An Overview of the EKS-V1 System

L. Vieille†   P. Bayer   V. Küchenhoff
A. Lefebvre   R. Manthey
e-mail: {petra, volker, al, rainer}@ecrc.de

Abstract
EKS-V1 is a prototype Knowledge Base Management System providing extensive inferential capabilities on large amounts of data. EKS-V1 offers persistent storage of data on secondary storage and allows to derive new information from the data being explicitly stored using a declarative rule language. Integrity constraints can be specified over base and derived relations. These constraints are required to remain true in any state of the database. Integrity of the database is efficiently ensured by the system. The procedural facilities of EKS-V1 allow various forms of basic and sophisticated update of the database. In this paper the structure of EKS-V1 is outlined and examples are given to illustrate its functionalities.

†current address: ENPC-CERMICS, 93 167 Noisy-le-Grand Cedex, France; laurent@enpc.fr

European Computer-Industry
Research Centre GmbH
Arabellastr. 17
D-8000 München 81
Germany
1 Introduction

1.1 What is EKS-V1?

EKS-V1\(^1\) is a Knowledge Base Management System developed along the deductive database approach [GMN84]. It can be seen as a database system providing extensive and efficient inferential capabilities on large amounts of data.

Its inferential capabilities are of the following types:

1. **Deduction of new information** (through deduction rules given by the user) from the data being explicitly stored. The system distinguishes between base predicates, corresponding to data explicitly stored (i.e. the extensional database), and virtual predicates, defined by means of declarative rules (i.e. the intensional database). EKS-V1 offers persistent storage of the extensional database as well as the intensional database.

2. **Maintenance of database integrity.** The user can specify integrity constraints over base and/or derived relations. These constraints are required to remain true in any state of the database, and thus restrict the possible states to those that are actually meant by the database designer.

3. **Post-conditional updates and hypothetical reasoning.** These facilities are high-level tools with which a user can write complex transactions.

For expressing deduction rules and integrity constraints, a declarative logic-based language is offered which allows to express what is desired and not how to do it. Integrity constraints are closed formulae of this language (i.e. yes/no queries), whereas the body of a deduction rule is an arbitrary formula. Technically, the rule language can be seen as an extension of Datalog with negation, existential and universal quantifiers, aggregate functions and external predicates; this rule language supports full recursion. From the point of functionality the declarative language extends the functionalities offered by conventional database query languages e.g., with recursion and negation.

As both deduction rules and integrity constraints are purely declarative statements, the system has to take care of efficiency and termination of the inference processes. The manipulation of data involved in the inference process is done along set-oriented, relational-like techniques.

Post-conditional updates and hypothetical reasoning belong to the procedural facilities provided by EKS-V1. Procedural facilities are required for the development of database applications and of complex transactions. For this purpose, EKS-V1 is accessible from MegaLog (see also [HBD89]), an extended Prolog environment, in which its query and updating facilities are integrated. Further EKS-V1 allows to use external (procedural) predicates within the declarative language.

Updates are possible only within transactions. Transactions can contain an arbitrary sequence of update and query statements. The consistency of the various states of the database with regard to the integrity constraints is ensured: whenever a new constraint is defined, the constraint is checked over the entire database and is accepted only if the current database is consistent with this constraint. On the other hand, if base relations are updated, EKS-V1 checks whether these updates violate the constraints currently defined on the database in an efficient way (see 2.6). The constraint handling and checking relies on the same compiler and evaluator which are already used for query answering.

The remainder of the paper is organized as follows. The introduction is completed with historical notes, objectives and the development of EKS-V1. Section 2 gives examples for the functionalities of EKS-V1. In section 3 the components of the system and their interaction is described. Section 4 compares EKS-V1 to the deductive database systems LDL, CORAL, Aditi, Glue-NAIL.

---

\(^1\) EKS stands for ECRC Knowledge Base Management System.
1.2 Historical Notes

The EKS-V1 project was started in March 1989, as part of an ongoing research effort pursued by the Knowledge Base (KB) group of ECRC since 1984.

The first three years of this research effort were devoted to the study of basic modeling, algorithmic and engineering issues arising when designing a KBMS (Knowledge Base Management System). These issues included: extension of logic-based models with semantic modeling facilities (cf. the KB2 project [Wal86]); development of methods to check the internal consistency of integrity constraints (resulting in the SATCHMO method [MB88]); to check the consistency of the database with the constraints (SOUNDCHECK prototype and later contributions [Dec86, BDM88]) and to efficiently evaluate deductive (recursive) queries (work around QSQ and later contributions [Vie89, Bry90]); integration of logic programming and database technology (EDUCE prototype and later systems [Boc86, HBD89]); design and implementation of multi-dimensional file structures (the BANG file [Fre87, Fre89]).

As a natural follow-up to this first research phase, an integration phase started at the beginning of 1988. The ultimate goal of this second phase was to design a KBMS integrating the various functionalities studied so far. During the year 1988, a considerable amount of work was invested in this integration. This resulted in a proposal for a Knowledge Base Language (KBL [MKW89]) and in a better understanding of the mutual interaction of the various issues and previous results; as an important by-product, the knowledge of these results was no more confined to the mind of the experts, but started to be shared by other people.

As a first step towards a system fulfilling the ultimate goal of this research, it was decided, at the beginning of 1989, to start the design and the development of a system prototype, perhaps with less functionalities than the ones ultimately aimed at, but which had to be running by the end of the year. This short-term project was christened EKS-V1.

1.3 Objectives

The interest of this short-term project was understood as follows. First, it had to validate several solutions and methods produced during the first research phase; such a validation could only be achieved by a system prototype convincingly demonstrating the corresponding functionalities. Second, it was a first attempt to come up with a design of a KBMS integrating at least a subset of the desirable functionalities; as such, it should become a testbed and an important source of observations to be used for later designs. Third, it should demonstrate the interest of a KBMS by hopefully supporting new applications (i.e. applications poorly treated by existing generic systems such as Relational Database Management Systems, and currently requiring ad-hoc developments).

To facilitate the design and the development of EKS-V1, it was agreed that it should rely, as much as possible, on the existing system prototypes: the BANG file system, the MEGALOG platform [HBD89] integrating data base access in a logic programming environment and the DedGi^* deductive query evaluator [LV90]. The BANG file system was providing a convenient storage manager thanks to its multi-dimensional nature; the MEGALOG platform, as a run-time support for EKS-V1, was offering a high-level of integration between a logic programming language and BANG; in particular, its full garbage collecting scheme guaranteed continuous operation when interacting with a database. As for DedGi^*, it was to provide a first version of a deductive query evaluator (limited to recursive Datalog, i.e. without negation nor external predicates), already using the BANG file system; the rule compiler and query evaluator of EKS-V1 were to derive from DedGi^*.

The functionalities to be offered by EKS-V1 were delimited as follows. First, it would provide a declarative, logic-based language allowing quantifiers, negation, disjunction, conjunction and implication as logical connectives; this language should be used both to state integrity constraints and to write the body of derivation rules. Expressions of this language could refer indifferently to base, virtual or external predicates. This declarative language would provide facilities for aggregate operations, in particular recursive aggregates. Second, EKS-V1 was to support basic updating primitives, as well as more sophisticated facilities, such as hypothetical reasoning and conditional updates. EKS-V1 was first to be a single-user system, as the versions of MEGALOG and of BANG available at that time did not allow concurrent
access.

These functionalities were chosen in a pragmatic way, while trying to fulfill the main objectives of EKS-VI stated above. For instance, the full support of integrity by EKS-VI takes advantage of the maturity of the previous work, while providing an unrivaled functionality for complex applications; incorporating quantifiers, a full set of connectives or external predicates in the declarative language was not difficult conceptually and potentially very useful in practice (this is the case in particular for external predicates); the semantics and the evaluation of recursive aggregates required some basic research, justified by their importance for the applications being considered; conditional updates and hypothetical reasoning could be designed in a pragmatic way, while providing a useful tool to write complex transactions.

1.4 Development

The functionalities listed above were developed both by partially redesigning and modifying existing software and by developing new components.

On the one hand, the expressive power (and the interface) of the set-oriented operators provided by BANG was enriched to allow new treatments on the tuples produced by these operators. The rule compiler and query evaluator were extended to support (recursive) aggregate operations, stratified negation, external predicates, the notion of admissible and non-admissible instantiation pattern and evaluation on the old or new states of the database (for evaluation during update propagation); finally, the rule compiler was enabled to call itself recursively, so that compilation could be performed on demand rather than systematically.

On the other hand, new components were designed and developed either as preprocessors to the rule compiler and query evaluator (eg, to handle quantifiers or to generate propagation rules for integrity checking and materialized view handling) or to support new functionalities on their own (fact updates, hypothetical reasoning, maintenance of the rule and constraint base).

The main development phase of EKS-VI ended with the demonstrations made at EDBT'90 (Venice, March 1990) and at SIGMOD (Atlantic City, USA, June 1990). However, development did not stop at this date, as the support of generated predicates, the study of new applications, the integration of the multi-user environment provided by MEGALOG and BANG and finally a thorough debugging have been done since.
2 EKS-V1 Functionalities

In this section, some key functionalities of EKS-V1 are outlined by means of examples. We consider an inventory database, where the following information is explicitly stored (as facts): basic parts; assembly links to make up composite parts; quantity currently held in inventory for a basic part; reliable suppliers.

Extensional database:

basic_part(Part, Supplier, Cost)
assembly(Part, SubPart, Qty)
inventory(Part, Qty_on_hand)
reliable_supplier(Supplier)

2.1 Logic-based Language

The declarative language of EKS-V1 is logic-based: it uses connectives and/or, quantifiers forall/exists, etc. The Prolog convention is used: constants start with a lower-case letter and variables with an upper-case letter. No function symbols are allowed.

As an example, the following rule defines the basic parts whose cost are below a given threshold:

cheap_part(Part, Supplier, Cost, Threshold) <-
basic_part(Part, Supplier, Cost) and
Threshold > Cost.

The query:

?- find cheap_part(tyre, Supplier, Cost, 200).

will return the suppliers of tyres and their selling costs, provided these costs are less than 200.

Such a query is meaningful if and only if the variable Threshold is instantiated when called, as no literal in the body of the rule can potentially give values to Threshold. We say the variable Threshold has no range in the body of the rule. In general, EKS-V1 requires expressions and queries to be range-restricted, i.e. that all variables have either ranges or are instantiated at query-time.

The body of a deduction rule can be any formula of our language (i.e. with and/or/forall/exists quantifiers) and can refer either to base relations, to virtual relations, or to external predicates (i.e. predicates defined by means of a procedure written in a procedural language: Prolog, C\(^2\), ...).

As an example, the predicate manufacturable defined below is true for all composite parts which can be manufactured, i.e. for which one has enough quantity on stock for all its basic subparts (the predicate part_subpart_qty is defined later):

manufacturable(Part) <-
forall [Subpart, Qty]:
( part_subpart_qty(Part, Subpart, Qty)
  and basic_part(Subpart, _, _) )
  -> available_part(Subpart, Qty).

available_part(Part, Qty) <-
inventory(Part, Q_inv) and
Q_inv >= Qty.

\(^2\)Only Prolog predicates are currently supported.
2.2 Aggregate functions

EKS-V1 permits the specification of aggregate functions, i.e. of functions that take as input sets of facts. From the aggregate point of view, EKS-V1 incorporates solutions to many issues that remain problematic in SQL systems (see [KG85]).

Aggregates are specified by giving three elements: a (base/derived) literal literal which specifies the set to be taken as input for the aggregate functions; a list of grouped by variables (a subset of the variables of literal) along which the set of facts will be partitioned for computing the aggregate functions; and the specification of one or several aggregate functions by "VarAgg isagg count" or "VarAgg isagg Function(Var)" where Var is an argument of literal, and Function is some predefined aggregate function. The aggregate operators available in EKS-V1 are min, max, sum, avg and count.

As an example, the predicate min_cost returns the minimal cost for which a given basic part can be purchased:

\[
\text{min\_cost(Part, MinCost) \leftarrow}
\]
\[
\text{agg}(
\text{basic\_part(Part, _, Cost)}
\text{grouped\_by [Part]}
\text{where [ MinCost isagg min(Cost)]})
\]

If the predicate min_cost is called with the argument Part instantiated, then it will return one value (the minimal cost of Part); if Part remains free, then min_cost will remain the binary table of all the basic parts and their minimal cost.

2.3 Recursion, Recursion and Aggregates

EKS-V1 supports any kind of recursive definitions and guarantees termination and completeness of the evaluation of queries involving such recursive definitions. As an example, the virtual relation part\_subpart is defined below as the transitive closure of the assembly relation:

\[
\text{part\_subpart(Part, SubPart) \leftarrow}
\]
\[
\text{assembly(Part, SubPart, _)}. \\
\text{part\_subpart(Part, SubPart) \leftarrow}
\]
\[
\text{assembly(Part, InterPart, _) and}
\text{part\_subpart(InterPart, SubPart)}. 
\]

EKS-V1 accommodates stratified databases, i.e. databases where intermixing of recursion and negation (either explicit negation or negation implicitly present because of quantifiers) is not allowed. This restriction is standard in deductive systems and prevents any fact to be provable from its negation. In EKS-V1, it has been adapted in order to accommodate quantifiers in the body of the rules.

A key aspect of EKS-V1 is to allow recursion and aggregate functions to be interleaved. As an example (the well-known bill of material application), the total cost of a composite part is obtained by summing up the cost of its subparts, taking into account the number of their occurrences. Such a query needs the introduction of aggregate functions to be expressible (bom stands for 'bill-of-materials'):

\[
bom(Part, TotalCost) \leftarrow
\]
\[
\text{agg}(
\text{subpart\_cost(Part, SubPart, SubCost)}
\text{grouped\_by [Part]}
\text{where [TotalCost isagg sum(SubCost)]})
\]
subpart_cost(Part, Part, Cost) <-
  min_cost(Part, Cost).
subpart_cost(Part, SubPart, Cost) <-
  assembly(Part, SubPart, Quantity) and
  bom(SubPart, TotalSubCost) and
  times(Quantity, TotalSubCost, Cost).

Note that the predicate times is an external predicate.

As another example, the following predicate part_subpart_qty associates a Part and its Subparts with the number of occurrences of this Subpart:

part_subpart_qty(Part, Subpart, Qty) <-
  agg
    part_subpart_qty1(Part, Inter_part, Subpart, Inter_Qty)
    grouped_by [Part, Subpart]
    where [Qty is agg sum(Inter_Qty)]
  
part_subpart_qty1(Part, Part, Subpart, Qty) <-
  assembly(Part, Subpart, Qty).

part_subpart_qty1(Part, Inter_part, Subpart, Inter_Qty) <-
  assembly(Part, Inter_part, Qty) and
  part_subpart_qty1(Inter_part, Subpart, Inter_Qty1) and
  times(Inter_Qty, Qty, Inter_Qty1).

In general, recursive aggregates may cause semantic and algorithmic difficulties. The solutions developed for EKS-V1 are outlined in [VBKL90] and presented in more detail in [LeF91].

2.4 Integrity Constraints

Integrity constraints are closed formulae of the language (i.e. all variables must be quantified and given a range). For instance, the database designer may specify the following constraint (for each basic part stored in the database, its suppliers must be reliable):

Constraint 1:

forall [Part, Supplier, Cost]:
  basic_part(Part, Supplier, Cost)
  =>
  reliable_supplier(Supplier).

In this case, the system will reject any update and/or transaction that would result in a state where one basic part has a non reliable supplier.

As another example consider the following constraint (over a recursively defined virtual predicate) which requires that the part_subpart relationship remains acyclic:

Constraint 2:

not exists [X]:
  part_subpart(X,X).

(Note that the above constraint only reflects the semantics of the data: the query evaluator would terminate and return a complete answer even in case of cyclic data).
According to their semantics, constraints are exploited by EKS-V1 on two occasions. When the designer enters a new constraint, EKS-V1 checks whether the current database satisfies this constraint; this is achieved by evaluating the constraint as a yes/no query. When a user updates base relations, EKS-V1 checks whether these updates violate the constraints currently defined on the database. If an integrity violation occurs, then all updates resulting from the transaction are undone and the state of the database before the start of the transaction is restored.

A key aspect for integrity checking efficiency is the following: as constraints are required to hold before each transaction, the check performed at the end of a transaction can concentrate on the changes implied by this transaction. Consider again constraint \( c \) and a simple transaction inserting one fact into the basic_part relation, say basic_part(Leather_seat, BMW, 6000). It is enough to check whether BMW is a reliable supplier: all other suppliers present in the basic_part relation were already known to be reliable suppliers before the transaction. A check needs to be done only on insertion of facts into the basic_part relation and on removal of facts from the reliable_supplier relation.

In general, limiting the check to the data relevant to the updates can lead to considerable gains in performance. In order to achieve this goal, EKS-V1 compiles constraints into propagation rules (see 2.6) that are invoked only when needed. For example, for the constraint of example \( c \), EKS-V1 will generate two rules: the first one takes as input a (set of) tuple(s) inserted into basic_part and will check that there are corresponding facts in the reliable_supplier relation; the second one takes as input a (set of) tuple(s) removed from the reliable_supplier relation and will check that there are no corresponding tuples in the basic_part relation. If none of these rules can be successfully evaluated, no constraint is violated and the update is accepted.

Finally, it is interesting to compare the integrity facility of EKS-V1 to the trigger rules (see for instance [Sto86]) facility sometimes advocated to support integrity enforcement. When using triggers, the user specifies what operations have to be performed on insertion or on deletions of facts. In EKS-V1, given a declarative specification of the constraint, the system generates programs to be evaluated when updating a relation involved in the constraint.

2.5 Generated Predicates

A generated predicate is a predicate which is defined by means of declarative rules and for which the set of answers is materialized in a base relation. The motivation for this approach is the following: if a predicate is often queried and turns out to be expensive to evaluate it might be an advantage to materialize once the set of answers for this predicate. Now further queries simply access the materialized (base) relation. This can result in a non negligible gain in efficiency at query time.

However, additional efforts are now required when updating data. As updates might affect materialized relations (some tuples will be added/deleted implicitly), updates of materialized relations have to be performed by the system. In EKS-V1 a materialized relation is not completely recomputed after an update but added/deleted tuples are incrementally detected by means of update propagation.

As an example of a generated predicate, we model the minimal enclosing box of a polygon. After a transaction, the system recomputes only the minimal enclosing boxes of those polygons that were modified or defined during the transaction. The explicit storage of minimal enclosing boxes is essential for, e.g., inclusion queries: which are the polygons containing a given point? To answer such a query, a range query is first issued to find the enclosing boxes containing the point. This returns a first set of candidate polygons, on which an exact inclusion test is then performed.

If the minimal enclosing boxes were not explicitly stored, then the exact test would have to be performed on every polygon in the database. Several file structures specialized for geometrical applications automatically maintain enclosing boxes (see [Ore90]). Here, this can be specified in a higher-level language:

\[
\text{min_max_box}(\text{Polygon}_\text{Id}, \text{Xmin}, \text{Xmax}, \text{Ymin}, \text{Ymax}) \leftarrow \\
\quad \text{agg}( \\
\quad \quad \text{polygon_points}(\text{Polygon}_\text{Id}, \text{X}, \text{Y}) \\
\quad \quad \text{grouped_by} [\text{Polygon}_\text{Id}] \\
\quad \quad \text{where} \quad [\text{Xmin} \text{ is min}(\text{X}), \text{Xmax} \text{ is max}(\text{X}), \\
\quad \quad \text{Ymin} \text{ is min}(\text{Y}), \text{Ymax} \text{ is max}(\text{Y})] 
\]
2.6 Update Propagation

Integrity checking as well as the maintenance of generated predicates is performed at the end of a transaction. If an integrity violation occurs, then all updates resulting from the transaction are undone and the state of the database before the start of the transaction is restored.

As constraints are required to hold before each transaction and materialized relations are required to be correct before each transaction, the checks performed at the end of a transaction can concentrate on the changes implied by this transaction.

Integrity checking is performed by using the device of propagation rules (see also [VBK91] and [Küc91] when constraints are given to the system, they are rewritten into propagation rules, which are compiled like any other rules. Their evaluation, at the end of a transaction, has the effect of propagating the updates throughout rules and constraints. While this method builds further on recent work on integrity checking in deductive databases [BDM88] [SK87] [LJ87] [DW88], particular emphasis has been put on simplicity (integrity checking is reduced to propagating updates through rules) and on providing solutions for a smooth integration of integrity checking and query evaluation.

An example is now given to illustrate which propagation rules are generated for an integrity constraint and how updates are propagated at update time. Consider again the constraint expressing that the part-subpart relationship should be acyclic. The constraint is rewritten into a denial and has the internal number 2.

Constraint 2 rewritten as denial:

\[ \text{inc}_2 : - \text{part-subpart}(X, X). \]

The task of propagation rules in the context of integrity checking is to determine which updates might add the derivability of inc_2 to the database. Starting top-down from the denial the propagation rules are generated to analyse how derivability proofs can be added. For constraint 2 the following propagation rules are generated:

1: if add(inc_2) then reject_update.
2: if add(part-subpart(X, X)) then add(inc_2).
3: if add(assembly(X, Y, _)) then add(part-subpart(X, Y)).
4: if add(assembly(X, Z, _)) and part-subpart(Z, Y) then add(part-subpart(X, Y).
5: if add(part-subpart(X, Y)) and assembly(Z, X, _) then add(part-subpart(Z, Y).

At the end of a transaction queries are invoked over the set of updated base relations. If one of the queries succeeds, an integrity constraint was violated, the user receives a message which constraint is violated and the transaction is rejected.

If for example the update \(\text{ins}(\text{assembly}(\text{spoke}, \text{bicycle}, 1))\) is performed which introduces a cycle into the bicycle-parts hierarchy (assuming that \text{spoke} is already defined to be a subpart of \text{bicycle}), the query \(\text{add}(\text{assembly}(\text{spoke}, \text{bicycle}, 1))\) is invoked (i.e. rule 3 and rule 4). Rule 4 asks for the subparts of \text{bicycle} and propagates them further as subparts of \text{spoke}. At this point the information is propagated that a \text{spoke} is a subpart of itself. Now rule 2 is activated which checks whether part and subpart are the same. Rule 2 activates rule 1 which gives a message and rejects the transaction. Simply note here that the case of deletion and of negated literals need some refinements but are basically treated along the same approach.

The same approach is applied for the maintenance of generated predicates: for each generated predicate a set of propagation rules is compiled which update the materialized relations for the generated predicates.

---

2Update propagation uses the same query evaluation techniques as normal query evaluation.

3A rejection of the update is also achieved by activating rule 5: rule 3 propagates that a part-subpart \text{bicycle} is a subpart of \text{spoke}. Rule 5 is recursively activated until the information is derived that a \text{bicycle} is a subpart of itself; now rule 2 can be activated.
at the end of a transaction. Thus propagation rules represent a uniform framework for integrity checking and for generated predicates handling. Yet, the coordination of their execution must comply with different requirements:

1. the violation of only one integrity constraint is sufficient for a transaction to be invalid; as soon as such a violation is detected, the remaining propagation rules do not need to be activated;

2. to be complete, the maintenance of generated predicates must activate all possible propagation rules.

As a consequence, the strategy used to coordinate the activation of propagation rules must be chosen with care. EKS-V1 stops as soon as one integrity constraint is violated by an update while for the maintenance of generated predicates the process of update propagation is saturated.

2.7 Procedural environment: conditional and hypothetical updates

The procedural environment extends Prolog by providing some high-level control constructs (such as "foreach ... do ... " or "while ... do ... ") which reduce the need for the notorious cut and fail primitive of Prolog.

EKS-V1 also provides a transaction language which allows the user to specify complex transactions. This is achieved using a solution classical in relational systems, i.e. by providing primitives to start, to commit and/or to abort a transaction within the host language. The database is required to be consistent with the updates only at transaction-commit time.

More interestingly, EKS-V1 provides facilities for conditional and hypothetical updates (in addition to basic update primitives - ins and rem).

Post-conditional updates are updates that are retained if and only if a given condition is fulfilled after these updates. The updates to be retained can be specified by an updating procedure (say Upd_Goal), and the condition to be fulfilled can be, in general, any goal (say Goal).

Post-conditions are specified in EKS-V1 using the construct:

\[
\text{update Upd\_Goal if\_new Goal}
\]

Hypothetical updates are updates which are performed (assumed) only for the evaluation of a given (Prolog) goal. Again, the updates to be assumed can result from the execution of any updating procedure (say Ass\_Goal), and the goal to be evaluated (say Goal) can contain queries and/or updates. Hypothetical updates are specified using the construct:

\[
\ldots, \text{assume Ass\_Goal for Goal,} \ldots
\]

As an example of a conditional update, one may accept a cost increase from a given supplier if and only if there remains another cheap supplier. This is expressed as follows:

\[
\ldots, \text{update change\_cost(snow\_tyre, michelin, 300 )}
\text{if\_new find basic\_part\_less(snow\_tyre, 200),} \ldots
\]

where change\_cost is an updating procedure defined by the following Prolog program:

\[
\text{change\_cost(Part, Supplier, NewCost) :-}
\text{rem\_all basic\_part(Part, Supplier, _OldCost),}
\text{ins basic\_part(Part, Supplier, NewCost).}
\]

and where basic\_part\_less is a virtual relation defined as follows:
basic_part_less(\text{Part}, \text{Limit}) \leftarrow \\
\quad \text{basic_part}(\text{Part}, _, \text{Cost}) \text{ and} \\
\quad \text{Cost} \leq \text{Limit}.

As an example of a \textit{hypothetical update}, one may want to \textit{assume} a cost increase just for evaluating a query in this hypothetical situation (here, we will compute the total cost of a \texttt{bmw.526i}). In EKS-V1, this is specified as follows:

\begin{verbatim}
assume change_cost(snow_tyre, michelin, 300) 
for   find bom(bmw_526i, TotalCost)
\end{verbatim}

The database is updated only for the evaluation of the query. When it is completed, the updates are undone.

As another example of hypothetical reasoning, \texttt{Ass.goal} may increase, by a given percentage, the salary of all employees, and \texttt{Goal} may first compute the company next year deficit and then store the result in a table. The user can then run several cases, corresponding to various hypotheses for the increase rate; at the end of this sequence of runs, no salary would have changed, but a table would contain a summary of the deficits under the various hypotheses.

These various \textit{conditional and hypothetical update primitives can be arbitrarily nested}; this allows writing complex transactions or sequences of hypotheses.
3 Structure of EKS-V1

In the following an overview of the structure of the EKS-V1 system is presented.

In the first section the main components of the system are listed. We distinguish between the functional modules, which implement one important function, or class of functions, and the handlers, the role of which is to support the interaction with the user or application program. In the second (resp.) third section, the design principles of the handlers (resp. functional modules) are discussed with some details.

3.1 The Main Components

It is convenient to distinguish between two kinds of components in EKS-V1:

The functional modules: these modules are functional in the sense that they perform one function or one class of related functions: these functions are of main importance to EKS-V1 and therefore isolated in separated modules. These functional modules are invoked from handlers and/or from other functional modules and return to their callers, usually either by failure or by success. We distinguish:

- the rule and constraint manager which is made up of a rewriting component, a dictionary manager and a propagation rule manager;
- the optimizing rule compiler;
- the query evaluator;
- the storage manager.

The handlers: their role is to control the interaction with the user and the application program:

- the query handler;
- the schema (including rule and constraint) update handler;
- the fact update handler.

As an exception, the compiler, listed here as a functional module, can also be directly called by the user.

3.2 The Functional Modules

The Rule and Constraint Manager

This module includes a number of operations associated with the management of rules and constraints. It is divided in three submodules:

- The Rewriting Submodule
  This module rewrites (general) rules and constraints into rules expressed in an extended Datalog language. Several syntactic checks, local to one expression, are also performed during the rewriting (eg standardization).

  One of the main interests of this rewriting is to simplify the language internally manipulated by EKS-V1: this kernel needs only deal with rules expressed in an extended Datalog.

- The Propagation Rule Manager
  The role of this module is the generation of propagation rules and their maintenance whenever the set of rules and constraints is updated (see [VBK91a]). After their generation, the propagation rules are compiled nearly like any other rule.

  An important aspect of the propagation rule manager is to ensure that the base of propagation rules remains as small as possible. This ensures that only the relevant propagations are performed, i.e. only those which may eventually affect a constraint or a generated predicate.
**Rule and Constraint Dictionary Management**

This component is in charge of maintaining meta-data about rules and constraints and of checking its syntactic correctness.

These meta-data are kept in a rule and constraint dictionary and consist of:

- general information about predicates known to the database:
  * predicate name and arity;
  * whether the predicate is external, recursively defined, defined by an aggregate operation or simply virtual and non-recursive;
  * whether the predicate is derived or generated;
  * whether the predicate is a user predicate or an internal predicate (introduced by the rewriting);
  * attribute types;
- a dependency graph between predicates;
- information about mutual recursion;
- information about the admissible and non-admissible instantiation patterns for each predicate.

The use and meaning of this concept can be described as follows:

External predicates may be called only when some of their arguments are bound: admissible instantiation patterns indicate the configurations for which an external predicate is callable. Similarly, negated body literals require their arguments to be bound at execution. In both cases, if the body of the rules in which these literals appear are not able to provide sufficiently many bindings for the corresponding variables, these bindings must be provided by the queries asked on the head of the rule. As a consequence, constraints on the instantiation patterns of literals negated or built on external predicates potentially induce constraints on the instantiation patterns of virtual predicates.

This information is maintained whenever the rule and constraint base is updated.

**The Optimizing Rule Compiler**

The optimizing rule compiler takes as input the internal rules produced by the rewriting component and expressed in extended Datalog. This extended Datalog accepts (stratified) negation, invocation of external predicates, (recursive) aggregates and evaluation in the old and new states (this notion refers to the states before and after a fact transaction and is needed for evaluation during update propagation).

The compiler is invoked for a given predicate and a given instantiation pattern. It proceeds in two phases:

- **In the first phase**, it tries to find an admissible, optimal order for evaluating the body literals of each of the corresponding rules. An admissible order is an order in which each body literal is called with sufficiently many arguments instantiated, i.e. is called along one of its admissible patterns. The optimal (admissible) order is determined using both syntactic and (simple) statistical criteria.

  During this first phase, the compiler may have to call itself recursively. When investigating a particular order, the compiler may consider a virtual body literal for which it is not known whether the instantiation pattern induced by this order is admissible. In this case, the compiler is recursively called with this body literal and this pattern as arguments: this recursive call succeeds if and only if the pattern is admissible. A loop checking mechanism is implemented so that the compiler does not loop in case of recursive rules.

  Also, this first phase incorporates the detection of several optimizations for recursive predicates (tail-recursion optimization, syntactic non-admissibility).

  The first phase produces an internal data structure called a data-flow sequence, used as input for the second phase.
The second phase of the compiler is a code generation phase and is conceptually rather straightforward.

It takes as input the data-flow sequences and generates sequences of data manipulation operators (essentially the join, difference and selection operators provided by the BANG system).

The Query Evaluator

The role of the query evaluator is to generate answers to queries by interpreting the code produced by the compiler. These queries can either come from the query handler (primitives find and display.all) or be issued by other components of the system (evaluation of new constraints to check the consistency of the database w.r.t. these new constraints; propagation of fact updates to check the consistency of these updates with existing constraints, and to maintain generated predicates).

The main task of the query evaluator can be understood as coordination: the compiler has produced sequences of operators which, when linked together, make up a (potentially cyclic) evaluation graph; it remains to execute and coordinate the operations specified in this graph.

The main aspects of this coordination task are the following ones:

- Decide about and apply a strategy to execute the sequences of operators corresponding to the different rules defining a predicate.
  
  This strategy can be depth-first (execute one rule and propagate the answers at once - later backtrack to execute the remaining rules), or breadth-first (execute all the rules before propagating the answers).

  The strategy currently implemented is breadth-first.

- Coordinate the operations for negated (resp. aggregate) literals.

  Indeed, queries over these predicates must be completely answered before their answers are used in the corresponding difference operation (resp. in an aggregate operation). This is required for the sake of soundness.

  The breadth-first strategy used in EKS-V1 fits well with this requirement.

- Coordinate the evaluation of recursive predicates and of their respective cliques.

  The completeness and the termination of the evaluation requires that, for recursive predicates, it saturates sets of subqueries and sets of answers. In this respect, EKS-V1 implements a delta mechanism. Further, in order not to violate the above requirement for negation and aggregates, the evaluator sets up clique environments (in a stack-wise manner), so that a clique is completely answered before the evaluator returns to its parent environment.

- Handle non-stratified aggregates.

  The evaluation of recursive (or non-stratified) aggregates is complex and requires special treatment (see [Lef91] and [VBK+91b]).

As a key feature, all manipulations of data are done by means of set-oriented operators: no data are introduced in the Prolog environment until the end of the evaluation. The output of the query evaluator is a BANG temporary relation containing (all) the answers to the query. This temporary relation is then used by the component calling the query evaluator.

The Storage Manager

The storage manager provides the adequate support for storage, manipulation and update of information kept within an EKS-V1 database. This information consists both in the facts, rules and constraints given by the user and in all meta-data and internal data structures generated and maintained by EKS-V1.

A uniform interface is provided by MEGALOG to the BANG file system in order to store flat tuples as well as complex structures. In particular, EKS-V1 inherits from MEGALOG the primitives to interact
with a BANG file on tuple-at-a-time basis (insertion, deletion, backtrackable retrieval) and primitives to 
perform simple relational operations (join/difference/selection). Similarly, the primitives provided by the 
MEGALOG interface to manipulate the schema of a BANG database are used within EKS-V1.

However, the interface to the BANG file system used in EKS-V1 goes beyond the interface provided by 
the MEGALOG system: the interface to the set-oriented operators has been extended so as to be able 
to specify complex operations on the tuples produced by the operators. Indeed, in addition to simply 
inserting the resulting tuples into temporary relations (basic features), one may want, for instance: 1) to 
perform aggregate functions, 2) to evaluate external predicates, 3) to perform the operations related to 
the QSQ algorithm.

3.3 The Handlers

The Query Handler

The query handler takes queries as input and returns answers to the user or the application program.

Queries are literals built over base or virtual predicates. Two query primitives, both taking a query as 
argument, are supported:

- the `find` primitive returns the answers one-at-a-time to the Prolog environment;
- the `display_all` primitive displays (all) the answers on the screen.

The query handler interacts with other modules in the following way:

- For queries over base relations, the query handler generates and executes a call to the storage 
  manager; this call performs a (set-oriented) selection.
- For queries over virtual relations, the query handler may have to call the rule compiler if no compiled 
  code is available for this query. Afterwards, the query handler calls the query evaluator.

From the user (or application program) point of view, a key aspect of query evaluation is that all answers 
to a query are computed before the first answer is returned (these answers are stored in a temporary 
BANG relation). Hence, a (Prolog) program can safely issue a query (using the `find` primitive), and then 
perform updates over base relations while backtracking over the set of answers: the system guarantees 
that these updates will have no side-effect on the result of the query issued previously.

The Fact Update Handler

The fact update handler provides the user (or application program) with facilities to update the data 
stored in base predicates.

The fact update handler provides two kinds of update primitives:

- basic primitives to insert and delete facts (ins, rem);
- more sophisticated primitives for hypothetical reasoning (assume ... for ...) and conditional updates 
  (update ... if new ...).

The fact update handler supports the concept of a fact transaction and provides the associated primitives. 
A fact transaction is either explicitly delimited (primitives begin, commit, abort.fact.transaction or fact-
transaction(Goal)), or implicitly delimited (any update statement delimits a fact transaction if it is 
not within a fact transaction). A fact transaction can be any Prolog program containing (fact) update 
statements (both basic and sophisticated) and queries. Hence, the fact update handler may have to 
interact with the query handler.
The main notion currently associated with a fact transaction is that of *integrity*: at the end of any fact transaction, the database is required to be consistent with the constraints specified by the user; however, inconsistent states are allowed *during* the transaction.

The Schema (Rule and Constraint) Update Handler

The schema update handler provides the user (or application program) with facilities to update the rule scheme (see [VBK+91b]) of the database. In general, rules and constraints are considered as part of the rule scheme, as well as the meta data of the rule and constraint dictionary.

The schema update handler provides primitives:

- to insert and delete *rules and constraints*,
- to declare and undeclare external predicates, the types of their arguments and their admissible invocation patterns;
- to declare virtual predicates as derived or as generated;
- to declare the types and the admissible invocation patterns of virtual predicates.

The schema update handler supports the concept of a *schema transaction* and provides the associated primitives. A *schema transaction* is either explicitly delimited (primitives *begin, commit, abort_expr_transaction* or *expr_transaction*(Goal)), or implicitly delimited (any update statement delimits a schema transaction if it is not within a schema transaction). A schema transaction can be any Prolog program containing update statements and declarations. Unlike a fact transaction, it may not contain queries.

The following tasks are performed whenever the rule and constraint base is updated (this is done when committing the embedding schema transaction):

- Global syntax checks, such as stratification, absence of name clash, etc ...
- Maintenance of the dependency graph and of the information about mutual recursion.
- Derivation of types for attributes of virtual predicates; verification of their compatibility both intra- and inter-rules.
- For *updated* virtual predicates, the patterns previously known to be admissible are required to remain admissible; this is checked by recompiling each pattern previously known to be admissible.
- *Newly* introduced virtual predicates are required to have at least one admissible pattern; this is checked by compiling new virtual predicate for the instantiation pattern bb..bb.

If one of these tasks fails, then the whole schema transaction is aborted.

At the end of a schema transaction, the database is required to be consistent with the constraints specified by the user. Hence, in order to validate a schema transaction (at *commit-time*) every new constraint is evaluated as a YES/NO query over the database. If one of these queries fails, the transaction is aborted and the updates are undone (see [VBK+91b] and [HBD89]).

15
4 Comparison with other Systems

In this section EKS-V1 is briefly compared with LDL (see [NT89]), Aditi (see [VRK+90]), CORAL (see [RBSS90]) and Glue-NAIL (see [PG91] and [Phi90]).

The language interface of all these systems is Datalog, extended with stratified negation and aggregate operators, and embedded in a procedural environment. Although EKS-V1 offers a rule- and query language involving disjunction and quantifiers, the system translates these constructs into extended Datalog as well, thus offering just a more elaborate syntax but no additional expressive power. Additional features like complex terms, lists or sets are provided by LDL, CORAL, Aditi and Glue-NAIL, but not by EKS-V1.

For the evaluation of Datalog rules two approaches are usually distinguished: the 'bottom-up method' optimized with Magic Sets (see eg, [BR87]) and the 'top-down method' optimized with memoization techniques (see eg, QSQ, [Vie88] or OLDT, [TS86] or [DW87]). Both methods generate the same intermediate data structures (subquery and answer relations). In [Bry90] a particularly clear explanation of the relationship between the different approaches is presented. From this point of view all the systems perform a set-oriented top-down evaluation of the initial user rules. LDL, Aditi, CORAL and Glue-NAIL implement this top-down evaluation via a transformation of the initial user rules into modified and additional rules along the Magic Sets approach which are then evaluated bottom-up (via semi-naive iteration). The rewritten rules (extended Datalog) express the top-down propagation of subqueries and ensure that only those subqueries and answers are generated which are relevant for the query. These modified rules are first transformed into relational algebra expressions which are then evaluated bottom-up. In EKS-V1 the Datalog rules are directly translated into relational algebra and the evaluation is performed along the QSQ approach. However the algebra expressions generated by EKS-V1 are very similar to the rewritten rules generated by the other systems.

EKS-V1 maintains an execution time control structure (called a nodes array) by means of which a particular strategy of semi-naive evaluation is implemented. The implementation details of the corresponding semi-naive algorithms in the other systems are not described in details.

Some of the systems offer a more or less elaborate module system. This is not supported by EKS-V1. On the other hand none of the other systems supports integrity checking, generated predicates or sophisticated update primitives. Further EKS-V1 is storing facts and rules persistently while languages like LDL or CORAL have up till now been implemented by main-memory systems only.

LDL

LDL, developed at MCC in Austin (see [NT89]), tries to emphasize declarativity, nevertheless it is admitted that procedural notions are necessary. To maintain the declarative philosophy a declarative semantics is given to procedural parts. As there is no syntactic separation between declarative and procedural parts, updates can be used in the body of declarative rules. LDL also supports a non-deterministic choice operator, which none of the other systems does.

Aditi

Aditi (see [VRK+90]) is a disk-based deductive database system developed at the University of Melbourne. The Aditi project aims to investigate what implementation methods and optimization techniques make a deductive database system competitive with current commercial relational database systems. The structure of the Aditi prototype is based on a client-server model: users interact with a front-end process that is regarded as a client of the system. The client communicates with a back-end process (server) that performs the usual set of database operations. The back-end of Aditi uses relational technology, but the system also employs several optimizations specific to deductive databases, eg, differential evaluation, magic set transformation and constraint propagation. Thus, Aditi offers an evaluation scheme which is more flexible than the fixed evaluation scheme of EKS-V1. Performances of Aditi and EKS-V1 are comparable.
CORAL

CORAL (see [RBSS90]) is a database programming language developed at the University of Wisconsin. CORAL offers a powerful and complex module structure: the modules have a compile-time semantics as well as a run-time semantics. The imperative and declarative constructs of CORAL are separated via the module structure: a module is either declarative or imperative. The procedural language is currently C++ extended with some new types (relations and tuples) and constructs; updates are possible only in imperative modules. The module definition allows to import and export predicates from and to other modules but modules are not allowed to call themselves recursively.

For the evaluation of the declarative modules a variety of optimizations are planned, e.g., Magic set rewriting, factoring, existential query optimization, semi-naive evaluation. The first prototype of CORAL was supposed to be a main-memory implementation; the imperative modules have not yet been implemented in this prototype.

Glue-NAIL

Glue-NAIL (see [PG91] and [Phi90]) is a deductive database system developed at Stanford University which combines declarative and procedural aspects. Glue is a procedural language for deductive databases and is designed to complement the purely declarative language NAIL. Glue was designed to be as similar to NAIL as possible to avoid the impedance mismatch. Therefore both languages have a similar syntax and allow tuples and relations as basic data types. As NAIL code is compiled into Glue code, only Glue code has to be optimized. In contrast to LDL and CORAL, where sets are different from relations, Glue-NAIL uses normal predicates in attributes to define sets. Updates, input and output are possible only within Glue procedures. In Glue-NAIL modules can be defined, but these modules are only a compile-time concept, they do not have a run-time semantics. Glue-NAIL is mainly implemented in Prolog, some parts are written in C. An efficient relational back end still has to be designed for Glue.

5 Conclusion

We believe that EKS-V1 has convincingly validated the idea of knowledge base systems. In particular, to our knowledge, it has been the first system to provide such a range of functionalities, including full support of integrity checking, recursive aggregates, hypothetical reasoning, recursion. Also we feel that a number of features of its architecture, design decisions and implementation techniques are of interest for the design of future KBMS products. Its power has been demonstrated in particular on the bill-of-material application: generalized integrity checking (the acyclicity constraint); recursion (part-subpart hierarchy); recursive aggregates (total cost of a composite part); hypothetical reasoning (total cost of a car, if the cost of a tyre were raised by 10 percent). Many points would require further study: support of richer data modeling facilities (set-valued attribute, semantic modeling, etc); more sophisticated optimization criteria; better code optimization; integration of data structures other than BANG (B-trees, hash); more flexible coordination strategies (dynamic choice between depth-first and breadth-first); and so on.

Acknowledgments

EKS-V1 does not only result from the work of the EKS-V1 team (Petra Bayer, Volker Küchenhoff, Alexandre Lefebvre and Laurent Vieille; under the responsibility of Rainer Manthey since August 1990; with the technical help and support of Michael Dahmen), but from a research effort collectively undertaken by the KB group, headed by Jean-Marie Nicolas (by Mike Freeston since September 1990). The members of the EKS-V1 team thank their colleagues of the KB group for their technical and human support.
Bibliography


