The GPR System: an Architecture for Integrating Active and Deductive Rules on Complex Database Objects

Fabrizio Angiulli
ISI-CNR
c/o DEIS
Università della Calabria
87036 Rende (CS), Italy.
Email: `angiulli@si.deis.unical.it`

Luigi Palopoli
Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria
87036 Rende (CS), Italy.
Email: `palopoli@si.deis.unical.it`

Riccardo Torlone
Terza Università di Roma
c/o IASI–CNR
Viale Manzoni, 30
00185 Roma, Italy.
Email: `torlone@iasi.rm.cnr.it`

This work has been partially supported by Consiglio Nazionale delle Ricerche and by MURST under the project "Sistemi formali e strumenti per basi di dati evolute".

Abstract

This paper illustrates a prototype system, called GPRS, supporting the Generalized Production Rules database language (GPR for short). The language GPR integrates, in a unified framework, active rules, which allow the specification of event driven computations on data, and deductive rules, which can be used to derive intentional relations in the style of logic programming. The prototype realizes the operational semantics of GPR, implementing both active and deductive behaviors. The rule-evaluation engine of the system is unique. The data model of reference is object based and the system is implemented on top of an object oriented DBMS. Thus, the GPRS prototype represents a concrete proposal of an advanced DBMS for complex objects that supports both active and deductive styles in rule programming.

1 Introduction

A significant body of work has been recently done on the problem of improving nowadays database management systems in order to meet the requirements of new application domains (like, for instance, CAD, multimedia, robotics and expert systems). Along this line, several researchers have proposed the integration of production rule languages (traditionally used in expert system shells) within database environments [8, 11, 17, 20, 21, 23, 24, 27, 6, 7], while others have studied the possibility of adding deductive capabilities to database systems [1, 2, 4, 5, 18]. As a result of these studies, advanced research prototype systems have been produced (e.g., LDL++, Coral, Starburst, Postgres).

More recently however, it has been noticed that the integration of these paradigms (active and deductive computations) into a unique homogeneous semantic framework would represent a further important improvement [28]. In fact, from a practical point of view, all the systems developed so far support well only one of these paradigms, thus limiting their applicability in several important application domains. On the other hand, from a theoretical point of view, the two paradigms present strong similarities being based on *rules* that, roughly speaking, connect a cause with an expected effect. However, the very different semantics (both abstract and operational) provided so far for these paradigms make this integration a difficult task to achieve.

This paper illustrates a prototype system, called GPRS, that supports the Generalized Production Rule database language [19]. This language gives the possibility of expressing, into a unique framework, both deduction-based and event-driven computations.

The prototype system has been implemented at DEIS, Università della Calabria. It is programmed in O–SQL with C as the host language on top of the Iris object oriented DBMS. The system runs on workstations Apollo 700 under Unix.

GPRS realizes the operational semantics of GPR using a unique rule-evaluation engine. In order to achieve a better flexibility in programming triggering, rule activation is realized through unification (instead of simple matching) which is implemented by adapting the standard unification algorithm to GPR rules. The system supports an object-based data model for complex objects with identity, and uses the object oriented DBMS for managing persistent data. The architecture features a good degree of modularity and this makes the system ease to extend and modify.

We describe a number of interesting aspects related to the evolutionary design and the development of an integrated active/deductive rule base management system for complex object databases. Among them, we cite the implementation of triggering based on deletions and modifications. Indeed, the implementation of rule triggering caused by deletions must take into account the referential integrity maintenance policy implemented in the underlying object oriented DBMS. As far as triggering caused by modifications is concerned, a general problem to be solved is that of maintaining both the old and the new database state referred by the given modification. Both these problems are solved in GPRS by systematic use of action deferment and logs.

The prototype structure is based on an abstract interpreter that realizes the formal semantics of GPR as described in [19]. The underlying object-oriented DBMS has been used to implement GPR schemes and as the back-end for basic querying and updating on the current database instance.

A first release of the system is available, and experiences with using GPRS for programming a number of applications are going on. The two main applications we are working on are: (1) a traditional academic library application, where alerters are used to signal book-loan time period deadlines and last-copy status for books to lend, and (2) a robot motion planning application, where rules are used to decide the trajectory of a robot from a given starting point to a given target point into a noisy (due to the presence of moving obstacles) reference area. The development of a second release of the system, featuring a graphical user interface based on Motif is ongoing.

The rest of the paper is organized as follows. In Section 2 an overview of the GPR language is presented. The overview is quite informal. The interested reader can refer to [19] for a formal presentation of GPR. Section 3 illustrates the GPRS prototype. In more details, Section 3.1 describes system architecture. Section 3.2 illustrates the parsing process from GPR schemes onto Iris schemes. Section 3.3 describes GPR programs and goals parsing. Implementation of the rule triggering mechanism is illustrated in Section 3.4. Rule execution is described in Sections 3.5–3.7. Finally, in Section 4 we draw our conclusions.

2 The rule language GPR

In this section we informally illustrate the rule language GPR. As already pointed out, the interested reader can refer to [19] for a thorough presentation of this language.

2.1 Structure of rule languages

Traditionally, rule database systems are based on statements of the form:

$$Cause \Rightarrow Effect$$

defining a cause/effect relationship between operations on data. Those rules can be classified according to the kind of operations forming the cause and the effect of the rule. Specifically, a *deductive rule* corresponds to a statement as above where both the cause and the effect are queries. For instance, the rule

$$man(X, Y) \Rightarrow emp(X, Z), dep(Z, Y)$$

specifies that a query to the relation *man* causes a query involving the relations *emp* and *dep*.[1] In a *production* (or *active*) *rule* instead, the effect is generally an update and the cause is a query. In this case, the query states a condition under which the rule should be fired. As an example, the rule

$$emp(X, Y), emp(X, Z), Y \neq Z \Rightarrow -emp(X, Z)$$

states that if there are two tuples in the database denoting a situation where the same employee is associated with different departments, then one of them must be discarded. This general scheme is extended in the context of *ECA rules* [17], where it is possible to explicitly specify the events triggering the rules. Thus, for instance, we can use the ECA rule

$$-dep(Z, Y) \bullet emp(X, Z) \Rightarrow -emp(X, Z)$$

to specify that when a department is deleted (Event) and there are employees working in such a department (Condition), then those employees must be deleted as well (Action). Thus, in the more general form of an active rule, the event and the condition part form the cause whereas the action forms the effect of the rule.

2.2 An overview of GPR

The language GPR is based on a new kind of rule, called *generalized production rule.* A rule of this kind can be viewed as a further generalization of a production rule, in which the role played by various parts is different. More specifically, it has the form

$$E \Rightarrow Q \bullet U$$

where: (1) E (called the *event part*) specifies an *extended event* triggering the rule and forms the cause of the rule; (2) Q (*the query part*) denotes a query to be evaluated on

[1] Actually, deductive rules are often represented using a logic programming style notation, i.e.:

$$man(X, Y) \leftarrow emp(X, Z), dep(Z, Y)$$

the underlying database instance and also states a logical constraint to be satisfied for the rule to be activated; (3) U (the *action part*) specifies a list of updates to be executed on the underlying database when the rule is activated.

A generalized production rules differs from usual production rules mainly in two aspects. Firstly, the condition of a generalized production rule is semantically part of its effect instead of being part of the cause. This allows to use conditions for constructing bindings to be passed not only to the action part but also to the caller of the rule. This makes it possible to capture the semantics of deductive rules. Secondly, general unification is used in the place of simple matching in rule activation. This makes the activation mechanism more general and flexible.

Similarly to an ordinary production rule, a generalized production rule has an event-driven activation (that is, a rule is fired when a particular situation takes place) and its informal meaning is the following: "when E takes place, answer the query Q and if this answer is not empty, then execute the updates specified in U''. Thus, the cause is an event and the effect is a query and (possibly) an update. It follows that the execution of a rule with respect to a certain database state returns both the answer to a query and a change of state. The answer can be returned to the triggering event, which may contain variables, thus simulating a "deductive-like" top-down evaluation of the rule. The execution model of a set of rules is based on the concept of extended event: intuitively, an extended event corresponds to the request (done explicitly by a user or through implicit triggering during the evaluation of a program) to perform some operations on data (queries or updates). Data manipulations can be basic operations (e.g., an insertion) or "logical" operations, defined by a name. These logical names may appear in the query part of a rule and recursive executions can result when the same name appears both in the event and in the query part. This last feature can be exploited to implement recursive queries.

The language is defined in the framework of a simple data model for complex objects with identity, called ODM, that generalizes the relational model in a natural way.

2.3 Examples of GPR rules

We now show the use of GPR language by means of a number of practical examples that refer to a department library system, where books are acquired, borrowed, returned and consulted under certain policies. The ODM database scheme and instance of reference are reported in Figure 1.

In the following examples, atoms preceded by the symbol ! are *generalized query atoms* (which, intuitively, denote complex sequences of data manipulation and querying,

$$typ(\texttt{book}) = [title : string, authors : \{\texttt{author}\}, total : integer, avail : integer],$$
$$typ(\texttt{a\_book}) = [code : integer, book : \texttt{book}, year : integer],$$
$$typ(\texttt{person}) = [name : string, position : string],$$
$$typ(\texttt{loan}) = [pers : \texttt{person}, copy : \texttt{a\_book}, period : [from : date, to : date]],$$
$$typ(\texttt{author}) = [name : string, nation : string].$$

author

	name	nation
a1	Dante	Italy
a2	Shakespeare	England
a3	Machiavelli	Italy
a4	Gorkij	Russia
a5	Marx	Germany
a6	Engels	Germany

book

	title	authors	total	avail
b1	The Commedy	{a1}	2	1
b2	Amlet	{a2}	2	1
b3	The Prince	{a3}	1	1
b4	The Tempest	{a2}	1	0
b5	The Mother	{a4}	2	0
b6	Manifesto	{a5,a6}	1	0

a_book

	code	book	year
c1	12.02	b1	1902
c2	12.80	b1	1980
c3	15.56	b2	1856
c4	15.70	b2	1970
c5	18.55	b3	1955
c6	23.70	b4	1970
c7	35.36	b5	1936
c9	46.66	b6	1966

Figure 1: An *ODM* database scheme and instance

whereas atoms preceded by the symbol ? denote *object query atoms*, i.e., atomic query operations on stored data. Symbols +, –, * denote insertions, deletions and modifications, respectively [19].

To begin with, let us consider simple goals where the event part is missing. The semantics of one such a goal, say $\Rightarrow C \bullet U$, is to modify the underlying database instance according to the updates specified in U, if the condition C is satisfied.

Example 1 *Assume we want to cancel the books having Poe as author, together with all its copies. This can be done using the following rule:*

$$\Rightarrow ?\mathtt{author}(\mathtt{X}, [\mathtt{name} : \text{“}\mathtt{Poe}''\text{''}]), ?\mathtt{book}(\mathtt{Y}, [\mathtt{author} : \mathtt{Z}]), \mathtt{X} \in \mathtt{Z}$$
$$\bullet - \mathtt{book}(\mathtt{Y}), -\mathtt{a\_book}(\mathtt{W}, [\mathtt{book} : \mathtt{Y}]).$$

The object query atoms denoting the condition are evaluated by looking for the unifying substitutions against objects stored in the database. The constructed substitutions are returned to the caller of the goal as the answer. If the answer is not empty, then these substitutions are also applied to the update part, that is then executed for each of them. In the case at hand, the condition is not satisfied in the database of Figure 1, and therefore no update is executed and the empty answer is returned. As particular cases, if no condition is specified, we obtain unconditional updates and if no action is specified, we obtain simple queries on the database. This latter case is illustrated by the following example.

Example 2 *Assume we want to know all the titles written by Shakespeare stored in the database. Then, we can use the following goal:*

$$\Rightarrow ?\mathtt{author}(\mathtt{X}, [\mathtt{name} : \text{“}\mathtt{Skakespeare}''\text{''}]), ?\mathtt{book}([\mathtt{title} : \mathtt{Y}, \mathtt{authors} : \mathtt{Z}]), \mathtt{X} \in \mathtt{Z}$$

Next, we consider rules where generalized query atoms occur. We start with rules with no action part. This corresponds to defining deductive rules.

Example 3 *Assume we need to find people who are late with returning borrowed books. This can be programmed as follows.*

$$\Diamond(!\mathtt{late}(\mathtt{X})) \Rightarrow ?\mathtt{loan}([\mathtt{pers} : \mathtt{X}, \mathtt{period} : [\mathtt{to} : \mathtt{Y}]]), ?\mathtt{today}(\mathtt{Z}), \mathtt{Z} > \mathtt{Y}.$$

where we assume that the evaluation of the (system) object query atom ?today(Z) returns today's date in the variable Z.

Example 4 *Assume we want to know all the books lent to each person. The following rule accomplishes this task:*

$$\begin{aligned}&\Diamond(!\texttt{lent}(\texttt{X},\texttt{Y})) \Rightarrow\\ &\qquad ?\texttt{person}(\texttt{Z},[\texttt{name}:\texttt{X}]), ?\texttt{loan}([\texttt{pers}:\texttt{Z},\texttt{copy}:\texttt{V}]),\\ &\qquad ?\texttt{a\_book}(\texttt{V},[\texttt{book}:\texttt{W}]), ?\texttt{book}(\texttt{W},[\texttt{title}:\texttt{Y}]).\end{aligned}$$

The following goal can be for instance used to know the people who borrowed a copy of the book entitled "Amlet".

$$\Rightarrow !\texttt{lent}(\texttt{X}, \text{“}\texttt{Amlet}\text{”})$$

The execution of the goal above causes the invocation of a generalized query atom. The corresponding event, namely, $\Diamond(!\texttt{lent}(\texttt{X}, \text{“}\texttt{Amlet}\text{”}))$, is then raised. Note that, differently from a traditional event, a generalized event may contain both variables and constants: rules are then activated constructing *unifiers* of their event parts and raised events (instead of using simple matching). After that, the associated body is executed. In the example at hand, the body contains only the query part, which is evaluated and the constructed bindings are composed and returned to the caller. As a result, the name of the people who borrowed the book entitled Amlet are bound to the variable of the goal. Note that here the consequence of a rule is simply a query (that is specified in the query part). Thus, in this case, the rule behaves as (and looks like) a Datalog rule. In a similar fashion we can also define recursive queries, as shown in the following example.

Example 5 *Assume we want to know, for each author X, the names of all the authors that are related to X, where we assume that an author Y is related to X if X and Y were coauthors of a book or X and Z were coauthors of a book and Z is related to Y. The following program implements this query:*

$$\begin{aligned}&\Diamond(!\texttt{coauthors}(\texttt{X},\texttt{Y})) \Rightarrow ?\texttt{book}([\texttt{authors}:\texttt{Z}]), \texttt{X}\in\texttt{Z}, \texttt{Y}\in\texttt{Z}, \texttt{X}\neq\texttt{Y}.\\ &\Diamond(!\texttt{related}(\texttt{X},\texttt{Y})) \Rightarrow !\texttt{coauthors}(\texttt{X},\texttt{Y}).\\ &\Diamond(!\texttt{related}(\texttt{X},\texttt{Y})). \Rightarrow !\texttt{coauthors}(\texttt{X},\texttt{Z}), !\texttt{related}(\texttt{Z},\texttt{Y}).\end{aligned}$$

If we want to know the authors related to Engels we can use the goal:

$$\Rightarrow !\texttt{related}(\texttt{X}, \text{“}\texttt{Engels}\text{”})$$

We assume a sequential execution of rules, that is, triggered rules are executed one at a time. Clearly, it may happen that more than one rule is ready for activation at a given specific time, but only one can be selected and activated. The policy according to which this selection is made influences the semantics of programs. In the literature, different policies have been proposed. For GPR a solution has been adopted consisting in associating priorities to rules which are employed in the selection process (higher priorities served first). Ties on priorities are solved in favor of lesser rules according to their relative position in the program text files. Further ties (that may arise when two instances of the same rule are triggered at the same time) are solved non-deterministically — at the implementation level this simply implies that the rule instance executed first is the one corresponding to the substitution generated first.

A traditional form of active rule corresponds to the implementation of "update propagations" where triggering is determined by an update executed on the database, and the effect is itself an update.

Example 6 *Assume we wish to enforce the referential constraint between authors and books by deleting an author Y of a book X if X is deleted from the database and there exists no other book having Y as one of the authors. We can use the following rules:*

$$\Diamond(!\mathtt{otherbook}(\mathtt{X},\mathtt{W})) \Rightarrow ?\mathtt{book}(\mathtt{Z},[\mathtt{authors}:\mathtt{V}]), \mathtt{X}\in\mathtt{V}, \mathtt{W}\neq\mathtt{Z}.$$
$$\Diamond(-\mathtt{book}(\mathtt{X},[\mathtt{authors}:\mathtt{Y}])) \Rightarrow \mathtt{Z}\in\mathtt{Y}, \mathtt{not}(!\mathtt{otherbook}(\mathtt{Z},\mathtt{X}))-\mathtt{author}(\mathtt{Z}).$$

The execution of a rule can cause the activation of other rules in turn (cascading activation). In this respect, a language can implement different strategies depending on whether the rules are evaluated depth-first or breadth-first. In the first case, if the evaluation of an atom in a goal (or in the body of a rule) triggers a set of rules, these rules are handled first, before other atoms possibly occurring in the same goal are considered. In the latter case the goal is first completely evaluated and then the triggered rules are taken into account. In the literature, these execution strategies are known as *immediate* and *deferred* execution modalities, respectively [17]. Depending on the chosen strategy, condition evaluation and action execution might be done at different times and thus on different database states. Hence, different results might be obtained. The semantics of GPR is defined for both modalities. However, the current release of GPRS implements the immediate execution modality only. The implementation of the deferred modality will be included in a future release of the system.

The same scheme used in the previous examples can be employed to program conditional data manipulations triggered by a generalized query.

Example 7 *Assume we want to modify the* `loan` *class in order to record the loan of a book copy to a certain person. This can be done as follows:*

$$\begin{array}{l}\Diamond(!\texttt{new\_loan}(\texttt{X},\texttt{Y})) \Rightarrow \\ \quad ?\texttt{a\_book}(\texttt{V},[\texttt{code}:\texttt{Y},\texttt{book}:\texttt{W}]), ?\texttt{book}(\texttt{W},[\texttt{avail}:\texttt{M}]), \texttt{M} > 1, \\ \qquad \texttt{today}(\texttt{D}), \texttt{person}(\texttt{P},[\texttt{name}:\texttt{X}]) \\ \quad \bullet + \texttt{loan}([\texttt{pers}:\texttt{P},\texttt{copy}:\texttt{V},\texttt{period}:[\texttt{from}:\texttt{D},\texttt{to}:\texttt{D}+15]]), \\ \qquad *(\texttt{book}(\texttt{W}),[\texttt{avail}:\texttt{D}-1]).\end{array}$$

Note that rule events may also contain object atoms. In this case, the answer is constructed by taking the union of the answers obtained by matching against the database and those obtained from rule activation.

In GPRS it is possible to use "external" triggering, i.e., triggering raised by alerters "outside" the user program. These kinds of triggers are practically relevant for time-dependent rule executions. An example follows.

Example 8 *Assume we want to generate a report associated to all late-comers. This report has to be generated for each day in the week. The following rule is triggered by the generalized event* $\Diamond(!newday())$ *which is raised each time a new day starts according to the system calendar. The generalized query atom* $!write\_on\_report(X)$*, whose definition is not reported for simplicity, simply records the oids corresponding to late-comers.*

$$\Diamond(!\texttt{newday}()) \Rightarrow !\texttt{late}(\texttt{X}), !\texttt{write\_on\_report}(\texttt{X})$$

3 The GPR System

3.1 System architecture

The GPR language described above has been implemented in the prototype GPRS. The architecture of GPRS is structured as shown in Figure 2 and consists of the following main components. The *User Interface (UI)* that accepts user commands specified through a multiple-choice menus (e.g., execute a goal, compile a program, compile a new database scheme, and so on), and outputs results of evaluations, in a rather simple form. The *Schema Parser (SP)* that compiles an *ODM* scheme provided by the user through the UI onto an equivalent Iris scheme. The *Program and Goal Parser (PGP)* that compiles GPR user programs and goals to an internal representation. The *Triggered Rules Handler (TRH)* that both monitors rule triggering and manages triggered rules. The *Rule Executor (RE)* that takes a goal as input and evaluates it using the other modules. The *Query Resolver (QR)* that evaluates object query atoms. The *Update Executor (UE)* that executes updates on the underlying database. The *Deduction Evaluator (DE)* that evaluates subsets of rules corresponding to pure deductions.

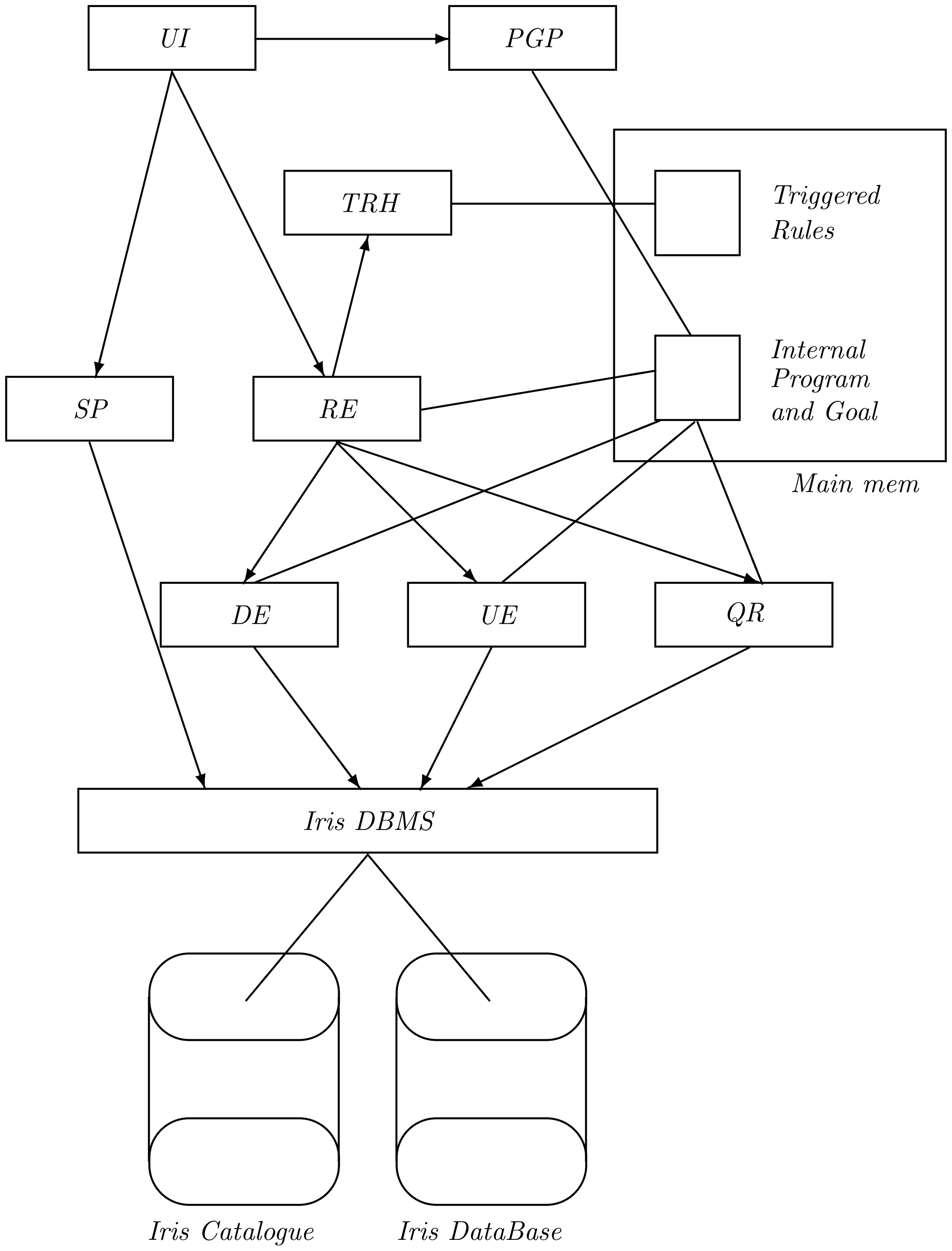

Figure 2: GPRS Prototype Architecture

We now illustrate the above modules in more detail.

3.2 Schema Parser

The purpose of this module is to obtain a translation of an ODM scheme supplied by the user through the UI into an equivalent Iris scheme. ODM schemes are edited off-line and stored in ordinary text files which constitute the input for the SP (that is, the user specifies the name of the file to be compiled). We describe the translation operated by the SP using the ODM example scheme reported in Figure 1.

The translation is carried out in two steps. The first step consists in flattening complex terms. The second one consists in translating flat ODM classes into an Iris scheme. Thus, the first step consists in rewriting ODM user classes into a number of classes, where no nested constructs occur. For each user class, the translation yields one flattened user class, plus a number of system generated classes, which are used to implement structured terms. System generated classes are kept distinct from user ones, and are not visible to the user.

In system generated classes, the outmost type constructor can be either the tuple or the set or the sequence constructor, whereas only the tuple constructor is allowed as the outmost type constructor of user classes.

As an example, consider classes defined in the ODM scheme above. The product of the first step of the translation is shown below.

`author`	=	[name : string, nation : string].
`book`	=	[title : string, authors : `book_0`, total : integer, avail : integer].
`book_0`	=	{`author`}.
`a_book`	=	[code : integer, book : `book`, year: integer].
`loan`	=	[pers : `person`, copy : `a_book`, period: `loan_0`].
`loan_0`	=	[from : integer, to : integer].

Note that, using the rewriting rules, the system is able to manage classes containing any number of nested constructors.

Then, types corresponding to user classes are created as subtypes of a `Usr_Def` system type (see below). Similarly, types corresponding to classes introduced by the system become subtypes of a `Sys_Def` system type (see below). For the example above, the following declarations are produced.

```
create type author subtype of Usr_Def as forward;
create type book subtype of Usr_Def as forward;
create type book_0 subtype of Sys_Def as forward;
create type a_book subtype of Usr_Def as forward;
create type loan subtype of Usr_Def as forward;
create type loan_0 subtype of Sys_Def as forward;
```

Next, we consider the problem of representing class instances. Instances are implemented by defining functions taking object identifiers as the input and returning attribute values. Depending on the outmost constructor associated to the given class, different translations are used.

Case 1, tuple constructor. Attributes of classes where the outmost constructor is the tuple are translated using single-valued functions, whose domain is the same class and whose co-domain is the class type, or the base type, associated with the corresponding attribute. For instance, the attribute `title` in the class `book` is translated into the function `book_title`, whose input argument has type `book` and returns a string. The translation of all the attributes in the class `book` is shown below:

```
create function book_title(book arg key) → Charstring as forward;
create function book_authors(book arg key) → book_0 as forward;
create function book_total(book arg key) → Integer as forward;
create function book_avail(book arg key) → Integer as forward;
create function to_be_deleted(book arg key) → Boolean as forward;
```

where *arg* is a name denoting the formal argument of the function. The additional attribute `to_be_deleted` is a system generated attribute that is used to obtain a correct implementation of triggering based on deletions (see below).

To obtain a better efficiency, the translation of the class `book` is closed with a cluster declaration that groups all the functions we have defined on `book` into a unique data structure, as shown below.

```
cluster book,
   book_title on arg,
   book_authors on arg,
   book_total on arg,
   book_avail on arg,
   to_be_deleted on arg;
```

Case 2, set constructor. Instances of (system-generated) classes whose outmost type constructor is the set (e.g., `book_0` in the example at hand) are obtained using a pair of functions. The first function is many-valued; the second one is single-valued. The many-valued function (e.g., `book_el_0` – see the example below) returns a set of pairs $\langle n, x\rangle$, where n is an integer used for fast identification of set elements, and x denotes an element belonging to the represented set. The single-valued function (e.g., `book_card_0` – see the example below) returns the cardinality of the given set and is automatically updated by the system whenever an element is inserted or deleted.

We continue with the previous example, by showing the representation associated to the class `book_0`:

```
create function book_el_0(book_0 nonkey) → <Integer, author> as stored;
create function book_card_0(book_0 arg key) → Integer as forward;
cluster book_0, book_card_0 on arg;
```

Case 3, sequence constructor. System generated classes where the outmost type constructor is the sequence one have an analogous representation as classes where the outmost constructor is the set. The only difference being the role played by the first argument of the pair returned by the multi-valued function: in this case, it denotes the position of the element into the sequence (and, as such, it has a semantics by itself).

It is worth pointing out here that the translation scheme we have adopted is not the only possible one. This scheme has been preferred because returned types have a uniform structure, independent on the nesting level of ODM user classes, which allowed us to obtain simple implementation algorithms for their manipulation.

Figure 3 shows the structure of a target scheme corresponding to the ODM scheme of Figure 1. In the figure, classes are represented by rectangles. Class attributes, described above, are not reported for simplicity. It can be noted that each object is an instance of the general class `GPRS_Root`. This most general class has three specializations: `Usr_Def` (User–Defined Classes), `Sys_Def` (System–Defined Classes) and `Delete-Log`. User classes appear in this scheme as specializations of the class `Usr_Def`, whereas system defined classes are specializations of the class `Sys_Def`. The special class `Delete-Log` is used to implement deletions of objects from the database and triggering of rules possibly associated to these deletions. A more detailed explanation of the role of the special class `Delete-Log` will be given in Section 3.4.

3.3 Program and Goal Parser

The module PGP is in charge of compiling GPR programs and goals producing corresponding internal representations.

GPR programs are edited off-line by the user and stored in ordinary text files which

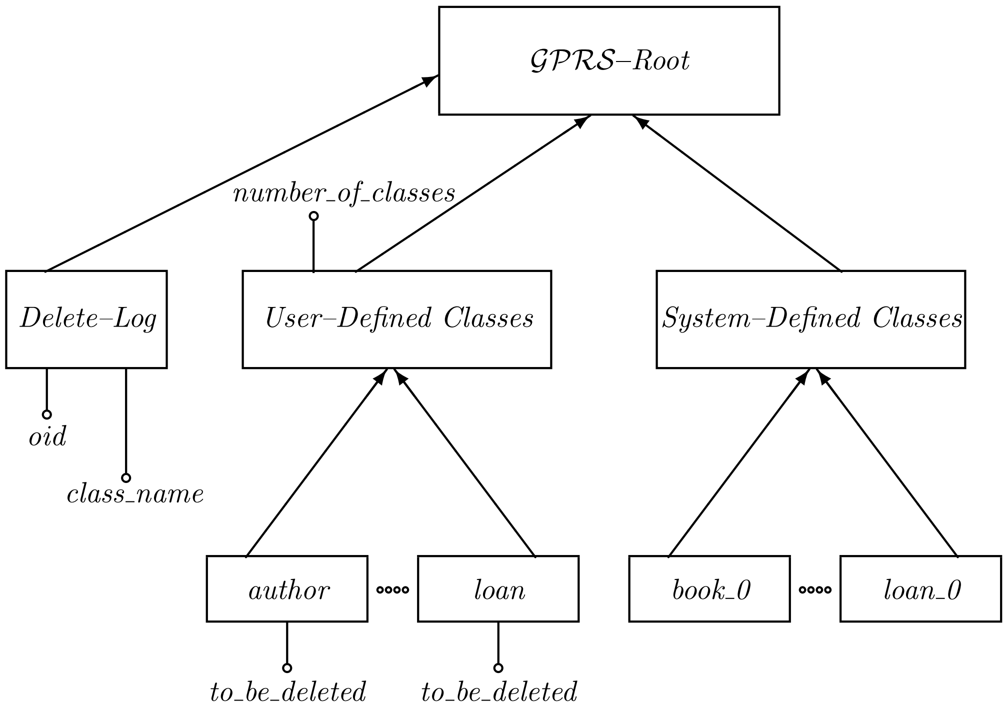

Figure 3: Iris target scheme for an ODM scheme

are then scanned by the PGP. Each rule included in a GPR program must be preceded by the specification of its priority. The priority is specified using a notation of the form (n), where n is a positive integer less than 100. If no priority is specified for a rule, that rule is assumed to have priority 1 by default. GPR goals to be submitted for evaluation can be either edited off-line and stored in text files, or edited on-line and directly evaluated within the current session.

In more details, the PGP works as follows:

1. It checks that rules specified by the user are syntactically correct. If this is not the case, errors are notified. If the rules are syntactically correct, they are stored in main memory into a data structure called *Gen_rule*. *Gen_rule* is a multi-list whose structure closely resembles that of GPR rules.

2. Variables appearing in each Gen_rule structure are substituted by system index variables. This technique allowed us to obtain both variable renaming and efficient access to variables and associated terms within substitutions (see below). For instance, the rule

$$\diamond(!p(X,Y)) \Rightarrow q(X,2)$$

will be represented in the corresponding Gen_rule structure as

$$\diamond(!p(\#1,\#2)) \Rightarrow q(\#1,2).$$

3. Gen_rule structures are then stored into an array, called *Int_prog*. This dynamic array is the structure storing the internal representation of the GPR program at hand. Int_prog is ordered according to the following policy: ordering is lexicographic on the event part atom name; ties are solved in favor of the rules associated with highest priority; further ties are solved in favor of rules occupying lesser positions in the original program text file. Int_prog is used by the module TRH (see below) to single out potentially triggered rules, where a rule is said to be *potentially triggered* if the predicate name specified in the currently considered atom is the same as the predicate name appearing in the rule's event part. Potentially triggered rules are individuated using a binary search algorithm.

4. All rules realizing purely deductive computations are singled out and marked, in order to have them evaluated by the special module DE (see below). A rule is singled out as "purely deductive" if its action part is empty and the action part of

each rule recursively potentially triggered by atoms occurring in its condition part are empty as well.

The process described above yields in Int_prog the "static" representation of the GPR program at hand.

At execution time, an instance of a rule r to be considered for execution is represented by a pair $(\#r,\, \theta)$, where $\#r$ is a reference to the Gen_rule structure representing r and θ is the currently considered substitution instantiating it. Analogously, an instance of an atom a is represented by a pair $(\#a, \theta)$, where $\#a$ denotes a reference to a substructure of the Gen_rule pointed by $\#a$ corresponding to an occurrence of a in r.

Next, we explain how substitutions are represented. Each substitution is implemented using a dynamic array storing pointers to terms. Such an array has n entries, where n is the number of variables occurring in the rule which the substitution is applied to.

Consider a pair $(\#r,\, \theta)$. A term t pointed by the ith entry of the array implementing θ means that the ith variable appearing in r is bound to t by θ[2]. On the contrary, a null pointer in the same position indicates that this variable is free in θ.

The size of the array associated to a substitution may increase during rule evaluation because of the introduction of new variables which may be needed due to the way the unification algorithm works (see, below, Section 3.4.1).

The pair $(\#r,\, \emptyset)$ represents the trivial rule instance, i.e., a rule which no substitution is currently applied to. Instantiation through substitution may be indeed composed. For instance, if the substitution θ_2 is applied to the instance $(\#r,\, \theta_1)$ of the rule r, then $(\#r,\, \theta_1 \circ \theta_2)$ will denote the resulting instance, where $\theta_1 \circ \theta_2$ represents the usual composition between substitutions applied to θ_1 and θ_2 [16].

3.4 Trigger Rules Handler

This module includes all those procedures that control the management of triggered rules. In particular:

1. the procedure Trig that individuates the rules triggered by a given atom;
2. the procedure Unification, that realizes the unification between GPR terms;
3. the procedure Valid, that is in charge of updating the set of triggered rules waiting for execution, by eliminating those rules whose triggering events have been

[2]Recall that variables are represented by system generated indexes.

invalidated[3];

4. the procedures managing the priority queue in which triggered rules waiting for execution are stored.

In more details, the module TRH realizes the following functionalities:

1. Individuating, within the GPR program stored in main memory, all the rules which are potentially triggered by the atom currently under consideration. The present version of the system implements a simple rule search algorithm based on binary search.

2. Singling out, in the set of potentially triggered rules, those rules which are actually triggered and consequently constructing the instantiation of rules' *activation records*, that are the structures embedding triggered rules (for more details about activation records, see, below, Section 3.4.2).

3. Storing activation records in a priority queue (according to priorities specified with rules). Priority queues are stored in main memory.

4. Updating the content of the priority queue so that it stores, in every moment, only rules whose triggering events are valid.

5. Selecting rules from the priority queue, whenever the module RE asks for a new rule to be executed, according to the following policy: rules with highest priority are selected first. Ties on priority are resolved in favor of rules occupying lesser textual position in the user program file. In the case of further ties (that may arise when two instances of the same rule are triggered at the same time) the selection is non-deterministic — at the implementation level this simply implies that the rule instance executed first is the one corresponding to the substitution generated first.

Next, we consider in details the more involved functionalities implemented in the module TRH.

3.4.1 Unification

The unification algorithm is implemented in the procedure UNIFICATION. This procedure takes in input two atom instances $(\#p(t_1, \ldots, t_n), \theta)$ and $(\#p(t'_1, \ldots, t'_n), \theta')$, called

[3] As an example, consider a rule that was triggered by the insertion of an object O, and this object is deleted before the rule is executed.

activating and *activated* atom, respectively; it returns two substitutions θ_1 and θ_2 if the two atoms can be indeed unified; returns failure, otherwise. The substitution θ_1 binds variables from the activating atom to variables, constants or complex terms of either the activated or the activating atom. The substitution θ_2 binds variables from the activated atom to either constants or complex terms of the activating atom or to variables, constants or complex of the activated one.

Before presenting the Unification algorithm implemented in GPRS, we need to define the concepts of equivalent term and substitution associated to an equivalent term.

Thus, let $S_1, \ldots, S_n$ be disjoint sets of variables and $\theta_1, \ldots, \theta_n$ be a set of associated substitutions. Let Y be a variable. The substitution associated to Y is the unique θ_i such that $Y \in S_i$. Now, let X be a variable. Then $Y = E^{S_1,\ldots,S_n}_{\theta_1,\ldots,\theta_n}(X)$ is the term equivalent to X w.r.t. $S_1, \ldots, S_n, \theta_1, \ldots, \theta_n$ and is defined as follows: if a binding of the form X/Z belongs to $\theta_1 \cup \ldots \cup \theta_n$ then $E^{S_1,\ldots,S_n}_{\theta_1,\ldots,\theta_n}(X)$ equals $E^{S_1,\ldots,S_n}_{\theta_1,\ldots,\theta_n}(Z)$; otherwise, $E^{S_1,\ldots,S_n}_{\theta_1,\ldots,\theta_n}(X)$ equals X.

Example 9 *Consider the sets of variables* $S_1 = \{X, Y, Z, W\}$ *and* $S_2 = \{A, B, C\}$ *and the substitutions* $\theta_1 = \{X/Y, Y/A, Z/B\}$ *and* $\theta_2 = \{A/ < B, C >, B/C\}$*. Then we have:*

$E^{S_1,S_2}_{\theta_1,\theta_2}(X) \;=\; E^{S_1,S_2}_{\theta_1,\theta_2}(Y) \;=\; E^{S_1,S_2}_{\theta_1,\theta_2}(A) \;= < B, C >$;
$E^{S_1,S_2}_{\theta_1,\theta_2}(Z) \;=\; E^{S_1,S_2}_{\theta_1,\theta_2}(B) \;=\; E^{S_1,S_2}_{\theta_1,\theta_2}(C) \;=\; C$.
$E^{S_1,S_2}_{\theta_1,\theta_2}(W) = W$;

The substitutions θ_1 and θ_2 resulting from the unification of $(\#p(t_1, \ldots, t_n), \theta)$ and $(\#p(t'_1, \ldots, t'_n), \theta')$ are built by the following algorithm (which is actually a variant of the classical unification algorithm used in logic programming):

1. $\theta_1 := \theta$; $\theta_2 := \theta'$; $i := 1$; let S_1 denote the set of variables appearing in $t_1, \ldots, t_n$ and S_2 denote the set of those appearing in $t'_1, \ldots, t'_n$.

2. for each pair of variables (t_i, t'_i), construct the term t equivalent to t_i and the term t' equivalent to t'_i (w.r.t. S_1, S_2, θ_1 and θ_2) and the associated substitutions (call them θ_t and $\theta_{t'}, respectively$).

3. if t is a variable and t' is a constant or a structure, then if $\theta_t = \theta_1$ then set θ_1 to $\theta_1 \cup \{t/t'\}$ else set θ_2 to $\theta_2 \cup \{t/t'\}$; goto 9;

4. if t' is a variable and t is a constant or a structure, then if $\theta_{t'} = \theta_1$ then set θ_1 to $\theta_1 \cup \{t'/t\}$, else set θ_2 to $\theta_2 \cup \{t'/t\}$; goto 9;

5. if both t and t' are variables, then if $\theta_t = \theta_1$ then set θ_t to $\theta_t \cup \{t/t'\}$, else set $\theta_{t'}$ to $\theta_{t'} \cup \{t'/t\}$; goto 9;

6. if both t and t' are constants then if $t = t'$ then goto 9 else goto 8;

7. if t and t' are structures of the same type, UNIFICATION is recursively called on each pair of elements from t and t' occurring in corresponding positions; in the recursive execution, step 1 is not executed; goto 9;

8. **return** Failure; (I.e., the two input atoms are not unifiable)

9. $i := i + 1$;
   if $i \leq n$ then goto 2 else **return** $\langle \theta_1, \theta_2 \rangle$ (I.e., the unification terminates successfully!).

An example follows.

Example 10 *Consider the atom* $!p(<X,Y>, X, Z, W)$ *and the generalized event:*

$$\Diamond(!p(A, B, <B, A, C>, B))$$

The unification algorithm above can be traced on the corresponding input pair $\langle \#p(<X,Y>, X, Z, W), \#p(A, B, <B, A, C>, B) \rangle$*, as follows:*

$i = 1.$ $\quad \theta_1 = \emptyset$, $\theta_2 = \{A/<X,Y>\}$;

$i = 2.$ $\quad \theta_1 = \{X/B\}$, $\theta_2 = \{A/<X,Y>\}$;

$i = 3.$ $\quad \theta_1 = \{X/B, Z/<B, A, C>\}$, $\theta_2 = \{A/<X,Y>\}$;

$i = 4.$ $\quad \theta_1 = \{X/B, Z/<B, A, C>, W/B\}$, $\theta_2 = \{A/<X,Y>\}$.

No variables of the activating atom can appear in the substitution θ_2. Therefore, in the case that during the construction of substitutions, a variable of the activated atom is bound to a structure from the activating atom in which variables occur, this structure is copied and its variables are substituted with (possibly new) variables associated to the activating atom.

In more details, the following operations are executed (on termination of the Unification algorithm) in order to eliminate from θ_2 all variables occurring also in the activating atom.

1. for each binding Y/S belonging to θ_2, where S is a non-ground complex term from the activating atom, we set θ_2 to $(\theta_2 - \{Y/S\}) \cup \{Y/S'\}$, where S' is a copy of S;

2. let $X_1, \ldots, X_n$ be all the variables occurring both in the activating atom and in S'; first, for each X_i, $1 \leq i \leq n$, the term t_i equivalent to X_i w.r.t S_1, S_2, θ_1 and θ_2 is constructed; then the following steps are executed:

(a) if t_i is a either a complex term or a variable of the activated atom or it is a constant, then t_i is substituted to each occurrence of X_i in S';

(b) if t_i is a complex term occurring in the activating atom, a copy S'' of t_i is created and S'' is substituted to each occurrence of X_i in S'; this algorithm is then recursively called on S'';

(c) if t_i is a variable from activating atom, a new variable Y' is substituted to each occurrence of X_i is S' and we set θ_1 to $\theta_1 \cup \{t_i/Y'\}$.

Optimizations have been implemented to avoid the creation of actual copies of the structures.

Going back to Example 10 above, the application of the algorithm finally produces the following pair of substitutions:

$$\theta_1 = \{X/B, Z/{<}B, A, C{>}, W/B, Y/Y'\}, \theta_2 = \{A/{<}B, Y'{>}\}$$

where Y' is a new variable associated to the activated atom.

3.4.2 Parameters' Passing

After having individuated rules of the program potentially triggered by the current atom, the module TRH singles out those rules which are actually activated, by calling the procedure TRIG that in turn uses the unification algorithm illustrated above.

Those rules correspondent to successful event part unifications with the current activating atom are stored in a priority queue according to their priorities, as specified in the text program. A new priority queue BR is created for each execution of the procedure VERIFY (see below). Each priority queue is realized as an array (one entry for each of the allowed priorities). Each array entry stores in turn a queue of *activation records*. Activation records include the following information:

1. a pointer $\#r'$ to the activated rule;
2. a pair of substitutions (θ_1, θ_2), produced by the unification algorithm (see previous section).

Activation records containing rules with events specifying query atoms are instantiated within the procedure TRIG through calling the procedure UNIFICATION. In this case the actual input parameters of UNIFICATION are:

1. the instance $(\#q, \theta)$ of a query atom q occurring in a rule r, and

2. the trivial instance $(\#h, \emptyset)$ of the atom in the event part h of a rule r' that is potentially triggered by q.

Priority queues are operated upon by either inserting a new activation record or by extracting the highest priority activation record.

An activation record associated to a rule with priority p is inserted in the queue stored at the pth entry of the array. Furthermore, textual ordering in the original program is used to decide precedences between rules with the same priority. For instance, if $r_1, \ldots, r_n$ are all the rules of a GPR program within priority p, in the textual order of appearance in the program file, and both the rule r_i and the rule r_j are activated, where $1 \leq i < j \leq n$, then r_i is inserted in the queue before r_j.

Before the evaluation of the activated rule can take place, parameter passing from the activating rule to the activated one must be realized. This is obtained by applying the substitution θ_2 to the activated rule. Thus, the activation record $(\#r', (\theta_1, \theta_2))$ is extracted from the priority queue. The instance of the activated rule to be evaluated is then $(\#r',\ \theta_2)$.

Since variables occurring in the activated rule cannot be bound to variables occurring in the activating rule (i.e., the rule where the activating atom occurs), these variables can be freely bound by unifications implied in the ensuing execution.

At the end of evaluation of the activated rule a parameter passing takes place in the reverse direction, i.e., from the activated rule towards the activating one. Parameter passing is realized by executing the assignment $\theta_1 := \theta_1 \circ \theta_2$ where we recall that $\theta_1 \circ \theta_2$ denotes the composition of θ_1 and θ_2 [16].

Opposite to unification, the substitution θ_1 yielded by composition must contain no variables from the activated atom. This is guaranteed as follows.

For each variable Y of the activated rule occurring in θ_1: (1) the set $\{X_1, \ldots, X_n\}$ of all variables from the activating rule such that Y is the term equivalent to each X_i, $1 \leq i \leq n$ w.r.t. S_1, S_2, θ_1 and θ_2 is determined; (2) we set θ_1 to $(\theta_1 - \{X_1/t_1, \ldots, X_n/t_n\}) \cup \{X_1/X_n, \ldots, X_{n-1}/X_n\}$, where $t_1, \ldots, t_n$ are the terms bound to $X_1, \ldots, X_n$ by θ_1 (and which are all equivalent to Y, by definition); (3) finally, X_n is substituted to each occurrence of Y in θ_1.

If $n = 0$, a new variable X of the activating rule is created and substituted to each occurrence of Y in θ_1.

3.4.3 Update Triggering

As already pointed out, rules triggered by the update atom currently considered in the procedure Execute are singled out by the procedure Trig. The operations Trig carries out in this case depend on the update atom type. In more details:

insertions: let $u = +C(o_{id}, o_{ex})$ and $\diamond(+C(o'_{id}, o'_{ex}))$ be the event part of a rule r. Then, r is activated by u if both o_{id} unifies with o'_{id} and o_{ex} unifies with o'_{ex}. Moreover, all attributes specified in o'_{ex} must be specified in o_{ex} as well. Conversely, attributes specified in o_{ex} may be not specified in o'_{ex};

deletions: let $u = -C(o_{id}, o_{ex})$ and $\diamond(-C(o'_{id}, o'_{ex}))$ be the event part of a rule r. Then, r is activated by u only if o_{id} unifies with o'_{id}. In this case, however, the unifiability of o_{ex} and o'_{ex} is not sufficient to determine the activation of r. This is because o_{ex} could denote only partially the object value associated to o_{id}. The same holds for o'_{ex}. As an example, consider an update that deletes all employees working in a given department and a rule that is triggered by the deletion of female employees (say, to update some statistical information bunch). This is the reason why, in executing the deletion, the object query atom $?C(o'_{id}, o'_{ex})$ is evaluated first, and its valuation must be non-empty. The unification of terms associated to attributes appearing both in o_{ex} and in o'_{ex} is guaranteed by the preceding unification of o_{id} and o'_{id} and evaluation of the object query atom $?C(o_{id}, o_{ex})$ against the current database instance carried out within the procedure Perform;

modifications: let $u = *(C(o_{id}, o_{ex}), o'_{ex})$ and $\diamond(*(C(o''_{id}, o''_{ex}), o'''_{ex}))$ be the event part of a rule r. Then, r is activated by u only if o_{id} unifies with o''_{id}. As for deletion, however, the unifiability of o_{ex} and o''_{ex} is not sufficient to determine the activation of r. Therefore, the object query atom $?C(o''_{id}, o''_{ex})$ is evaluated on the current database instance and must yield a non-empty valuation. Since the procedure Perform has not yet actually executed the requested modification when Trig is executed (see below), by evaluating this object query atom we verify that the object value associated to o_{id} preceding its modification corresponds to the specification appearing in the rule event part. For the rule to be actually triggered is moreover necessary that the object value associated to o_{id} after that the modification has been executed corresponds to the second component of the specification appearing in the rule event part. To verify this latter condition, the unification algorithm is called on o'_{ex} and o'''_{ex}. In particular, each attribute occurring in o'''_{ex} must occur in o'_{ex} as well. The converse is not needed. Moreover, the second component of the specification appearing in the event part of the rule must correspond to the specified modification pattern. In other words, the rule is activated only if the attributes specified in o'''_{ex} are modified and assume the specified values. For instance, the modify atom

$$*(book(X, [title : \text{“}Amlet\text{”}, total : T]), [total : T + 1])$$

does not trigger a rule with event part:

$$\diamond(*(book(X, [title : \text{“}Amlet\text{”}, avail : A]), [avail : A - 1])$$

since the attribute `avail` of the class `book` was not modified and despite the fact the object on which the modification was performed unifies with that specified in the rule event part.

3.5 Rule Executor

The module RE takes a goal as the input and evaluates it. It uses the modules TRH, QR and UE. Informally speaking, the module RE implements the control structure of the interpreter of the language. The GPR interpreter consists of a main program, called INTERPRETER, and three procedures: VERIFY, EXECUTE and CALCULATE. Next, we will detail the description of the interpreter.

3.5.1 Procedure Interpreter

The main procedure INTERPRETER is reported in Figure 4. This main procedure takes an initial database state and a goal as the input and returns the answer to the submitted goal plus a new state. Answer substitutions are computed by calling the procedure VERIFY. If the computed answer is not empty, then the new state is produced by evaluating the goal action part instantiated with each computed substitution, by calling the procedure EXECUTE.

Next, we illustrate some implementation details concerning the interpreter. BR_{up} and $depth_{up}$ are two global variables of the module RE. BR_{up} plays an analogous role as the variable BR used by the procedure VERIFY. However, differently from the case of BR, here priority queue allocation is explicitly done within RE and therefore, instead of an array realizing a priority queue, here BR_{up} is a stack, where each element is an array realizing a priority queue. In this framework, the variable $depth_{up}$ stores the pointer to the top of the stack BR_{up}. BR_{up} is used to make activation records associated to triggered rules whose event parts specify update atoms all accessible to the procedure VALID. A detailed description of the management of the structure BR_{up} is given in Section 3.5.3.

```
procedure INTERPRETER;
input: a goal $\Rightarrow Q \bullet U$ and a database state $s$;
output: a set $\Theta$ of answer substitutions and an updated database state $s_{out}$;
begin
    $depth_{up}$:=0;
    VERIFY($Q, \Theta$);
    output($\Theta$);
    if $\Theta \neq \emptyset$ then EXECUTE($U\Theta$);
    PERFORM_DELETION_NOW()
end; {INTERPRETER}
```

Figure 4: The GPR Interpreter – Main procedure

3.5.2 Procedure Verify

The procedure VERIFY, shown in Figure 5, takes a query part Q as the input and constructs and returns the associated (possibly empty) set Θ of computed answer substitutions.

The procedure VERIFY is basically based on a top-down evaluation strategy. However, also the bottom-up evaluation strategy is used to evaluate purely deductive rule subsets. All other evaluations (i.e., those involving somehow the execution of update atoms) are carried out top-down.

In the top-down case, the goal to be evaluated is selected according to the left-to-right ordering of goals within the query part Q[4] under consideration. As already said, selection policy for rules is instead that of choosing those associated with higher priorities first and ties on priority are solved in favor of rules occupying a lesser textual position within original program text file. The search tree associated to the computation of answer substitutions is visited depth-first. Pure deductions are evaluated bottom-up using the semi-naive fixpoint, which features a great simplicity coupled with reasonable performances. In any case, answer substitution construction is set-oriented.

The procedure VERIFY considers one atom q from the query part Q at a time and evaluates it according to its characteristics. More precisely:

1. If q is a built-in atom, the set Θ_c of current computed answers is built by calling the procedure CALCULATE. This procedure, implemented in the module RE, includes

[4]The same selection policy is adopted in the procedure EXECUTE that evaluates action parts.

```
procedure VERIFY($Q$ : a condition; var $\Theta$ : a set of answer substitutions);
var $\Theta_c, \Theta_x$: sets of substitutions; $BR$: a priority queue; $fail$: boolean;
begin
    $\Theta := \emptyset$; $BR :=<>$; $fail$:=false;
    while $Q \neq \emptyset$ and not $fail$ do
        $\Theta_c := \emptyset$;
        extract from $Q$ the next atom $q$;
        if $q$ is a built–in atom then CALCULATE($q\Theta$,$\Theta$) and exit;
        if $q$ corresponds to a pure deduction then EVALUATE($q\Theta$,$\Theta$) and exit;
        if $q$ is an object query atom then
            if $| \Theta | \leq 1$ then $\Theta_c :=$ VAL$(q\Theta, s)$ else $\Theta_c := \Theta \circ$ VAL$(q, s)$;
        $BR$:=TRIG$(P, q\Theta)$;
        while $BR \neq \emptyset$ do
            select from $BR$ the rule $E_x \Rightarrow Q_x \bullet U_x$ that comes first
                in the order determined by priorities and textual order;
            VERIFY$(Q_x, \Theta_x)$;
            if $\Theta_x \neq \emptyset$ then begin
                EXECUTE$(U_x\Theta_x)$;
                for $i$:=1 to $depth_{up}$ do
                    $BR_{up}[i]$ :=VALID$(BR_{up}[i], U_x\Theta_x)$;
                $\Theta_c := \Theta_c \cup \Theta_x$;
            endif
        endwhile
        if $\Theta_c = \emptyset$ then $fail$:=true;
        $\Theta := \Theta_c$;
    endwhile
end; {VERIFY}
```

Figure 5: GPR Interpreter: the procedure VERIFY

the code for realizing all built-in predicates.

2. If q is a generalized query atom whose computation is purely deductive, the set Θ_c of current computed answers is built through a call to the procedure Evaluate in the module DE, that evaluates the associated set of rules bottom-up, using a semi-naive fixpoint algorithm.

3. If q is a generalized or a an object query atom then we do the following. If q is an object query atom, it is evaluated against the current database instance. This operation is carried out by calling the procedure Val in the module QR and has the set Θ_c of current computed answers as its result. If q is a generalized query atom, then we set Θ_c to $\emptyset$. Then, in both cases, for each rule $E_x \Rightarrow Q_x \bullet U_x$ triggered by q (such rules are stored in the priority queue BR, which is duplicated for each activation of the procedure Verify, and are singled out by calling the procedure Trig of the module TRH) the associated set of answers Θ_x is computed by recursively calling the procedure Verify. If this recursive call to Verify is successful (i.e., a non-empty set of answers is returned), the procedure Execute is called to evaluate $U_x\theta$, for each answer substitution $\theta \in \Theta_x$[5]. Moreover, through calling the procedure Valid of the module TRH, all rules whose triggering events are invalidated by the execution of update atoms in $U_x\Theta_x$ are deleted from BR_{up}. The set of answers Θ_x is added to the set Θ_c.

   If, after evaluating all involved rules, the set Θ_c is empty, then the "subgoal" denoted by the atom q fails and, consequently, the query part Q fails globally in turn. In this case, the boolean variable *fail* is set to `true`.

On termination of the evaluation of the first atom q, the set Θ_c is assigned to the set Θ of computed answers. If the variable *fail* is set to `true`, the procedure Verify terminates with failure. Otherwise, the next atom from Q is considered. The set Θ_c computed for atoms of Q different from the first one is composed to the global set Θ. When all the atoms from Q have been evaluated, and if the failure flag has not been raised, the procedure Verify terminates returning the set Θ.

3.5.3 Procedure Execute

The procedure Execute is shown in Figure 6. It takes an action part U as the input and modifies the current database state according to the updates specified in U. We recall

[5]Recall that the order of application of substitution θ is non-deterministic.

that the system implements the immediate mode version of GPR semantics, i.e., rules are executed as soon as they are triggered. Next, we illustrate how the procedure EXECUTE works in more details.

The first operation carried out in EXECUTE is to add 1 to the value of the level variable $depth_{up}$. This corresponds to inserting a new element in the stack BR_{up}. After that, an update atom u is chosen and the corresponding operation is (maybe virtually) performed by calling the procedures PERFORM, MARK_OBJECTS_FOR_DELETION and PERFORM_MODIFICATION_NOW of the module UE. Then, rules triggered because of the execution of the update u are singled out by calling the procedure TRIG of the module TRH. Triggered rules are stored in the stack entry $BR_{up}[depth_{up}]$. For each rule triggered by u, say $E_x \Rightarrow Q_x \bullet U_x$ the condition part Q_x is evaluated by further calling the procedure VERIFY and, if no failure is notified, the action part U_x (suitably instantiated by computed answer substitutions yielded from evaluating Q_x) is executed and the current database instance consequently modified by recursively calling the procedure EXECUTE. At the same time, all triggered rules having an update atom specified in their event parts and whose triggering events have been invalidated because of the execution of update atoms from U_x, are deleted from the stack BR_{up}, by calling the procedure VALID of the module TRH. On termination of the evaluation of all atoms in U, the variable $depth_{up}$ is decreased by 1 and the procedure EXECUTE terminates.

3.6 Query Resolver

This module implements the procedure VAL of the interpreter. As such, it builds the set of computed answer substitutions associated to the evaluation of an object query atom. Thus, the QR receives an object query atom either from the module RE or from the module DE and returns its valuation constructed by submitting a suitable O–SQL query to the underlying object oriented DBMS.

The procedure constructing the O–SQL query corresponding to a given object query atom takes the class name and the value term of this atom as the input and returns three lists as the output. These lists are as follows.

1. *S–Part*, is a list of variable names and implements the *select–clause* of the O–SQL query;

2. *F–Part*, is a list of pairs ⟨type, variable name⟩ and implements the *for–each–clause* of the O–SQL query;

3. *W–Part*, is a list denoting a conjunction of conditions (each of which is implemented

```
procedure EXECUTE($U$ : an action part);
var $\Theta$, $\Theta_x$ : sets of computed answer substitutions;
begin
    $depth_{up}$:=$depth_{up}$+1;
    while $U \neq \emptyset$ do
        extract from $U$ the next atom $u$;
        $\Theta_q$ := PERFORM($u$);
        if $\Theta_q \neq \emptyset$ then
            $BR_{up}[depth_{up}]$ :=TRIG($P, u\Theta_q$);
            if $u$ is a delete atom then
                MARK_OBJECTS_FOR_DELETION(class of $u, \Theta_q$);
            if $u$ is a modify atom then
                PERFORM_MODIFICATION_NOW($u\Theta_q$);
            while $BR_{up}[depth_{up}] \neq \emptyset$ do
                select from $BR_{up}[depth_{up}]$ the rule $E_x \Rightarrow Q_x \bullet U_x$ that comes first
                    in the order determined by priorities and textual order;
                VERIFY($Q_x, \Theta_x$);
                if $\Theta_x \neq \emptyset$ then
                    EXECUTE($U_x\Theta_x$);
                    for $i$:=1 to $depth_{up}$ do
                        $BR_{up}[i]$ :=VALID($BR_{up}[i], U_x\Theta_x$);
                endif
            endwhile
        endif
    endwhile
    $depth_{up}$ := $depth_{up} - 1$;
end; {EXECUTE}
```

Figure 6: GPR Interpreter: the procedure EXECUTE

as a function call occurring in the list) realizing the *where–clause* of the O–SQL query

Lists are duplicate-free. The O–SQL query is then simply obtained from the three lists as follows:

```
SELECT S–Part
FOR EACH F–Part
WHERE W–Part;
```

Next, we give examples of GPR queries referring to the example database reported in Figure 1 and the corresponding translation to O–SQL. Assume we want to know all authors from Italy. The following object query atom can be submitted to the system:

$$\Rightarrow ?\texttt{author}([\texttt{name} : \texttt{X}, \texttt{nation} : \text{“}\texttt{Italy}''])$$

The translation algorithm constructs the following O–SQL query:

```
SELECT X
FOR EACH author Z, Charstring X
WHERE to_be_deleted(Z) = False AND
   author_name(Z) = X AND
   author_nation(Z) = ‘‘Italy";
```

The condition `to_be_deleted`(Z) = `False` is added in order to discard objects that have been (implicitly) deleted (see Section 3.7.2).

Assume now we want to know all the titles such that no copy is currently lent. The following object query atom can be submitted to the system.

$$\Rightarrow ?\texttt{book}([\texttt{title} : \texttt{Y}, \texttt{total} : \texttt{Z}, \texttt{avail} : \texttt{Z}])$$

The correspondent O–SQL query is the following:

```
SELECT Y, Z
FOR EACH book X, Charstring Y, Integer Z
WHERE to_be_deleted(X) = False AND
   book_title(X) = Y AND
   book_total(X) = Z AND
   book_avail(X) = Z;
```

With reference to the example database reported in Figure 1, the computed answers would be Y = “The Prince”, Z = 1.

The following query returns the author(s) of the book entitled "Amlet":

$$\Rightarrow ?\texttt{book}([\texttt{title} : \text{“}\texttt{Amlet}\text{”}, \texttt{authors} : \texttt{X}])$$

The correspondent O–SQL query and the answer substitution constructed for the database instance of Figure 1 are as follows:

```
SELECT X
FOR EACH book Z, book_0 X
WHERE to_be_deleted(Z) = False AND
   book_title(Z) = ‘‘Amlet" AND
   book_authors(Z) = X;
```

X = {Shakespeare}.

Note here the the system does not display the actual value bound to the variable X: this value is an oid (in this case, even belonging to a system-generated class) and, as such, is meaningless to the user. Therefore, displayed value is obtained by recursive dereferencing over oid values. Dereferencing is carried out within the procedure Val, on termination of the O–SQL query evaluation.

3.7 Update Executor

The module UE includes the procedures Perform, Mark_Objects_for_Deletion, Perform_Modification_Now and Perform_Deletion_Now, which are illustrated next.

3.7.1 Procedure Perform

The procedure Perform takes an update atom from the module RE as the input and returns a set of answer substitutions defining bindings between a variable and the oid of an object that must be or has been updated and, possibly, a new database state.

Depending on the form of the update atom to be executed, the procedure Perform carries out different manipulations, which are illustrated next.

$+C(o_{id}, o_{ex})$: the operation is executed only if o_{ex} is ground. The oid term can be either ground (e.g., has been previously bound through an object atom valuation – this may happen, for instance, when the same object is to belong to different classes), or a variable, in which case an oid value is invented by the system and assigned to o_{id}. Thus, Perform (possibly) request a new oid to be invented and then stores this oid together with the object extension (as specified by o_{ex}) in the class C. If complex terms are specified in o_{ex}, other objects are possibly created and put in corresponding system generated classes (see Section 3.2). If the oid is invented, the object's class counter (i.e., the value of the attribute *number_of_classes*) is set to 1, otherwise it is incremented by 1.

$-C(o_{id}, o_{ex})$: The first operation carried out by PERFORM in this case is to evaluate the object query atom $?C(o_{id}, o_{ex})$ on the current database instance. The returned objects will have to be deleted from the database. However, they are not immediately deleted, in order to allow a correct handling of triggering based on deletions. Rather they are "marked" as deleted and treated accordingly in following operations. We will detail about the exact mechanism used to implement deletions and modifications in the next section.

$*(C(o_{id}, o_{ex}), o'_{ex})$: Analogously to execution of deletes, the first operation carried out by PERFORM is to request the evaluation of the object query atom $?C(o_{id}, o_{ex})$ on the current database instance. This operation returns the set of objects from C to be modified. Even in this case, the update operations are not immediately executed (see next section).

3.7.2 Execution of delete and modify atoms

As briefly illustrated above, the procedure PERFORM does not execute any explicit deletions or modifications of objects stored in the current database instance. The reason is to allow a correct handling of triggering based on deletions and modifications within the procedure TRIG. The execution of modifications is postponed just slightly. Indeed, actual modifications are carried out by the procedure PERFORM_MODIFICATION_NOW which is executed only at the end of the procedure TRIG.

Differently from modify atoms, the execution of delete atoms needs to be procrastinated further until to the end of entire computation realizing the evaluation of the initial user goal. Indeed, the behavior of the referential integrity constraint mechanism of Iris object manager would make correct handling of triggering based on deletions impossible. This situation is illustrated by the following example. Consider the rule:

$$\diamond(-\mathtt{book(X)}) \Rightarrow \mathtt{true} \bullet -\mathtt{a\_book(Y, [book : X])}.$$

that deletes all book copies if the book itself is removed from the database. If we were to explicit execute the deletion $-$`book(X)` before the triggered rule is executed, the underlying object manager would assign, for each object stored in the database, the null value to all attributes of type `book` that were previously referencing the deleted object. As a consequence, differently from what the rule states, no objects from the class `a_book` would be deleted by executing this rule. Therefore, deletions are handled as follows. On termination of the procedure TRIG, if the currently considered atom is a delete atom, the procedure MARK_OBJECTS_FOR_DELETION is called. This procedure executes the following operations on each object to be deleted (we recall that the set of oids to be deleted is recorded within the procedure PERFORM):

1. the object and the class from which the object has to be deleted are stored in the `Delete-Log` (we recall that an object may belong to more than one class and, as such, a deletion from a class is not to necessarily imply a physical deletion from the database);

2. the object is implicitly deleted by assigning `true` to the attribute `to_be_deleted` of the specified class; note that QR generates object valuations by discarding those objects whose `to_be_deleted` attribute evaluates `true`.

Explicit deletion of objects stored in the `Delete-Log` takes place only on termination of the entire evaluation of the initial user goal, and is realized by the procedure PERFORM_DELETION_NOW called within the main procedure of the INTERPRETER. In more details, for each object specified in the `Delete-Log`:

1. the class counter (implemented in the attribute *number_of_classes*) is decremented by 1;

2. if the object does not belong to other classes, i.e., the value of its *number_of_classes* attribute equals 0, then the object is physically removed by invoking the Iris *delete object* operation, otherwise the object is simply forced to "loose" the class specified in the `Delete-Log` by invoking the Iris *remove type* operation.

3.8 Deduction Evaluator

This module serves the purpose of evaluating atoms corresponding to pure deductions using a bottom-up strategy, rather than using the top-down one. We recall that an atom corresponds to a pure deduction if its evaluation cannot directly or indirectly cause any update to be executed on the current database instance. The rationale for this is that top-down evaluation of rules presents some problems regarding termination of recursive rules, which do not characterize bottom-up strategy. In the presence of updates, due to the inherent procedurality of state transitions associated to them, the use of top-down execution can not be avoided. In the case of pure deduction, instead, the evaluation can be profitably executed bottom-up.

Thus, the module DE implements the procedure Evaluate, which takes the goal corresponding to one such evaluations as the input and evaluates the corresponding set of rules employing a semi-naive fixpoint computation algorithm.

4 Conclusions

In this paper, the GPRS prototype has been presented. The system supports the rule language GPR [19], that integrates, into a unified syntactic and semantic framework deductive and active rules. This characteristics makes the system a powerful tool for the development of database applications based on rules.

The system has been developed in O–SQL with C as the host language on top of Iris, an object oriented database system. The system runs under Unix on workstations HP Apollo 700.

The prototype structure is based on an abstract interpreter that implements the formal semantics of GPR working under the immediate execution modality, as described in [19]. The underlying object-oriented DBMS is used to implement database scheme definitions and basic manipulations on data (retrievals and updates).

At the moment, we are testing the system "on the field" by developing two rule applications. A first, more traditional one, regards the management of loans, acquisition and dismission of books in an academic library. The second one is a (simulated) robot motion planning application. (Simplified versions) of some of the GPR rules used in the former application are reported in this paper.

Several things are still missing in the present version of the system and will be added in the near future. First of all, we are developing a graphical interface based on Motif to substitute the present naive one. Then, we are going to add modules to have GPR run also under the deferred execution mode. Third, we are going to realize the porting of the system onto a commercial relational database management system in order the have the system testable in a wider number of environments. Last but not the least, we are planning to implement tools like termination and confluence checkers, that can ease the work of programmers in rule applications development.

References

[1] S. Abiteboul. Towards a deductive object-oriented database language. *Data and*

Knowledge Engineering, 5:263–287, 1990.

[2] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Comp. and System Sc.*, 43(1):62–124, August 1991.

[3] A Aiken, J. Widom, and J. Hellerstein. Behavior of database production rule: termination, confluence, and observable determinism. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 59–68, 1992.

[4] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object oriented data modeling with a rule-based programming paradigm. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 225–236, 1990.

[5] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Data Bases.* Springer-Verlag, 1989.

[6] S. Ceri, P. Fraternali, S. Paraboschi and L. Tanca. Active rule management in Chimera. In *Active Database Systems.* (S. Ceri and J. Widom, eds) Morgan-Kauffman, 1995, 101-128.

[7] S. Ceri and R. Manthey. Consolidated specification of Chimera, the conceptual interface of Idea. Technical Report IDEA.DD.2P.004, ESPRIT Proj. 6333 Idea, 1993.

[8] S. Chakravarthy and S. Nesson. Making an object-oriented DBMS active: design, implementation, and evaluation of a prototype. In *EDBT'90 (Int. Conf. on Extending Database Technology), Venice, LNCS 416*, Springer-Verlag, 1990.

[9] C. de Maindreville and E. Simon. A production rule approach to deductive databases. In *Fourth IEEE Int. Conference on Data Engineering, Los Angeles*, pages 234–241, 1988.

[10] S. W. Dietrich, S. D. Urban, J. V. Harrison, and A. P. Karadimce. A DOOD RANCH at ASU: integrating active, deductive and Object-Oriented databases. *Data Engineering Bulletin*, December 1992.

[11] E.N. Hanson. An initial report on the design of Ariel: a DBMS with an integrated production rule system. *ACM SIGMOD Record*, 18(3):12–19, 1989.

[12] E.N. Hanson, J. Widom. Rule processing in active database systems. *Int. Journal of Expert Systems*, 6(1):83–119, 1993.

[13] A. P. Karadimce and S. D. Urban. A framework for declarative updates and constraint maintenance in Object-Oriented databases. In *Ninth IEEE Int. Conference on Data Engineering, Vienna*, pages 391–398, 1993.

[14] S. Khoshafian and G. Copeland. Object identity. In *ACM Symp. on Object Oriented Programming Systems, Languages and Applications*, 1986.

[15] J. Kiernan, C. de Maindreville, and E. Simon. Making deductive databases a practical technology: a step forward. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 237–246, 1990.

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

[17] D. McCarty and U. Dayal. The architecture of an active database management system. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 215–224, 1989.

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

[19] L. Palopoli and R. Torlone. Generalized Production Rules as a Basis for Integrating Active and Deductive Databases. *Rapporto R.385, IASI–CNR*, Roma, 1994. An extended abstract of this paper appeared in the proceedings of *Fourth International Workshop on Research Issues in Data Engineering: Active Database Systems (RIDE-ADS'94)*, IEEE Computer Society Press, pag. 30–38, 1994.

[20] T. Sellis, C.C. Lin, and L. Raschid. Coupling production systems and database systems: a homogeneous approach. *IEEE Trans. on Knowledge and Data Eng.*, 5(2):240–256, April 1993.

[21] T. Sellis, C.C. Lin, and L. Raschid. Implementing large production systems in a DBMS environment: concepts and algorithms. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 404–412, 1988.

[22] E. Simon, J. Kiernan, and C. de Maindreville. Implementing high level active rules on the top of a relational DBMS. In *Eighteenth Int. Conf. on Very Large Data Bases, Vancouver*, pages 315–326, 1992.

[23] M. Stonebraker. The integration of rule systems and database systems. *IEEE Trans. on Knowledge and Data Eng.*, 4(5):415–423, October 1992.

[24] M.L. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedure, caching and views in database systems. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 281–290, 1990.

[25] L. Tanca. (Re-)action in deductive databases. In *Second Int. Workshop on Intelligent and Cooperative Information Systems, Como, Italy*, pages 55–61, 1991.

[26] J. Widom. *Deduction in the Starburst Production Rule System.* Report RJ 8135, IBM Research, San Jose, 1991.

[27] J. Widom and S. Finkelstein. Set-Oriented production rules in relational database systems. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 259–270, 1990.

[28] C. Zaniolo. On the unification of active databases and deductive databases. In *Advances in Databases – 11th British National Conference on Databases (BNCOD 11) LNCS 696*, pages 23–39, Springer-Verlag, 1993.