A Bottom-Up Interpreter for a Database Language with Updates and Transactions*

E. Bertino [1], B. Catania [1], G. Guerrini[2], M. Martelli[2], D. Montesi[†3]

1) Dipartimento di Informatica
Università di Milano
Via Comelico, 39 20133 Milano, Italy
bertino@hermes.mc.dsi.unimi.it, catania@ghost.dsi.unimi.it

2) Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Viale Benedetto XV, 3 16132 Genova, Italy
guerrini/martelli@disi.unige.it

3) Informatics Department
Rutherford Appleton Laboratory
Chilton, Didcot, Oxon OX11 0QX, UK
danilo@inf.rl.ac.uk

Abstract

Deductive databases with updates in rule bodies do not allow bottom-up execution model. This is due to the introduction of control in rule bodies. However, bottom-up execution models are very important due to the set oriented query-answering process of database systems. In [4] we have proposed a rule language to avoid the above drawback and to provide transaction optimization through transaction transformation. In this paper we describe a prototype that provide a bottom-up meta interpreter for the database rule language and will allow to check the validity of future extensions theoretical conjecture about transaction optimization and integrity constraints. The experience in the use of KBMS1 as a tool to develop a run time support for the rule language is reported together with an overview of the system architecture.

*This work has been supported by a grant from the Italian National Research Council.
†The work of D. Montesi has been partly supported by the ERCIM fellowship *Information and Knowledge Systems*.

1 Introduction

Recent database languages have been deeply influenced by the introduction of Horn clauses languages. Indeed, the database community has realized that rule languages are the most natural tool for uniformly modeling several database concepts such as: data, views, constraints and queries. However, two fundamental capabilities, needed by database applications, are not provided by Horn clauses languages: updates and transactions. Indeed, databases contain a large set of information which can be queried and updated. Due to the nature of data, the update language should be integrated with the query language. In many cases, data to be modified are determined by issuing queries. Moreover, updates should be collected into atomic execution units which are executed in *all-or-nothing* style, that is, as transactions. Transactions are a crucial functionality, since data integrity is a major requirement for database applications. In [4] we presented a new language based on a new approach to integrate a declarative query language with an update language. Such integration is achieved by taking into account the transactional behavior. The resulting language, called U-Datalog, provides both update and query capabilities and has a formal declarative semantics. Our approach is based on a two phases computation. In the first phase updates are collected and their consistency is checked. In the second phase the updates are executed together modeling a transactional behavior. The formal semantics is given in [8], while two optimization techniques are presented in [7]. We refer the reader to [9] for additional details. The computation model chosen for the U-Datalog interpreter is based on a bottom-up strategy. This is very important for two reasons: in database context the answer to a query is a set, hence the bottom-up strategy is the most appropriate; however, Datalog with updates in rule bodies do not fit into the bottom-up strategy.

In this paper we describe run time support for such language. In particular we describe the organization of a meta interpreter computing in a bottom-up style our rule language with update in rule bodies. Such meta interpreter has been realized on top of KBMS1 - a Prolog extension developed at Hewlett-Packard Laboratories in Bristol, providing modular program construction (through theory composition) [13]. The main feature of the U-Datalog interpreter is a modular organization of the architecture, which results from the use of software engineering principles such as modular program development and composition. Such modular organization will be useful for several future extensions of U-Datalog.

The remainder of this paper is organized as follows. Section 2 introduces the rule language and sketches its semantics through some examples. Section 3 presents the organization of the meta interpreter based on the bottom-up evaluation strategies. Moreover it introduces KBMS1 main features. Section 4 presents the architecture of the systems and the implementation choices. Finally, in Section 5 future work is outlined.

2 U-Datalog

A Datalog program consists of a set of *base relations* (EDB) and a set of *rules* (IDB). Many extensions to Datalog have been proposed to express updates (see [1] for a survey). In the following we summarize our approach based on non-immediate update semantics. Update-Datalog (U-Datalog) is a rule language which allows declarative specification of updates in program rules. The execution model of U-Datalog consist of two phases, the marking phase and the update phase [18]. The first phase collects the updates found during the evaluation process without, however, executing them. During the update phase they are executed altogether only if they are ground and consistent. If the set of updates is not consistent, or if it contains non-ground updates, the query is *aborted* and no update in the set is performed. The notion of consistency is an important one, in that it prevents a set of updates containing both an insertion and a deletion of the same fact to be executed. By contrast in DLP and $\mathcal{LDL}$, updates are executed as soon as they are evaluated, that is, they are executed as side effect of the derivation process. In this section we recall some basic notions on the syntax and the semantics of U-Datalog, which are defined by means of an instance of constraint logic programming schema (CLP) [12] called $CLP(\mathcal{AD})$ [9].

Updates in U-Datalog are in rule bodies. In addition we consider also bindings in rule bodies which are defined by means of a set of equations (this is related to the fact that U-Datalog is an instance of CLP). Updates to base relations are expressed as a set of special atoms prefixed by $\pm$. The relations can be either extensional or intensional. The current version of our language allows only updates to extensional relations.

Definition 2.1 *(Extensional database) The EDB is a set of ground (ie, without variables) relations. A state $EDB \in S$ is a (possibly empty) set of ground relations. S denotes the set of all possible database states.*

In the following we denote with $EDB_i, i = 1, \ldots, n$ the possible extensional databases.

Definition 2.2 *(Intensional database) The* intensional database *IDB is a set of rules of the form*

$$H \leftarrow b_1, \ldots, b_k, u_1, \ldots, u_s, B_1, \ldots, B_t.$$

where $B_1, \ldots, B_t$ (as in Datalog) is the query part, $u_1, \ldots, u_s$ is the update part and $b_1, \ldots, b_k$ is the binding part. The update and query parts cannot be both empty.

The intuitive meaning of a rule is: "if $B_1, \ldots, B_t$ is true, the bindings $b_1, \ldots, b_k$ and the updates $u_1, \ldots, u_s$ are consistent, then H is true". The notion of consistency is given informally. Intuitively, the bindings $X = bob, X = tom$ are not consistent, while the bindings $X = Y, Y = bob$ are consistent. Similarly, the updates $+p(X), -p(X)$, i.e. complementary updates, are not consistent. The updates $+p(Y), -p(X)$ could be consistent if the related bindings were for example $X = tom, Y = bob$. By contrast with the bindings $X = tom, Y = tom$, $+p(Y), -p(X)$ are not consistent.

Definition 2.3 *(Query) A* query *(or simple query) is a rule with no head of the form*

$$b_1, \ldots, b_k, u_1, \ldots, u_s, B_1, \ldots, B_t$$

where $B_i's, u_j's$ and $b_h's$ are as in Definition 2.2 and $B_1, \ldots, B_t$ cannot be empty.

The condition that $B_1, \ldots, B_t$ cannot be empty is due to the fact that the update phase must always follow the marking phase. Therefore, before updating a database, it must be queried, by means of a query in order to compute the bindings for the variables of the language. We refer to a query also as a *simple transaction*, to stress the transactional behavior of a query. Following the tradition in the examples we always prefix a query with the symbol '?'. A *complex transaction T* is a sequence of transaction $T_1; \ldots; T_n$. In the following the words query, goal, update query, and transaction are synonymous.

Definition 2.4 *(U-Datalog) An* U-Datalog program *with update (or database) $DB = IDB \cup EDB$ consists of the extensional database EDB and of the intensional database IDB.*

Example 2.1 *Consider $EDB_i = q(b)$ and*

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

The user transaction $T_1 = ?r(X)$ evaluated in $EDB_i \cup IDB$ computes the binding $X = b$ and collects the updates $-q(b), +t(b)$. Note that such updates form a transaction that we call induced transaction. Informally the new extensional database $EDB_{i+1} = t(b)$ is the result of the application of these updates to EDB_i. The transaction $T_2 = ?s(X)$ evaluated in $EDB_{i+1} \cup IDB$ computes the binding $X = b$ and does not compute any update, thus the new extensional database is still EDB_{i+1}. The transaction $T_3 = ? + q(X), s(X)$ evaluated in $EDB_{i+1} \cup IDB$ computes the binding $X = b$ and collect the update $+q(b)$, thus the new extensional database is $EDB_{i+2} = t(b), q(b)$. The transaction $T_4 = ? + q(X), p(X)$ computes the binding $X = b$, and collects the updates $+q(b), -q(b)$. They are not consistent and therefore T_4 aborts. $\diamond$

The semantics of an U-Datalog program is given in three steps. The first step semantics models the marking phase. We note that database systems use as default a set-oriented semantics, that is, the query-answering process computes a set of answers [11]. Therefore

$$Set(T, IDB \cup EDB) = \{\langle b_j, u_j \rangle \mid T \longmapsto^* \langle \tilde{b}_j, \tilde{u}_j \rangle \quad \}$$

denotes the set of pairs (bindings and updates) computed as the consistent answers of the transaction T. Such answers are computed in a bottom-up style. This semantics does not include the execution of the collected updates neither considers the transactional behavior. In order to model these features we define the semantics of a transaction T with respect to a database $IDB \cup EDB$. Before we define a function that performs the updates.

Definition 2.5 *Let EDB_i be the current database state and u be a consistent set of ground updates. Then the new database EDB_{i+1} is computed by means of the function $\Delta : 2^{\mathcal{B}^e} \times 2^U \rightarrow 2^{\mathcal{B}^e}$ 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 $2^{\mathcal{B}^e}$ is the set of possible database states and 2^U is the set of possible updates.

The second step provides the transactional behavior, modeling the update phase; the hypothetical updates computed by the marking phase are executed with a transactional mechanism, i.e. an *all or nothing* style. As *observable property* of a transaction we consider *the set of answers, the database state* and *the result of the transaction* itself, which can be *Commit* o *Abort*. We consider the set of updates collected by the marking phase, to which the bindings have been applied. The set $\bar{u}$ is obtained as the union of all the updates gathered by the different solutions, appropriately instantiated. It can be a ground consistent set of updates. If so, the result of the marking phase is a set of bindings and a set of hypothetical updates. If the updates collected are consistent and ground, the new database state is computed and the transaction commits. If $\bar{u}$ is not ground or it is inconsistent, the transaction aborts. Note that in such a way we model a set-oriented transactional behavior. The set of possible observable Oss_i is OSS. In the following we define the semantics of a transaction T with respect to the intensional database IDB as a function from extensional database to observable. In the following, $2^{\mathcal{B}^e}$ denotes the set of possible extensional databases.

Definition 2.6 *(Semantics of a transaction) Let $DB_i = IDB \cup EDB_i$ be the database. The* semantics *of a transaction is denoted by the function $\mathcal{S}_{IDB}(T) : 2^{\mathcal{B}^e} \longrightarrow OSS$. If a transaction T has the form $\tilde{b}, \tilde{u}, \tilde{G}$, then*

$$\mathcal{S}_{IDB}(T)(EDB_i) = \begin{cases} Oss_{i+1} & \text{if } OK \\ \langle \emptyset, EDB_i, Abort \rangle & otherwise \end{cases}$$

where $Oss_{i+1} = \langle \{b_j \mid \langle b_j, u_j \rangle \in Set(T, DB_i)\}, EDB_{i+1}, Commit \rangle$, EDB_{i+1} is computed by means of $\Delta(EDB_i, \bar{u})$. The condition OK expresses the fact that the set $\bar{u} = \bigcup_j u_j b_j$ is consistent, that is, there are no complementary ground updates. $u_j b_j$ denotes the ground updates obtained by substituting the variables in u_j with the ground terms associated with the variables in b_j. denotes the n-th component of the tuple Oss_i.

Example 2.2 *Consider the database $DB_1 = IDB \cup EDB_1$, where*

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

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

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

where $EDB_2 = \{t(a), t(b)\}$. *The semantics of* $T_2 = ?\ \mathtt{s(X)}$ *is*

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

The semantics of $T_3 = ?\ \mathtt{X = c, k(X)}$ *is*

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

with $EDB_3 = \{t(a), t(b), t(c)\}$ *and* $Fix(DB_3) = Proj(\mathcal{O}_e(\mathcal{O}_e(IDB) \cup \mathcal{O}_e(EDB_3)))$. *The semantics of* $T_4 = ?\ \mathtt{X = a, +t(X), s(X)}$ *is*

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

$\diamond$

The third step semantics is related to complex transactions.

Definition 2.7 *(Sequence) Let* $DB_i = IDB \cup EDB_i$ *be the database and* $T_1; T_2$ *be a transaction. The semantics of* $T_1; T_2$ *is denoted by the function* $\mathcal{S}_{IDB}(T_1; T_2) : OSS \longrightarrow OSS$.

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

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

Therefore, according to the above definition, the abort of a simple transaction in a sequence results in the abort of the entire sequence.

Example 2.3 *Consider the database*

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

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

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

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

- *The semantics of* $T' = ?\ \mathtt{r(X)}; ?\ \mathtt{s(X)}; ?\ \mathtt{X = a, k(X)}$ *is*

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

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

$\diamond$

3 Meta interpreter for U-Datalog

Several evaluation strategies have been investigated for rule based languages [2]. Our application provides a bottom-up strategy for the U-Datalog. Such strategy is implemented through a meta interpreter. Note that classic top-down meta interpreter is defined with few Prolog line code [17] and can be easily extended to CLP. However, to the best of our knowledge there are not bottom-up meta interpreters for the CLP schema. The emphasis of our application is not on an efficient run time support, rather is on extensibility and modularity in order to develop a core system which can be extended to test the validity of some ideas related to the transaction optimization and to constraint languages. The goal of the transaction optimization techniques we are developing is to reduce the transaction execution times by detecting invariant/redundant updates within the same transaction. Invariant and redundant updates can be indeed omitted from the execution of transactions without changing the final result of transaction executions. Thus, we are interested in a logical optimization rather than in a physical one. Similarly, our work on integrity constraints deals with optimizing constraint checking within complex transactions [3, 15, 16].

The meta interpreter allows to make all the above mentioned experiments. Currently, the core system supporting the full U-Datalog, introduced in Section 2, has been completely implemented. Ongoing extensions are related to the transaction optimization and constraint language. Integrity constraints in the database language are expressed as constraints in CLP. Thus the constraint solver will be extended by plugging a new constraint solver into the CLP meta interpreter. Note that this extensibility feature derives from the use of a CLP schema. The use of KBMS1 is related to the fact that its features nicely fit into our requirements in order to make the U-Datalog system a tool to check the validity of our ideas. However, other systems such as Gödel, [10] and Eclipse could be used. Hereafter we discuss the major features of KBMS1.

3.1 KBMS1

A Knowledge Base Management System (KBMS) is a programming system providing: (i) a declarative language, not necessarily a logic language, used both as query language and as host language; (ii) the main features of a database system (efficiency, data sharing, reliability and so on).

KBMSs represent for knowledge what databases represent for data. Data comprises facts, while knowledge comprises facts and rules. A particular domain knowledge can be shared among different users and applications, using the same logical paradigm, because a single unifying language is used for representing both facts and rules. A KBMS supports representation, storage, retrieval and update of great amounts of knowledge. Moreover a KBMS often provides tools for metaprogramming and for knowledge modularization.

Among the KBMSs, KBMS1 is a Logic Programming-based system which enables the storage, retrieval and update of very large volume of knowledge, expressed in an extended Prolog-like language. It supports persistent knowledge bases by providing a very tight integration between the interpreter and the storage manager. In the following subsections we briefly describe the main characteristics of the system, and finally we provide a few motivations on why to use KBMS1 instead of Prolog.

3.1.1 Theories

The main concept behind KBMS1 is the *theory*. A theory is a partition of the knowledge base; it is an unordered set of procedures, where a procedure is an ordered set of clauses which possesses the same primary functor and arity.
The main characteristics of theories can be summarized as follows.

- Theories are *first class objects*; so they are objects which can be modified by the programming language supported by the system.
- Theories are an exhaustive partitioning of the knowledge base. Therefore, the customary global database of standard Prolog is no longer needed.
- Theories can be modified in a completely declarative way through update operations. For example, if an update operation U is applied to theory T_1, a new theory T_2 is generated, whereas theory T_1 is left unmodified. The result of applying an update to a theory is in fact another theory.
- Theories allow to overcome some drawbacks and criticised features of standard Prolog. In particular a program cannot make side-effect updates to itself.
- The possibility of performing declarative theory updates leads to a powerful versioning facility, which allows the use of hypothetical reasoning.
- The system heavily uses substructures sharing for reducing the spatial complexity due to the versioning mechanism.
- Theories can be named by the user, so they can be easily identified.

Grouping sets of clauses in theories is useful also because it allows reasoning in multiple theories.

To provide a logical organization of theories, *system theories* are used. A system theory is a meta-theory which holds the correspondence between theories and theory names. Moreover each theory has associated a triple of named meta-theories. Such a triple consists of the following elements:

1. an *interpreter* theory, which is the *query−interface* of the associated theory; all the queries to a theory are in fact handled by its interpreter
2. an *assimilator* theory, which is the *update−interface* of the associated theory; updates on a theory are in fact handled by its assimilator theory

3. an *attribute* theory, which holds metadata related to the associated theory; these metadata are generally used by the interpreter and by the assimilator in the specialized inference process and in the specialized update process, respectively. An attribute theory is a way for abstracting from the specific object theory, allowing the interpreters and assimilators to be used on many different object theories. Examples of metadata that allow to abstract from the specific theory are functional dependencies among predicate arguments, number of clauses in a procedure and integrity constraints to be checked when executing updates.

The correspondence between a theory and its associated triple of meta-theories is kept by the system theory. The main role of the system theory is, in fact, to resolve names and to couple a theory with its metatheories.

Meta-theories are themselves theories and so they in turn have a triple of built-in meta-theories associated with them. These built-in theories are *PrologInt*, *PrologAss* and *none*, respectively. PrologInt is the system interpreter, which performs a Prolog-like inference. PrologAss is a trivial assimilator, which updates a theory exactly as told, without any control. None denotes the empty theory.

Other distinguishing KBMS1 concepts are the notions of *current theory* and of *current system theory*. The current theory is the (user) theory in which the current goal has to been refuted. The current system theory is the system theory that is currently to be used for determining the correspondence between theories and theory names, and between theories and meta-theories.

3.1.2 Other KBMS1 characteristics

The language of KBMS1 is kbProlog, a Prolog extension which integrates the theory mechanism and other features in the usual logic programming. It can be loosely regarded as a Prolog in which operations on the global database have been replaced by non-side-effecting operations on theories. kbProlog programs can be interfaced with C procedures (both call-ins and call-outs). Call-out is useful for encoding performance-sensitive procedures efficiently and for accessing to other systems (graphical systems, conventional databases, and so on). Call-in allows KBMS1 applications to be integrated into larger systems. The system can be used interactively and programmatically. Because of the theory update mechanism the distinction between the two modalities of use is more marked than in standard Prolog.

Another distinguishing KBMS1 feature is related to the support for a *global database*, shared by all the theories. The main difference between theory updates and global database updates is that updates of the first kind have no side-effects on the theory, whereas updates of the second kind do. So if an update U is performed on theory T_1, a new theory T_2 is generated. By contrast the global database is unique, therefore if U is performed on the global database, the global database itself (modified) is returned. Note that the side-effect updates on the global database leads to programs more difficult to understand and to test, but may be useful in many situations. In particular the global database may be used to assert global conditions, to be tested in different modules of a program. The global database increases the kbProlog power, allowing to store information, that, otherwise, would be lost during backtracking.

Assertions of facts in the global database may look like usual Prolog *assert*. Note, however, that global database assertions differ from Prolog asserts in that the kbProlog interpreter considers the deduction in the global database only if explicitly requested, and facts in the global database are not seen in usual deduction. A fact asserted in the global database is therefore considered in the deduction of a goal only if explicitly requested. By contrast, in Prolog there is a unique theory, hence every asserted fact is always considered in all the deductions following the assertion. The global database should however be used in such a way that the declarative interpretation of the program is not lost.

Finally KBMS1 supports the notion of *database theories*. These theories can be seen as collections of relations (holding structured, typed ground facts) and a relation schema. Database theories support non-declarative updates, therefore no versioning is possible. Updates on a database theory do not define a new theory, differing from the starting one for the realized updates, as in usual theories, rather they simply modify the starting theory, that keeps its identity.

3.1.3 Motivations

In this subsection we discuss a few motivations for the use of KBMS1 instead of Prolog. In Section 4, after having described the interpreter we have developed, we remark the KBMS1 features that have been most useful in the development of the interpreter. Generally speaking the main advantages of KBMS1 with respect to Prolog are

(a) modularization

(b) declarative updates.

The advantages of modularization are well-known. The benefits of developing modular programs are often stressed in software engineering field, in that modular program development leads to programs more easily maintained, easier to understand and to test. In the knowledge base field, modularization supports multiple theory reasoning, that is it allows to multiple domains to be handled and different kinds of reasoning to be performed on these domains. For example we may handle knowledge on different domains with a single inference mechanism.

In a usual logic program, in fact, all clauses may be used in the refutation of a goal, while in KBMS1 a goal is always directed to a specific theory, and its refutation only uses the clauses of the specified theory, until a context switch is requested. The refutation of a subgoal may in fact be requested to another specified theory. Advantages of declarative updates to theories can be found in a greater program readability and in the versioning mechanism provided.

4 Architecture of the system

In order to implement a bottom-up interpreter for U-Datalog KBMS1 has been used. Our choice has been motivated by the useful metaprogramming facilities offered by the system. In the following, implementation choices are discussed and an analysis of the tool is performed.

4.1 Implementation choices

In deductive databases, two basic strategies are found for query evaluation, namely top-down and bottom-up strategies. The main advantage of a top-down strategy is the use constants in the query in order to reduce the search space [2]. It is a one tuple at a time approach, in that the execution of a query generates one tuple at a time. Moreover, a top-down evaluation of a query is not ensured to always terminate. By contrast, a bottom-up strategy is a set oriented approach to query evaluation, in that all solutions to a query are returned as results of its execution. In addition the bottom-up strategy does not benefit from constant in the query to reduce the research space. This method is based on the computation of the fixpoint semantics. Whenever the signature is finite, the fixpoint semantics is finite too. Therefore the bottom-up computation always ends. Finally note that, extending Datalog with updates in rule bodies does not fit with the bottom-up strategy.

We note that the main features of a bottom-up computation are the set oriented approach and always ending computations. This properties are not satisfied by a top-down approach, however they are essential in database area. Therefore, to implement a meta interpreter for U-Datalog we have adopted a bottom-up strategy. This choice is made stronger from the fact that rule languages with updates in rule bodies do not fit into this computational model.

4.2 Knowledge partition

In order to implement in KBMS1 a meta interpreter for U-Datalog, the theory mechanism has been used. In particular, the following theories have been defined:

1. a number of user theories
2. an interpreter theory
3. an assimilator theory
4. an attribute theory

In the following these theories are briefly described.

User theories

A U-Datalog database is consists of two components: an intensional database IDB and an extensional database EDB. Such databases contain different information.

EDB is a knowledge base containing all the information about the situation we model. IDB is an inference system, that allows to obtain new intensional information from the extensional one. Moreover the EDB can be updated, whereas the IDB does not. These properties allow to represent every database as a theory.

The current architecture of the U-Datalog interpreter consists of only one extensional database and several intensional ones. Every intensional database is a *view* on our knowledge (the EDB). Therefore it is possible to have several views on the same extensional database. Note that having more than one intensional database allows to maintain different forms of the same view. For example, we can maintain an intensional predicate but also its optimized definition, performing some analysis on the performance of a given intensional predicate. We will refer to the theory representing the extensional database as EDB theory, and similarly to the theory representing an intensional database as IDB theory.

In addition a theory has been defined managing the environment in which the transactions are executed. This theory implements the interface operations. It takes an IDB and a transaction (given in input by the user), converts them in an internal

format and calls the interpreter to execute the transaction on the database composed by the IDB and by the only extensional database implemented in the system. The internal format allows a simple unification of the binding part, of the update part and of the query part of a rule.

Assimilator theory

The assimilator theory facility supported by KBMS1 has been used to implement the update operations on the extensional database theory. In particular, we have implemented two update operations.

- An insert operation $plus(Fact)$, where $Fact$ is the fact to be inserted in the extensional database. The assimilator inserts this fact (by the *assert* predicate) only if $Fact$ is ground and it is not already present in the theory. If the fact is ground and it is already present, no insertion is performed. If the fact is not ground, the operation fails.
- A delete operation $minus(Fact)$ where $Fact$ is the fact to be removed from the intensional database. The assimilator removes this fact (by the *retract* predicate) only if $Fact$ is ground and it is present in the theory. Otherwise the assimilator has the same behavior as in the case of the insert operation.

Attribute theory

The attribute theory, associated with the theory representing the extensional database, has been used for maintaining the information useful for the execution of a transaction. In particular, the attribute theory maintains the fixpoint semantics of the current database. It is updated by the bottom-up interpreter.

Interpreter theory

The interpreter theory associated with the EDB theory implements the bottom-up interpreter for U-Datalog. As it is associated with the EDB theory, all queries are sent to the extensional database. The interpreter has three main tasks:

(a) managing transactions

(b) generating the fixpoint semantics (*marking phase*)

(c) updating the extensional database (*update phase*)

In the following we examine in a little more detail these tasks.

(a) Given a sequence of goals, the interpreter sequentially executes all of them. If the execution of a goal generates an abort condition, the evaluation is terminated. Indeed, in this case the observable returned by the execution of the sequence is determined without evaluating all the goals in the sequence and it is equal to $(\emptyset, EDB, Abort)$, where EDB is the extensional database existing before the execution of the sequence. In this case the extensional theory is not updated. Otherwise, i.e. no abort condition is generated, the goal is evaluated and the extensional theory is updated.

(b) In order to evaluate a goal according to the bottom up strategy, the fixpoint semantics is computed. The semantics refers to the current database, consisting of the current extensional theory and of the chosen IDB. The fixpoint semantics is maintained in the attribute theory and it is updated during the deduction of a sequence of goals.

(c) In order to obtain the solutions of a goal, the attribute theory is inspected. All the obtained solutions are maintained in a list and represents the result of the marking phase. After the execution of a goal, the list obtained as result of the marking phase is inspected to identify possible abort conditions. If no abort conditions arises, i.e. if all the updates generated by the goal are ground and consistent, the extensional theory is updated.

Global database

The global database has been used to maintain temporary information. In particular:

- a fact is asserted in the global database when an abort condition arises
- if no abort conditions arise, the solutions of a transaction are maintained in the global database, and are, then, used in the output operations.

4.3 Analysis of the tool

The main advantages in the use of KBMS1 for the development of an interpreter for U-Datalog can be summarized as follows.

- Intensional and extensional components, which in U-Datalog have a different syntactic forms, have been represented by two different theories, modeling adequately the characteristics of different database components.
- The features of explicit theory update supports an adequate model of the state evolutions resulting from goal executions.
- The mechanism that allows to define a specialized interpreter in an interpreter theory and to associate it with a given theory has been very useful. In this way, every goal reduction in the theory is held using the specified interpreter theory.
- The update functionalities of the assimilator theory have been useful in many respects, for example in the development of the mechanism checking the groundness of updates.

5 Conclusions and future work

We have presented a system developed at University of Genova which is a run time support for a database language supporting queries, updates and transactions. The updates and transactions have been developed through a new approach which nicely fit with the structure of KBMS1. Our system is open and will be used as a prototype to check the validity of future extensions such modular construction based on object oriented paradigm ([6]) as transaction optimization techniques ([5]) and integrity constraint support ([14]). Thus, it can be seen as an open prototyping database system where several new features can be "plugged in" with a limited impact on the other components of the system. This is also a nice feature of the CLP approach which allows the database language to be extended with minimal changes. Indeed, there are two research directions we are currently investigating. The first one is related to transaction optimization by means of some new theories. The second one is related include integrity constraints.

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*, Vol. 326 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, Berlin, 1988.

[2] F. Bancilhon and R. Ramakrishnan. An Amatures's Introduction to Recursive Query Processing Strategies. In *Proc. Int'l Conf. ACM on Management of Data*, pages 16–52, 1986.

[3] E. Bertino, D. Musto. Correctness of Semantic Integrity Checking in Database Management Systems. *Acta Informatica*, 28:25–57, 1988.

[4] E. Bertino, B. Catania, G. Guerrini, M. Martelli and D. Montesi. Formalizzazione e Ottimizazione di Transazioni di Modifica in CLP(AD). Proc. Italian Conference on Logic Programming (GULP'93), 1993.

[5] E. Bertino, B. Catania, G. Guerrini and D. Montesi. Static Analysis of Transactional Intensional Databases. *Proc. Second ICLP-Workshop on Deductive Databases and Logic Programming of International Conference on Logic Programming*, Genova, 1994.

[6] E. Bertino, G. Guerrini and D. Montesi. Deductive Object Databases, To Appear on *Proc. European Conference on Object Oriented*, Bologna, 1994.

[7] E. Bertino, B. Catania, G. Guerrini, and D. Montesi. Transaction Optimization in Rule Databases. 1993. Fourth IEEE Research Issues in Data Engineering: Active Database Systems (RIDE-ADS'94), IEEE Computer Society Press, 1994.

[8] E. Bertino, M. Martelli, and D. Montesi. An Incremental Semantics for CLP(AD). In A. Marchetti, Spaccamela, P. Mentrasti, and M. Venturini Zilli, editors, *Proc. Fourth Italian Conference on Theoretical Computer Science*, pages 53–67. World Scientific, 1992.

[9] 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*, pages 120–132, 1992.

[10] A. D. Burt, P. M. Hill, and J. W. Lloyd. Preliminary Report on the Logic Programming Language Gödel. Technical Report TR 90-02, Computer Science Department, Univeristy of Britstol, 1990.

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

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

[13] J. Manley, A. Cox, K. Harrison, M. Syrett, and D. Wells. KBMS1 A User Manual. Information System Centre Hewlett-Packard Laboratories, March 1990.

[14] D. Montesi and E. Bertino. Queries, Constraints, Updates and Transactions within a Logic-based Language. 1993. *Proc. of International Conference of Information and Knowledge Management*, Washington, 1993

[15] J-M. Nicolas. Logic for Improving Checking in Relational Data Bases. *Acta Informatica*, 18(3):227–253, 1982. Springer-Verlag, Berlin.

[16] F. Sadri and R. Kowalski. A Theorem-Proving Approach to Database Integrity. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pages 313–362. Morgan-Kaufmann, 1987.

[17] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Mass., 1986.

[18] M. Zloof. Query-by-example: a Data Base Language. *IBM Systems Journal*, 16(4):324–343, 1977.