
| $p(\tilde{X}) \leftarrow c', B \in \mathcal{O}_\Omega(P)$ | iff | (by Definition of $\rightsquigarrow_{\mathcal{O}_\Omega(P),S}$) |
| $true, G \rightsquigarrow_{\mathcal{O}_\Omega(P),S} c'', (G \setminus p(\tilde{t})), B$ | | |

where $c'' = c' \cup c_0 \cup \{\tilde{X} = \tilde{t}\}$. Then, by Lemma B.3, $c'$ and $c$ have the same solutions for the variables of $\tilde{X}$ and therefore by Proposition B.1 $c$ and $c''$ have the same solutions for the variables of $G$. For $R$ as in 2 the same holds with $R, B$ and $G \setminus p(\tilde{t}) = \emptyset$ and this completes the proof.

∎

COROLLARY B.1. *Let $P_1, P_2$ be $\Omega$-open programs. If $\mathcal{O}_\Omega(P_1) = \mathcal{O}_\Omega(P_2)$ then $P_1 \approx_\Omega P_2$.*

**Proof**
Straightforward by Theorem B.2.

∎

LEMMA B.4. *Let $P$ be a constraint logic program, $G$ be a goal and $\tilde{Y} = Var(G)$. Then $true, G \rightsquigarrow_{P,R} c, N$ iff $true, G \rightsquigarrow_{P,R'} c', N'$, where $c$ and $c'$ have the same solutions for the variables $\tilde{Y}$, and the derivation $true, G \rightsquigarrow_{P,R'} c', G'$ is obtained from $true, G \rightsquigarrow_{P,R} c, N$ by changing the order in which the atoms are selected.*

**Proof**
Straightforward by noting that the computation is performed by accumulating constraints and, since constraints are considered as sets, the ordering in which constraints are added is not relevant.

∎

THEOREM B.3. *Let $P_1$ be an $\Omega_1$-open program, $P_2$ be an $\Omega_2$-open program and let $P_1 \cup_\Omega P_2$ be defined. Then $\mathcal{O}_\Omega(\mathcal{O}_{\Omega_1}(P_1) \cup_\Omega \mathcal{O}_{\Omega_2}(P_2)) = \mathcal{O}_\Omega(P_1 \cup_\Omega P_2)$.*

**Proof**
First note that, by Definition B.10, $Pred(\mathcal{O}_\Omega(P)) \subseteq Pred(P)$. Therefore, by Definition B.2, if $P_1 \cup_\Omega P_2$ is defined then also $\mathcal{O}_{\Omega_1}(P_1) \cup_\Omega \mathcal{O}_{\Omega_2}(P_2)$ is defined. By Definition B.10 and by Proposition B.1, it is then sufficient to show that $\exists R$ such that

$$true, p(\tilde{X}) \rightsquigarrow_{P_1 \cup P_2, R} c, B_1, \ldots, B_n$$

iff $\exists R'$ such that

$$true, p(\tilde{X}) \rightsquigarrow_{\mathcal{O}_{\Omega_1}(P_1) \cup \mathcal{O}_{\Omega_2}(P_2), R'} c, B_1, \ldots, B_n$$

with $Pred(B_1, \ldots, B_n) \subseteq \Omega$. Let us prove the two implications separately.
$\Leftarrow$) Assume, without loss of generality, that

$$\{\tilde{X} = \tilde{t}\}, A_1, \ldots, A_{i-1}, p(\tilde{t}), A_{i+1}, \ldots, A_n \rightsquigarrow_{\mathcal{O}_{\Omega_1}(P_1), R'}$$

$$\{\tilde{X} = \tilde{t}\} \cup \{\tilde{l} = \tilde{t}\} \cup c, A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots, A_n$$

in one step, using the rule $p(\tilde{l}) \leftarrow c, B_1, \ldots, B_n \in \mathcal{O}_{\Omega_1}(P_1)$. By Definition B.10 and by Proposition B.1, $\exists R$ such that $true, p(\tilde{X}) \rightsquigarrow_{P_1, R} c', B_1, \ldots, B_n$ with $c'$ and $c \cup \{\tilde{X} = \tilde{l}, \tilde{t} = \tilde{l}\}$ have the same solutions for the variables $\tilde{X}$. Then, by Lemma B.3, $true, p(\tilde{t}) \rightsquigarrow_{P_1, R} c'', B'_1, \ldots, B'_n$ and $c'$, $c''$ have the same solutions for the variables $\tilde{X}$. Hence in $P_1$ there exists the derivation

$$\{\tilde{X} = \tilde{t}\}, A_1, \ldots, A_{i-1}, p(\tilde{t}), A_{i+1}, \ldots, A_n \rightsquigarrow_{P_1, R}$$

$$\{\tilde{X} = \tilde{t}\} \cup c'', A_1, \ldots, A_{i-1}, B'_1, \ldots, B'_n, A_{i+1}, \ldots, A_n$$

.
By definition of $B'_1, \ldots, B'_n$ and since the bindings for variables in $A_1, \ldots, A_n$ are determined by the variables in $p(\tilde{t})$ we have that $c''$ and $\{\tilde{l} = \tilde{t}\} \cup c$ have the same solutions for the variables $\tilde{X}$ Therefore the thesis holds by a straightforward inductive argument.
($\Rightarrow$) Suppose that $true, p(\tilde{X}) \rightsquigarrow_{P_1 \cup P_2, R} c, B_1, \ldots, B_n$ with
$Pred(B_1, \ldots, B_n) \subseteq \Omega$, and, without loss of generality, suppose that the first rule used in the derivation is in $P_1$. By Lemma B.4 we can assume that the selection rule $R$ is $S$ as specified in the proof of Theorem B.2 (considering as non-open atoms the $p(\tilde{t})$'s such that $p \notin \Omega_1$). Let $c', N$ be defined as follows

1. $c', N = c', (G \backslash p(\tilde{X})), B$ is the first resolvent ($\neq G$) in the derivation $true, p(\tilde{X}) \rightsquigarrow_{P_1 \cup P_2, \mathcal{S}}$ $c, B_1, \ldots, B_n$ such that $S(R) = A$ and $Pred(A) \in \Omega_1$,

2. $N = c', B_1, \ldots, B_n$ if there not exist any $c', N$ as specified in 1.

Note that, by definition of $\cup_\Omega$, if $P_1 \cup_\Omega P_2$ is defined then $Pred(P_1) \cap Pred(P_2) \subseteq (\Omega_1 \cap \Omega_2) \subseteq \Omega_1$. Therefore, every atom $A$ selected before $c', N$ is rewritten using a rule in $P_1$. Moreover note that, by definition of $\mathcal{S}$, $Pred(N) \subseteq \Omega_1$.

Therefore, by Definition B.10, $p(\tilde{X}) \leftarrow c', N \in \mathcal{O}_{\Omega_1}(P_1)$. Obviously, $p(\tilde{X}) \leftarrow c', N \in \mathcal{O}_{\Omega_1}(P_1)$ iff $true, p(\tilde{X}) \rightsquigarrow_{\mathcal{O}_{\Omega_1}(P_1), R} c', B_1, \ldots B_n$ in one step. Then we have the following implications

$true, p(\tilde{X}) \rightsquigarrow_{P_1 \cup P_2, S} c, B_1, \ldots B_n$

(iff by definition of $N$ and of $S$)

$true, p(\tilde{X}) \rightsquigarrow_{P_1, S} c', N \rightsquigarrow_{P_1 \cup P_2, S} c' \cup c'', B_1, \ldots, B_n$

(iff by previous remarks)

$true, p(\tilde{X}) \rightsquigarrow_{\mathcal{O}_{\Omega_1}(P_1), S} c', N \rightsquigarrow_{P_1 \cup P_2, S} c' \cup c'', B_1, \ldots, B_n$

where $c$ and $c' \cup c''$ have the same solutions for the variables $\tilde{X}$. Therefore the thesis follows by induction.

∎

## B.4 Semantics and compositional semantics

We have defined the notion of open program and the corresponding compositional semantics. Therefore, we have now the formal setting (i.e., language and semantics) for modular construction of CLP programs and thus for U-Datalog database. However, once we have this nice formal setting, we would also like to hide the information about the fact that a program was made of several components. That is, we want to consider the program no more as an open one. This means that we want to hide information about the structure whenever it is not necessary. In other words we want to remove the rules from the semantic domain. In the remainder of this chapter we show the relationship between the two semantics for CLP programs ($\mathcal{O}(P)$) (Definition A.14) and that for compositional CLP one ($\mathcal{O}_\Omega(P)$) (Definition B.10). Theorem B.4 will show the expected result between those two semantics. By $Proj$ we denote a function which maps a set of conditional constrained atoms into a set of annoted atoms.

DEFINITION B.12. *Let $I$ be set of (class of equivalence of) rules of the form $H \leftarrow c, B_1, \ldots, B_k$. Then*

$$Proj(I) = \{H \leftarrow c, B_1, \ldots, B_k \in I \mid Pred(B_1, \ldots, B_k) = \emptyset\}.$$

THEOREM B.4. *Let $P$ be a program. Then $\mathcal{O}(P) = Proj(\mathcal{O}_\Omega(P))$.*

**Proof**
By Definitions A.14 and B.10 $\mathcal{O}(P) \subseteq \mathcal{O}_\Omega(P)$. Indeed if $\Omega = \emptyset$, $\mathcal{O}_\emptyset(P) = \mathcal{O}(P)$. The function $Proj$ returns exactly the (equivalence classes of) sets of constrained atoms, that is rules such that $Pred(B_1, \ldots, B_k) = \emptyset$ and thus the thesis holds.

∎

The following example summarize the various results provided in this appendix.

EXAMPLE B.2. *Let $EDB$ and $IDB$ be the components of the program of Example B.1, that is the following $\Omega$-open programs with $\Omega = \{parent\}$.*

$$
\begin{aligned}
EDB = \;& \texttt{parent(henry, peter).} \\
& \texttt{parent(bob, peter).} \\
& \texttt{parent(peter, john).}
\end{aligned}
$$

$$
\begin{aligned}
IDB = \;& \texttt{anc(X, Y)} \leftarrow \texttt{parent(X, Y).} \\
& \texttt{anc(X, Z)} \leftarrow \texttt{anc(X, Y), parent(Y, Z).} \\
& \texttt{rmanc(X, Z)} \leftarrow -\texttt{parent(X, Y), rmanc(X, Y), parent(Y, Z).} \\
& \texttt{rmanc(X, Y)} \leftarrow -\texttt{parent(X, Y), parent(X, Y).}
\end{aligned}
$$

*Then $\mathcal{O}_\Omega(EDB) = EDB$ and*

$$
\begin{aligned}
\mathcal{O}_\Omega(IDB) = \{ \;& anc(X,Y) \leftarrow parent(X,Y), \\
& anc(X,Y) \leftarrow parent(X,Z_1), parent(Z_1,Y), \\
& \vdots \\
& anc(X,Y) \leftarrow parent(X,Z_1), \ldots, parent(Z_n,Y), \\
& rmanc(X,Y) \leftarrow -parent(X,Y), parent(X,Y), \\
& rmanc(X,Y) \leftarrow -parent(X,Z_1), -parent(Z_1,Y), \\
& \qquad\qquad\qquad parent(X,Z_1), parent(Z_1,Y), \\
& \vdots \\
& rmanc(X,Y) \leftarrow -parent(X,Z_1), \ldots, -parent(Z_n,Y) \quad \}
\end{aligned}
$$

*$\mathcal{O}_\Omega$ contains enough information to compute the semantics of the composition. In fact $\mathcal{O}(EDB \cup IDB) \subseteq \mathcal{O}_\Omega(EDB \cup IDB)$, that is*

$$
\begin{aligned}
\mathcal{O}_\Omega(EDB \cup IDB) = \{ \quad & parent(henry, peter), \\
& parent(bob, peter), \\
& parent(peter, john), \\
& anc(henry, peter), \\
& anc(bob, peter), \\
& anc(peter, john), \\
& anc(bob, iohn), \\
& anc(henry, john), \\
& anc(X, Y) \leftarrow parent(X, Y), \\
& anc(X, Y) \leftarrow parent(X, Z_1), parent(Z_1, Y), \\
& rmanc(henry, peter) \leftarrow -parent(henry, peter), \\
& rmanc(bob, peter) \leftarrow -parent(bob, peter), \\
& rmanc(peter, john) \leftarrow -parent(peter, john) \\
& rmancr(bob, iohn) \leftarrow -parent(bob, peter), \\
& \qquad\qquad -parent(peter, john), \\
& rmanc(henry, john) \leftarrow -parent(henry, peter), \\
& \qquad\qquad -parent(peter, john), \\
& rmanc(X, Y) \leftarrow -parent(X, Y), parent(X, Y), \\
& rmanc(X, Y) \leftarrow -parent(X, Z_1), -parent(Z_1, Y), \\
& \qquad\qquad parent(X, Z_1), parent(Z_1, Y) \quad \}
\end{aligned}
$$

*and $\mathcal{O}_\Omega(EDB \cup IDB) = \mathcal{O}_\Omega(\mathcal{O}_\Omega(IDB) \cup \mathcal{O}_\Omega(EDB))$ according to Theorem B.3. Then, according to Theorem B.4, $\mathcal{O}(EDB \cup IDB) = Proj(\mathcal{O}_\Omega(EDB \cup IDB))$ with*

$$
\begin{aligned}
\mathcal{O}(EDB \cup IDB) = \{ \quad & parent(henry, peter), \\
& parent(bob, peter), \\
& parent(peter, john), \\
& anc(henry, peter), \\
& anc(bob, peter), \\
& anc(peter, john), \\
& anc(bob, iohn), \\
& anc(henry, john), \\
& rmanc(henry, peter) \leftarrow -parent(henry, peter), \\
& rmanc(bob, peter) \leftarrow -parent(bob, peter), \\
& rmanc(peter, john) \leftarrow -parent(peter, john) \\
& rmancr(bob, iohn) \leftarrow -parent(bob, peter), \\
& \qquad\qquad -parent(peter, john), \\
& rmanc(henry, john) \leftarrow -parent(henry, peter), \\
& \qquad\qquad -parent(peter, john) \quad \}
\end{aligned}
$$

$\diamond$