Journal of Intelligent Information Systems, 1, 271–292 (1992)
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Intelligent Databases: Old Challenges and New Opportunities

CARLO ZANIOLO
Computer Science Department, University of California, Los Angeles, CA 90024

Abstract. The evolution of existing information systems and a new wave of data-intensive applications are creating a strong demand for database-centered programming environments much more sophisticated and intelligent than those supported by current database systems. In this paper, we describe the contributions that deductive databases offer to the evolution of databases and information systems to satisfy said demands. In addition to all database essentials, deductive databases support rule-based logic-oriented languages that allow terse formulations of complete applications, along with reasoning and queries. Thus, they support a rule-based interface that eliminates the impedance mismatch problem (between programming language and query sublanguage) and elevates the design and development of database applications to the level of declarative, knowledge-based specifications. In this paper, we review the evolution of the enabling technology and architectures of deductive database prototypes; then we focus on their applications, as seen by the author through his experience with the $\mathcal{LDL}/\mathcal{LDL}$++ project. In particular, the paper describes the languages and the (bottom-up) execution technology used by the first generation of deductive database prototypes. Then the paper discusses how the experience with a first-generation system ($\mathcal{LDL}$) guided the design and implementation of a second-generation prototype ($\mathcal{LDL}$++).

Keywords: data dredging, data log, deductive databases, enterprise schemas, intelligent information systems, non-monotonic reasoning, rule-based systems

1. Introduction

The ever-increasing complexity of information systems and a new wave of data-intensive applications demand computing environments that are much more advanced than those supported by current database management systems (DBMS) (Silverschatz, et al., 1990). Two serious manifestations of the inadequacy of current commercial technology are as follows:

1. The *impedance mismatch* problem. This long-recognized problem stems from the mismatch between the computation models of the DBMS' query sublanguages and those of procedural programming languages (such as COBOL or C) in which these sublanguages are embedded. For instance, the data types used by DBMS are often different from those of programming languages; moreover the set-oriented, declarative semantics of relational DBMS' query languages, such as SQL, represents a substantial departure from the traditional imperative semantics of procedural programming languages. Unfortunately, due to the lack of expressive power of commercial query languages, such as SQL, all but the most trivial database applications must now be written as large procedural programs

with intermingled DBMS query statements. Now, the mismatch between the host procedural language and the embedded query language complicate the task of designing, writing and maintaining information systems, and also reduces the run-time performance of applications.

2. The *new wave of database applications* problem (Silverschatz, et al., 1990). In the past, the problems of SQL were circumscribed within specialized domains, such as the bill-of-materials applications that require the computation of transitive closures. But SQL's limitations are becoming more difficult to live with, as databases are now needed in new application domains, and new services are needed in more traditional domains. For instance, relational DBMS cannot manage or support well databases of scientific information, multimedia data, geographical data, and data describing complex engineering structures. Even in more traditional application realms, DBMS are now faced with the escalating expectations posed by very intelligent applications, which, in terms of sophistication levels and turnaround times required, represent a striking departure from the slow-changing routines of yesterday's data processing.

The new demands posed on traditional database systems are driven by the need of companies to become more competitive by continuously reviewing their production and marketing data to ensure time-to-market production, reduce costs and promote sales. The urgency of these changes is illustrated by what is happening in the more competitive industries. Consider, for instance, the fiercely competitive airline industry and their aggressive approach to "yield management" using their reservation systems, such as Sabre (American Airlines) and Apollo (United Airlines). Quoting Hopper (1990), yield management is the "process of establishing different prices for seats and allocating seats to maximize revenues. Computers review historical booking patterns to forecast demand for flights, monitor bookings, compare fares to competitors' fare, and otherwise assists dozens of flight analysts and operational researchers."

A cutting-edge information technology such as American Airlines' Sabre is not without risks: travel agents have started using a package developed by Associated Travel Management to "scan through a reservation system and snare customer-pleasing bargains that an agent might never spot" (Gellene, 1991). As this program basically circumvents their fare structures and yield management policy (by fare-breaking, selecting alternative airports and routings, etc.), airlines have now modified their reservation systems to detect, and penalize with hefty fees, reservation sessions with high-hit rates (a hit is a basic query, and the typical number of hits per session for a program is four times that of an agent). Of course, we have not yet seen the last chapter of this story, inasmuch as Associated Travel Management is currently developing a "stealth" version of the program, where the number of hits is reduced through intelligent browsing and memorization (Gellene, 1991).

The airlines' tale of measures and countermeasures is symptomatic of a corporate world where the information system becomes an enterprise's most critical tool for surviving and succeeding in business. As result of these changes, corporate information systems and their supporting database-centered computing environments are also evolving rapidly, moving forward in the following two directions:

- A shift toward technologies suitable for delivery more flexible, intelligent services and applications in a more timely fashion. Examples of this trend are represented by the use of sophisticated expert shells, decision support systems, and geographical information systems.
- A move toward streamlining and integrating all the information services of a company. This leads to the conceptual abstraction of an *enterprise schema*, as the vehicle for organizing and using critical knowledge about an enterprise's entities, operations, business rules, and their supporting data and application resources. In addition to supply the basis for a corporate data dictionary, the enterprise schema can then become an operational layer that ensures interoperability among the enterprise's databases and isolates applications from the underlying database systems.

As described in the next section, much of current database research is motivated by the need to solve the impedance mismatch problem and provide computing environments more supportive of advanced database applications.

2. Old problems and new opportunities

The impedance mismatch problem is deep-rooted in cultural and technological issues that have prevented the transfer to the commercial world of early research results aiming at solving the problem. Among the early attempts by database researchers, we find RIGEL (Rowe and Shoens, 1979) and Pascal-R (Schmidt, 1977), which focused on a better fusion of procedural languages and relational systems, and thus inherited the limitations of the relational data model. There were also several attempts from the programming language camp, which focused primarily on adding persistency to existing languages, frequently glossing over other issues such as concurrency and shareability of data. Here, we will only discuss briefly the attempts made of extending Prolog into a database language, which are directly relevant to the topic of our paper, for the following two reasons.

- Prolog is a member of the generation of powerful rule-based languages that commanded great attention the 1980s for their ability of supporting rapid development of sophisticated applications through knowledge-based programming.
- Prolog builds on the theoretical underpinning of Horn-clause logic. Horn clauses are near relatives of database query languages (as per the textbook exercises of mapping a nonrecursive Datalog program into domain calculus, tuple calculus, and relational query languages such as SQL (Ullman, 1989).

This similarities, and the realization that Prolog represents a query language much more powerful than SQL (Zaniolo, 1984), enticed several researchers into building intelligent database systems by coupling Prolog with relational DBMS or by extending Prolog with database capabilities (Ceri, et al., 1989). While these experiments have been successful in producing powerful systems, they have also revealed several problems that stand in the way of a complete integration. Many of these problems are the same as those experienced by the early works on persistent programming languages: it is difficult to add to a language and its compiler mechanisms to support DBMS' essential functions such as

- Persistency
- Data shareability
- Concurrency control
- Integrity and recovery

But, in addition to the challenges faced by persistent languages, Prolog inherits special problems from its semantics and enabling technology. For instance, the cornerstone of Prolog is SLD-resolution according to a left-to-right, depth-first execution order. This powerful mechanism provides the basis for an efficient implementation of Horn-clause logic, and an operational semantics for the constructs without a logic-based semantics that were added to the language for expressive power—e.g, updates, cuts and metalevel primitives. But the dependence of Prolog, and its enabling technology, on SLD-resolution presents serious drawbacks from a database viewpoint inasmuch as:

- Prolog rigid execution model corresponds to a navigational query execution strategy; thus, it compromises data independence and query optimization that are based on the nonnavigational semantics of relational query languages.
- This rigid SLD-based semantics is incompatible with several relational database concepts—in particular with the notions of database updates and transactions. For instance, in the style of many AI systems, Prolog update constructs (i.e., assert and retract) are powerful but unruly, inasmuch as they can modify both the data and the program. Furthermore, none of the nine different semantics for updates in Prolog proposed so far (Moss, 1986) are compatible with that of the relational data model. Indeed, the snapshot-based semantics of relational databases is incompatible with Prolog's execution model, which is instead oriented toward pipelined execution (Krishnamurthy, 1989). Supporting the notion of transactions, which is totally alien to Prolog, compounds these problems.
- The efficiency of Prolog's execution model is predicated upon the use of main memory. Indeed, all current Prolog implementations (Warren, 1983) rely on pointers, stacks, and full unification algorithms, which are not well suited for an implementation based on secondary store.

Because of the problems just mentioned, many of the deductive database projects, started in the mid -1980s, such as $\mathcal{LDL}$, *Nail!* and *Lola*, chose to support Horn clauses and logic rules using extensions of relational database technology. (Chimenti, et al., 1990; Morris, et al., 1987; Schmidt, et al., 1989). These projects thus abandoned SLD-resolution in favor of a bottom-up (fixpoint-based) approach (Chimenti, et al., 1990; Morris, et al., 1987; Schmidt, et al., 1989).

Deductive databases represent the first major research trend of the 1980s in the area of databases and programming languages. This research is motivated by the desire of providing an intelligent database environment that builds on a rule-based language capable of expressing advanced applications as well as complex queries and reasoning on databases—the impedance-mismatch problem is solved, since there is no longer a need for using two languages in developing an application.

The second important development of the 1980s is represented by object-oriented (O-O) databases. Each O-O DBMS uses the powerful data structuring and abstraction capabilities of some object-oriented programming language to build its schema definition facility and data manipulation language; thus, such a system avoids the limitations of the relational data model and any impedance mismatch toward the programming language in which it is embedded. However, O-O DBMS often lack powerful query capabilities and they are basically dependent on the data types offered by a specific O-O language; thus they can viewed as being embedded in the programming environment of such a procedural language, in which the application development is expected to take place.

Therefore O-O DBMS focus on creating an homogeneous environment based on a closely coupled procedural O-O language and database system, while deductive databases focus on elevating the computing environment through declarative logic-based rule languages. In this paper we discuss the deductive database approach to intelligent databases, which, by elevating database programming to the more abstract level of rules and knowledge-based programming, creates an environment more supportive of the new wave of advanced database applications. However, a deductive approach does not preclude object orientation, and several research efforts are currently devoted at combining these two paradigms (Kifer and Wu, 1989; Zaniolo, 1989).

3. Deductive databases

After generating significant theoretical interest in the 1970s deductive databases experienced a remarkable technological growth in the second half of the 1980s, when several research projects were started using a bottom-up approach (Morris, et al., 1987; Schmidt, et al., 1989). In this paper, we will focus primarily on the $\mathcal{LDL}$ project (Chimenti, et al., 1990) due to the level of maturity of the $\mathcal{LDL}/\mathcal{LDL}$++ prototypes and the author's familiarity with them. Valuable knowledge was acquired on technology development and technology transfer during the eight year life of the project, and can be summarized as follows:

1. A new implementation technology was developed for the efficient support of rule-based logic languages using techniques that are effective on secondary store as well as on main memory.
2. A new logic-based language was designed, which includes the design of several nonmonotonic constructs, and the definition of their formal semantics,
3. Critical know-how was gained on how to exploit the technology on advanced database applications.

The next two subsections contain a discussion of the first two items, and the following section discusses applications.

3.1. Language design

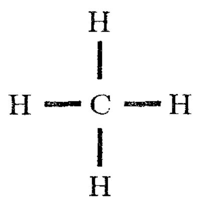
A first difference between a language such as $\mathcal{LDL}$ and Prolog is that the former is built around the notion of a database described through a schema and endowed with all database essentials. The schema defines the relations and their attributes (allowing for complex types and nested relations), unique key constraints, and indexing information. The schema also defines which relations are directly stored and managed by the $\mathcal{LDL}$ system and which are imported from external SQL databases (due to the limitations of SQL, the latter will have to be flat relations). In $\mathcal{LDL}$, the database is viewed as time varying, with the notions of recovery and database transactions deeply ingrained in the semantics of the language (Krishnamurthy, et al., 1989), the rule set and the database schema that define the program are instead time-invariant. Rules contain three kinds of constructs:

1. Horn-clause-based constructs
2. Nonmonotonic logic-based constructs (such as negation, aggregate operators, and choice)
3. Imperative constructs (such as updates and I/O)

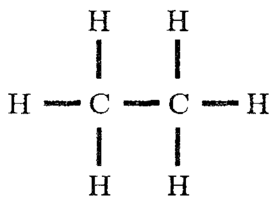
A language such as $\mathcal{LDL}$ shares with Prolog only the first of constructs above (i.e., Horn clauses), but not the remaining two. There are significant differences even with respect to Horn clauses, as illustrated by the fact that in deductive databases programs are less dependent on a particular execution model, such as forward-chaining or backward-chaining. A Prolog programmer can only write rules that work with backward chaining; an OPS5 programmer (Forgy, 1982) can only write rules that work in a forward chaining mode. By contrast, systems such as $\mathcal{LDL}$ (Naqvi and Tsur, 1989) and *Nail!* (Morris, et al., 1987), select the proper inference mode automatically, enabling the user to focus on the logical correctness of the rules rather than on the underlying execution strategy.

This point is better illustrated by an example. A methane molecule consists of a carbon atom linked with four hydrogen atoms. An ethane molecule can be constructed by replacing any H of a methane with a carbon with three Hs. Therefore, the respective structures of methane and ethane molecules are as follows:

Methane



Ethane



More complex alkanes can then be obtained inductively, in the same way: i.e., by replacing an `H` of a simpler alkane by a carbon with three `Hs`.

We can now define alkanes using Horn clauses. A methane molecule will be represented by a complex term, `carb(h, h, h)`, and an ethane molecule by `carb(h, h, carb(h, h, h))` (thus, we implicitly assume the presence of an additional `h`, the root of our tree). In general, alkane molecules can be inductively defined as follows:

```
all_mol(h, 0, Max).

all_mol(carb(M1, M2, M3), N, Max) ←

                          all_mol(M1, N1, Max),

                          all_mol(M2, N2, Max),

                          all_mol(M3, N3, Max),

                          N=N1+N2+N3+1, N <= Max.
```

In addition to defining alkanes of increasing complexity, these nonlinear recursive rules count the carbons in the molecules, ensuring finiteness in their size and number by checking that the tally of carbons never exceeds `Max`.

This alkane definition can be used in different ways. For example, to generate all molecules with no more than four carbons, one can write

```
? all_mol(Mol, Cs, 4).
```

To generate all molecules with exactly four carbons one will write

```
? all_mol(Mol, 4, 4).
```

Furthermore, say that there is a relation `alk(Name, Str)` associating the names of alkanes with their structure; then, the following rule will compute the number of carbons for an alkane given its name (assume that 10,000 is a large enough number for all molecules to have a lower carbon complexity).

```
find(Name, Cs) ← alk(Name, Str), all_mol(Str, Cs, 10000).
```

The first two examples can be supported only through a forward-chaining computation, which, in turn, translates naturally into the least-fixpoint computation that defines the model-theoretic based semantics of recursive Horn clause programs (Naqvi and Tsur, 1989). The least fixpoint computation amounts to an iterative procedure, where partial results are added to a relation until a steady state is reached.

Therefore, deductive databases support well the first two examples via forward chaining, while Prolog and other backward-chaining systems would flounder (by recurring on the first `all_mol` goal in the nonlinear recursive rule). In the last example, however, the first argument, `Str`, of `all_mol` is bound to the values generated by the predicate `alk`. Thus, a computation such as Prolog's backward chaining, which recursively propagates these bindings, is significantly more efficient than forward chaining. Now, deductive databases solve this problem equally well, by using techniques such as the *magic set method*, or the *counting method* that simulate backward chaining through a pair of coupled fixpoint computations (Ullman, 1989).

Since fixpoint computations check newly generated values against the set of previous values, cycles are handled automatically. This is a most useful feature since cyclic graphs are often stored in the database; furthermore, derived relations can also be cyclic. In our alkane example, for instance, there are many equivalent representations for the same alkane. To generate them, equivalence-preserving operations are used, such as rotation and permutation on the molecules. But, repeated applications of these operations bring back the previous structures. In order to detect these cycles in Prolog, a programmer would use a bag, passed around an as additional argument in the recursive calls, to "remember" the previously encountered structures. In deductive databases, there is no need to carry around such a bag since cycles are detected and handled efficiently by the system.

The declarative semantics and programming paradigm of deductive databases extend beyond Horn clause programming, to include nonmonotonic logic-based constructs, such as negation, sets, and choice operators. Thus, deductive databases currently support stratified negation (Naqvi and Tsur, 1989; Ullman, 1989), which is more powerful than negation-by-failure provided by Prolog. $\mathcal{LDL}$ also supports stratified aggregates. The areas of logic programming and nonmonotonic reasoning have been advanced significantly by the work on deductive databases.

3.2. Architectures

The key implementation problems for deductive databases pertain to finding efficient executions for a given query and a set of rules. For this purpose, deductive database systems perform a global analysis of rules—in contrast to Prolog compilers, which are normally based on local rule analysis. A global analysis is performed at compile time to evaluate the bound arguments and free arguments of predicates, on suitable representations such as the rule/goal graph (Ullman, 1989) or the predicate connection graph (Chimenti, et al., 1990). For a general idea of this global analysis is performed, consider the following example:

```
usanc(X, Y) <-- anc(X, Y), born(Y, usa).

anc(X, Z) <-- parent(X, Z).
anc(X, Z) <-- parent(X, Y), anc(Y, Z).
```

Thus, the last two rules supply the recursive definition of ancestors (parents of an ancestors are themselves ancestors) and the first rule choses the ancestors of a given X that were born in the USA (lowercase is used for constants, and uppercase for variables). Then a query such as

```
? usanc(mark, Y).
```

defines the following pattern:

usanc^{bf}

The superscript bf is an *adornment* denoting the fact that the first argument is bound and the second is not.

A global analysis is next performed to determine how the adornments of the query goal can be propagated down to the rest of the rule set. By unifying the query goal with the head of the usanc rule, we obtain the adorned rule:

$\mathrm{usanc}^{bf} \leftarrow \mathrm{anc}^{bf}, \mathrm{born}^{bb}.$

This adornment assumes that the first argument of born is bound by the second argument of anc, according to a sideway information passing principle (SIP) (Ullman, 1989). The next question to arise is whether the recursive goal anc^{bf} is supportable. The analysis of the anc rules yields the following adorned rules (assuming a left-to-right SIP):

$\mathrm{anc}^{bf} \leftarrow \mathrm{parent}^{bf}.$

$\mathrm{anc}^{bf} \leftarrow \mathrm{parent}^{bf}, \mathrm{anc}^{bf}.$

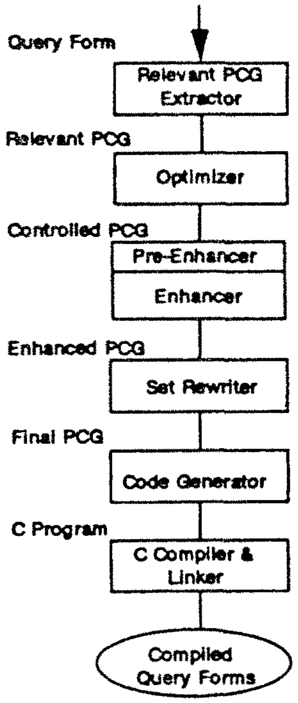


Figure 1. Architecture of the $\mathcal{LDL}$ system.

The analysis is now complete, since the adornment of the anc goal in the tail is the same as that in the head. Assuming that born and parent are database predicates, the given adornments can easily be implemented through a search taking advantage of the bound first argument in parent and both bound arguments in born. The recursive predicate anc can also be solved efficiently: in fact, a further analysis indicates that the recursive rule is left-linear (Ullman, 1989) and that the given adornment can, after some rewriting of the rules, be supported by a single-fixpoint computation (Ullman, 1989). When the recursive predicate cannot be supported through a single fixpoint, other methods are used, including the counting method, and the very general magic set method (Ullman, 1989).

Figure 1 describes the architecture of the $\mathcal{LDL}$ system. The first operation to be performed once a query form is given (a query form is a query template with an indication of bound/free arguments) is to propagate constants into recursive rules and to extract the subset of rules relevant to this particular query. By examining alternative goal orderings, execution modes, and methods for supporting recursion, the optimizer finds a safe strategy, that minimizes a cost estimate. For rules where all goals refer to database relations, the optimizer behaves like a relational system. The enhancer's task is to apply the proper recursive method by rewriting the original rules. A rule rewriting approach is also used to support the idempotence and commutativity properties of set terms.

Since recursion is implemented by fixpoint iterations, and only matching is needed at execution time, the abstract target machine and code can be greatly simplified, with respect to that of Prolog (Warren, 1983); thus, it can also be based on simple extensions of relational algebra. For instance, the first (limited) $\mathcal{LDL}$ prototype generated code for an intermediate relational-algebra language for a parallel database machine. The current $\mathcal{LDL}$ system is based on a uniform interface supporting get-next commands on databases residing in main memory or secondary store. The single-tuple interface supplies various opportunities for intelligent backtracking and existential variables optimization, exploited by the compiler to obtain good performance from the object code (Chimenti, et al., 1990). The intermediate object code is actually C, to support portability and a open architecture whereby external procedures can be incorporated into $\mathcal{LDL}$ and treated as database predicates.

Other experimental systems differ in several ways from the architecture of Figure 1. For instance, *Nail!* uses a relational algebra-based intermediate code, and employs *capture rules* (rather than cost-prediction based optimization) to drive the selection of a proper execution strategy (Morris, et al., 1987). The *CORAL* system also supports more general binding patterns, whereby certain variables of a predicate can remain unbound after the execution of a goal (Ramakrishnan, et al., 1992).

4. Applications

A most encouraging aspects of the $\mathcal{LDL}$ experience is represented by the range and significance of application domains for which it was found that deductive databases offer unique advantages. We now summarize some of the most significant applications, which are discussed in more details in (Tsur, 1990a, b).

4.1. Scientific databases

A major trend in modern science is toward large cooperative efforts in areas such as molecular biology, earth science, or neural science. Key to the success of these initiatives is the sharing of data and knowledge obtained by several teams of cooperating researchers (Pechura and Martin, 1991; Erickson, 1992). Databases are needed for sharing, protecting and organizing the project's scientific information, which frequently amounts to gigabytes or terabytes (Pechura and Martin, 1991; Erickson, 1992; Dozier, 1992). Deductive databases have much to offer for these applications, where the limitations of current database technology are obvious (Silverschatz, et al., 1990). In particular, considerable experience was gained with using $\mathcal{LDL}$ in solving various microbiology problems relating to the human genome initiative (Erickson, 1992). In addition to being used in developing several molecular biology applications, $\mathcal{LDL}$'s rules were used for modeling and supporting scientific taxonomies and concepts—thus filling the gap between high-level models and low-level experimental data (Tsur, 1990a).

Preliminary investigations on applying the $\mathcal{LDL}$ technology to support geophysical information systems have also produced encouraging results, but brought to focus the need for supporting abstract data types (ADTs) to provide specialized mechanism for spatial and temporal reasoning in rule-based systems. These topics will be further discussed in the next section.

4.2. Data dredging and knowledge discovery

These terms denote the emerging computational paradigm which supports "knowledge extraction" from, and the "discovery process' on the ever-growing repository of stored data (Tsur, 1990b). This usage of databases—in the past primarily associated with the intelligence community—is now becoming pervasive in medicine and science. Data dredging is also becoming common practice in such business applications as selective marketing and yield management (Hopper, 1990).

The source of the data is typically a large volume of low-level records, collected from measurements and monitoring of empirical processes, intelligence operations and businesses. The problem is how to use this data to verify certain conjectures and to help refine or formulate hypotheses. Typically, the level of abstraction at which hypotheses are formulated is much higher than that at which the data was collected. Thus, an iterative approach is needed, as follows:

1. Formulate the hypothesis or concept.
2. Translate the concept into an executable definition (e.g., a rule-set and query).
3. Execute the query against the given data and observe the results.
4. If the results fully confirms the hypothesis, then exit; otherwise, modify the initial hypothesis and repeat these steps.

Obviously, the decision to exit the process is subjective and falls upon the analyst or researcher who is carrying out the study. At any stage in this interactive process, the analyst may decide either that the concept is now adequately finalized and substantiated, or that the data does not support the initial conjecture and should be abandoned or tried out with different data. While, in principle, this procedure could be carried out using any programming language, the key to the experiment's practicality and timeliness hinges upon the ability to complete it with limited expenditures of time and effort. Thanks to the ability of quickly formulating very sophisticated queries and ruled-based decisions on large volumes of data, deductive databases are an ideal tool for data dredging. Our experience in developing such application with $\mathcal{LDL}$ also suggest that its open architecture is important in this process, inasmuch as, for example, a number of low-level, computation intensive tasks (such as filtering and preprocessing) must be used in the high-level, rule-driven discovery process.

4.3. Rapid prototyping of information system

The ability to model the data and procedures of a business enterprise is key to the successful development of an information system. The advantages offered by a deductive database environment in this domain were confirmed during the one-year field study described in (Ackley, et al., 1990), which summarizes the experience of using $\mathcal{LDL}$ in conjunction with a structured-design methodology called *POS* (Process, object, and state) (Ackley, 1991; Tryon, 1991).

A key idea of the POS methodology is that of using the ER framework for modeling both the dynamic aspects and the static aspects of the enterprise (Ackley, 1991). By using the notions of aggregation and abstraction within the ER framework, to capture what has traditionally been thought of as derived data, the ER model can specify most of the processing associated with a specific problem domain. This allows the capture of both data modeling and process modeling within one framework, thus eliminating the need for additional formalisms (such as workflow diagrams) in the final specifications. Furthermore, when a deductive computing environment is used, both the traditional generalization structures and the less-often-used aggregation structures can be directly encoded in a rule-based description, yielding executable specifications that are well-structured, easy to read, and have a formal semantics.

This basic approach was first tested in a case study, where a simplified information system was designed for the automobile registration authority (i.e., Department of Motor Vehicles). This information system involves modeling a set of entities (such as manufacturers, owners, garages, and motor vehicles of various types) and a set of events or transactions (such as the registration of various entities and the purchase and destruction of a motor vehicle). Several constraints must be enforced, including uniqueness, existence, and cardinality of entities, and restrictions on parties qualified by law to partake in different transaction types. Specific applications to be supported by such an information system include:

- Knowing who is, or was, the registered owner of a vehicle at any time from its construction to its destruction
- Monitoring compliance with certain laws, such as those pertaining to fuel consumption and transfer of ownership

In an informal study, also including a comparison with alternative prototyping frameworks, $\mathcal{LDL}$ proved very effective in this role and preferable to other approaches in terms of naturalness of coding, terseness, and readability of the resulting programs (Ackley, et al., 1991; Tryon, 1991). To a large extend, this is the result of the inherent ability of logic-based rule system to express and enforce business rules. For instance, a simple rule for the case study was

> "A given Motor Vehicle must have at least one and at most four legal entities as owners."

This business rule can be expressed as follows:

```
valid_ownership(Veh_Ser_No, Owner_id) <-

        owns_assertion(Veh_Ser_No, Owner_id),

        registered_vehicle(_, Veh_Ser_No),

        legal_entity(Owner_id),

        correct_no_owners(Veh_Ser_No).

correct_no_owners(Veh_Ser_No) <-

        owner_set(Veh_Ser_No, S),

        cardinality(S,N), N>0, N<5.

owner_set(Veh_Ser_No, <Owner>) <-

        owns_assertion(Veh_Ser_No, Owner).
```

In this case, a valid record of ownership is established by an assertion of ownership between a legal owner and a registered vehicle, all subject to the satisfaction of certain integrity constraints. Dynamic relationships dealing with validated events can be handled in a similar way. Such a declarative description of the business entities and activities is the key for successful enterprise modeling, rapid prototyping of new applications and the integration of various information systems of an enterprise (Tryon 1991,) which is discussed next.

4.4. Enterprise integration

For historical reasons, current operations in most companies are supported by patchwork collections of information systems and database systems, whose cooperation is impeded by the following discrepancies:

- Different representations for the same logical information
- Different data models and DBMS
- Different hardware and software platforms
- Distribution of data across different sites

In order to manage the increasing complexity of their business environments and to provide new services in a timely fashion, companies need to overcome present incompatibilities, integrate their information systems, and ensure logical and physical interoperability of their databases. In order to ensure the levels of interoperability, physical transparency and geographical transparency required by advanced applications, enterprises find it necessary to create an integration layer that supports a common interface between the external world and the underlying databases. While relational databases can now provide remote data access (RDA) protocols with distributed two-phase commit to ensure global interoperability and consistency, there remains the problem that the same information might be organized and store differently in different databases. Rules can be used to overcame these semantic differences and provide articulation axioms that map one representation to another. Furthermore, the information systems of many enterprises are still based on file systems or nonrelational DBMS. The semantic mapping of requests from the enterprise schema to the actual data thus require a mixture of logical transformations and procedural data access, which is well supported by the rules and open architecture of $\mathcal{LDL}$++.

Furthermore, it is often the case that old information systems and their data are poorly understood and maintained, thus creating what is known as the *legacy database* problem. Thus, some knowledge discovery and reverse engineering is often needed to recover the structure of the legacy database and the meaning of their original application programs. Unfortunately, it is often the case that the original database was stored redundantly (to ensure faster access) using unnormalized records, and that several ad hoc idiosyncratic fixes were later added to the original structure in the course of corrective maintenance or extensions. Thus, reclaiming the legacy database for inclusion into the global enterprise schema might involve schema redesign, data dredging, database sanitizing, and, possibly, rewriting of the old applications, after a rapid prototyping step to validate the results of the reverse engineering efforts. Thus legacy databases can benefit from deductive databases in several ways.

Given the high cost of having a database contaminated with errors, and the past trespasses of application programs in this domain (Bulkeley, 1992), one of the main functions of enterprise integration is to enforce global integrity. Therefore, the enterprise schema layer operates as a mediator that filters the requests by application programs, and dispatches the changes to the individual databases only after careful validation – this function is often called *data ingestion*. A sophisticated data ingestion discipline might actually use database triggers and relaxed transactions to fire sophisticated integrity checks, or perform compensatory actions to restore the database integrity under a "relaxed transaction" enforcement policy.

4.5. *Other applications*

The important applications just discussed were identified as areas of opportunities by prospective $\mathcal{LDL}$ users, rather than the $\mathcal{LDL}$ designers. The list of actual applications of $\mathcal{LDL}$ is in fact much longer and it also includes the two originally identified as areas of opportunities the $\mathcal{LDL}$ designers. The first such area is data-intensive expert systems. While several class projects were completed in this area, the only project of commercial significance was the design of a system for redistricting (e.g., gerrymandering) the state of Texas.

Inventory control and bill-of-materials applications were among the first to be generated on the $\mathcal{LDL}$ system, with independent studies confirming the potential of the technology in this application domain (Wahl, 1991). However, several of the applications developed in this domain had to use lists or imperative programs, since $\mathcal{LDL}$ does not allow aggregates or negation in recursive definitions—a limitation known as *stratification*. Thus the requirement to go beyond stratification emerged as a main design goal for $\mathcal{LDL}$++.

5. The $\mathcal{LDL}$++ system

By enabling the development of several pilot applications, the first generation of deductive database systems has proved the viability of this new technology and demonstrated its practical potential. Yet, this experience has also revealed the need for substantial improvements and extensions, which has motivated the design and development of a second generation of deductive database systems, such as the *Glue-Nail!* system, which supports a procedural shell called *Glue* that is closely integrated with declarative rules (Phipps, et al., 1991), and the $\mathcal{LDL}$++ system. (Also a system such as *CORAL* (Ramakrishnan, et al., 1992) that builds on the experiences with the first generation can be classified as a second-generation prototype.)

The prototypes of the second generation offer significant improvements and, in particular they (i) correct the limitations of their predecessors, (ii) reinforce their strengths and (iii) include the more recent advances in the enabling technology. The design of the new $\mathcal{LDL}$++ system was driven by these goals and the additional requirement of preserving upward compatibility at the language level with the $\mathcal{LDL}$ system.

5.1. *Development environment*

The $\mathcal{LDL}$ system operates by generating an equivalent program in C which can be up to 30 times the size of the original $\mathcal{LDL}$ program. The compilation and linking of such a C program is therefore slow. The slow turnaround on compilation frustrated the users, particularly newcomers who being in the experimenting and learning mode, performed frequent program changes and recompilations. Thus, a major objective of the $\mathcal{LDL}$++ system is a fast turnaround on compilation, which is obtained by interpreting directly the enhanced predicate connection graph (shallow compilation). The generation of equivalent C/C++ code (deep compilation) will remain an option to ensure a high performance level for the production code.

The development environment of $\mathcal{LDL}$++ will also be enhanced by a symbolic dedugger capable of both tracing the execution of the program and justifying the answers delivered by the program. It is believed that these extensions will greatly improve the ease of learning the language and of developing applications in a deductive database environment.

5.2. Open architecture

A significant contribution of deductive database prototypes is the realization that the new technology does not exclude, but, rather, can dovetail with existing programming environments. For instance, both $\mathcal{LDL}$ and $\mathcal{LDL}$++ support the incorporation of SQL schemas from external databases in the program, and the use of these relations as they were predicates defined in the $\mathcal{LDL}$ schema. Furthermore, the $\mathcal{LDL}$++ compiler analyzes predicates and rules defined on external relations to determine which are better implemented by off-loading them to the SQL database (e.g., select-join-project operations) and which should be implemented directly by the $\mathcal{LDL}$++ engine (e.g., fixpoint computations to support recursive predicates). Interfaces to nonrelational databases, including O-O DBMS, have also been demonstrated.

Any database query language is potentially faced with programming language interface problems at two levels. At the top level we find the programming language calling the query sublanguage (see the discussion of "impedance mismatch" problem) and at the bottom level we find the query sublanguage calling procedures defined in the programming language as subroutines. The bottom level interface is of paramount concern to current research in *extensible databases*. Extensible DBMS support the addition of new data types and relative operators (i.e., ADTs) to the query language (Stonebraker, et al., 1990). For instance, one might want to extend a query language with ADTs to express more easily and to support more efficiently queries and reasoning on spatial or temporal relationships. The question on how the definition of these ADTs using a programming language can then be incorporated into the query language and the underlying database system represent a significant research challenge of great current interest (Stonebraker, et al., 1990). To solve these problems, the $\mathcal{LDL}$++ system is designed to provide a complete integration with the C++ environment. Thus, classes defined in C++ can be imported into the $\mathcal{LDL}$++ environment and regarded as first-class members of the language through a correspondence that maps C++ structures into $\mathcal{LDL}$++ complex terms and C++ functions into $\mathcal{LDL}$++ predicates. Simple programming conventions must be followed in writing C++ classes to be imported in $\mathcal{LDL}$++; then, an imported C++ function can return multiple values as any logical predicate.

As an alternative to importation, new ADTs can also be defined through the module definition mechanism of the $\mathcal{LDL}$++ compiler. Since each $\mathcal{LDL}$++ module is compiled for execution into a C++ class, there is no run-time difference between imported ADTs and domestic ones. Furthermore, since compiled $\mathcal{LDL}$++ modules become C++ classes, they can be freely used in any C++ program.

5.3. Advances in the enabling technology

Deductive database technology is fast-maturing and important advances have been included in $\mathcal{LDL}$++, such as meta-level predicates with first order semantics, as (Phipps, et al., 1990).

A major extension featured by $\mathcal{LDL}$++ is its support for certain classes of nonstratified programs with negation and aggregates in recursion. The problem of going beyond stratification represents a topic of much current research, with contributions and ideas coming from the areas of AI, nonmonotonic logic and deductive databases. Among the most important contributions, there is the introduction of the concepts of well-founded models (Van Gelder, et al., 1991) and stable models (Gelfond and Lifschitz, 1988), which provide a declarative semantics for most programs of practical interest, including programs pertaining to various aspects of bill-of-materials applications. Formal declarative semantics, however, only represents one of the requirements that must be satisfied before nonmonotonic programs can be allowed in the recursive rules of the language. A second key requirement is amenability to efficient implementation; the third one is that the resulting programs must be easy to understand. While stratification satisfies all three conditions, general well-founded model semantics, or stable-model semantics, fail the second and third condition. As a result of this situation, the user might have to fall back to procedural programming whenever dealing with problems, such of the bill-of-materials requiring nonstratified negation or aggregates (thus the *Nail!* user will probably shift to Glue, while a *CORAL* programmer might use annotations to induce the proper execution strategy in the system.)

In $\mathcal{LDL}$++ however, the treatment of nonmonotonic construct will remain strictly declarative, according to the stable model semantics. This decision was taken to facilitate the writing of simpler and more efficient programs for critical algorithms for graph minimization problems (Ganguly, et al., 1991) or bill-of-materials applications. While a complete discussion of this complex topic is beyond the scope of this paper, the following discussion on the evolution of nondeterministic `choice` construct, will give the reader a concrete idea of the research problems faced and solution approach taken.

The idea of choice was introduced in (Krishnamurthy and Naqvi, 1988) to express nondeterminism in a declarative fashion. Thus, a construct such as choice(X, Y) is used to denote that the functional dependency $X \rightarrow Y$ must hold in the model defining the meaning of this program. For instance the following program, assign to each student an advisor from the same area:

```
st_ad(St, Ad)  ←  major(St, Area), faculty(Ad, Area),
                  choice(St, Ad).
```

Thus, while the result of joining major and faculty normally results in a many-to-many relation between students and professors (sharing the same area), the st_ad relation is restricted to be many-to-one, as a result of the presence of choice(St, Ad). In $\mathcal{LDL}$, this very powerful construct was disallowed in recursion, inasmuch as the functional-dependency based semantics proposed in (Krishnamurthy and Naqvi, 1988) suffers from technical problems such as a lack of justifiability property and its unsuitability to efficient implementation due to its static nature (Saccă and Zaniolo, 1990). These problems were avoided by using instead a semantics based on the use of negation and stable models (Giannotti, et al., 1991). For instance, the meaning of the rule above is defined its rewriting into the following nonstratified program:

```
      st_ad(St, Ad)  ←  major(St, Area), faculty(Ad, Area),
                        chosen(St, Ad).
     chosen(St, Ad)  ←  major(St, Area), faculty(Ad, Area),
                        ¬diffChoice(St, Ad).
 diffChoice(St, Ad)  ←  chosen(St, Ād), Ad≠ Ād.
```

Programs so rewritten always have one or more stable model, and each such a model obeys the *FD*: St→Ad. Now, the new semantics agrees with the original semantics for nonrecursive programs, but solves the problems of the old semantics in programs where choice occurs in a recursive predicate. For instance, say that a undirected graph is stored as pairs of edges, g(Y, X), g(X, Y). Then a spanning tree for this graph, starting from the source node a, can be computed as follows:

```
st(nil, a)

st(X, Y) ← st(_, X), g(X, Y), choice(Y, X).
```

Now, while the declarative semantics of choice rests on the theoretical foundations of nonmonotonic logic, a programmer need not be cognizant of the notion of stable models to make effective use of choice in his/her program; indeed, there is an equivalent operational semantics based on the notion of memorizing the old values of choice and checking every candidate new value for violations of the *FD* constraints. This operational semantics is also the basis for efficient implementations of the construct. Likewise, a combination of formal nonmonotonic semantics, intuitive appeal and amenability to efficient implementations are the ingredients that allow the relaxation of the stratification requirement in $\mathcal{LDL}$++.

6. Conclusion

Our experience with the $\mathcal{LDL}/\mathcal{LDL}$++ project fits the pattern that seems to be driving databases today: this can described as growing opportunities matched by escalating expectations. The initial project's goals, i.e., demonstrating the technical feasibility of supporting recursion and rules efficiently using database technology, were reached by the first-generation prototypes, such as $\mathcal{LDL}$. The introduction of these prototypes revealed that there are important areas of opportunity for the deployment of the new technology—opportunities well beyond those envisioned at the beginning of the project. On the other hand, that experience revealed several technical limitations of the first-generation prototypes, that had to be solved, so that they will not stand in way of the commercialization of the new technology. While many of the problems, such as lack of robustness and documentation, were typical of the growing pains of any new technology, the overcoming of other limitations, such as the addition of nonmonotonic constructs in recursion and of ADTs, required substantial research efforts. For the $\mathcal{LDL}$++ system, extensions to incorporate these two features have been has been designed, but still await implementation. Therefore, at the time of this writing, the $\mathcal{LDL}$++ prototype installed at UCLA (under MCC's license) supports several extensions, such as shallow compilation, linking to multiple databases, choice in recursion and metalevel predicates. However, nonstratified negation and aggregates are not yet implemented, nor are ADTs.

In the final analysis, therefore, the unavailability of a mature, supported, and affordable deductive database system remains the main obstacle for the spreading of the new technology in the commercial world. However, with the technology push and application pull of deductive databases growing steadily, it reasonable to expect that this situation will improve in the near future.

Acknowledgments

I would like to express my gratitude to Natraj Arni and Kayliang Ong for pushing forward with the design development of the $\mathcal{LDL}$++ system against all odds. Shalom Tsur must be credited for several contributions to the project, including most of the applications described in Section 4.

References

Ackley, D. (1991). Process-object-state: an integrated modeling method. *A Framework of Information Systems Architecture, Conf.* Virginia (reprints available from author: 210 Almeira Ave, Freemont, CA 94539, 415-656-1665)

Ackely, D., et al. (1990). System analysis for deductive database environments: an enhanced role for aggregate entities. *Proc. 9th Int. Conf. Entity-Relationship Approach*, Lausanne, Ch.

Bulkeley, William M., "Databases Are Plagued by Reign of Errors," The Wall Street Journal, Tuesday, May 26, 1992 (Information Age).

Ceri, S., Gottlob, G., and Tanca, L. (1989). *Logic Programming and Deductive Databases.* New York: Springer-Verlag.

Chimenti, D., et al. (1990). The *LDL* System Prototype. *IEEE Journal on Data and Knowledge Engineering, 2,* 76–90.

Connell, J.L. and Shafer, L.B. (1989). *Structured Rapid Prototyping.* Englewood Cliffs, NJ: Prentice Hall.

Dozier, J (1992). Access to data in NASA's earth observation system (Abstract). *Proc. ACM-SIGMOD Conf. Management of Data.*

(1989). The Rapid Prototyping Conundrum. *DATAMATION*, June.

Erickson, D. (1992). Hacking the Genome. *Scientific American*, April.

Forgy, C.L. (1982). Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence, 19,* 17–37.

Gane, C. (1989). *Rapid System Development.* Englewood Cliffs, NJ: Prentice Hall.

Ganguly, S., Greco, S. and Zaniolo, C. (1991). Minimum and maximum predicates in logic programming. *Proc. 10th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems* (pp. 154–164).

Gelfond, M. and Lifschitz, V. (1988). The stable model semantics of logic programming. *Proc. Fifth Int. Conf. Logic Programming* (pp. 1070–1080).

Gellene, D. (1991). Airlines Discourage Bargain Hunts—Travel Agencies Face Hefty Fees to Use New Computer Programs, *Los Angeles Times*, September 10, D1, D7 (Business Section).

Giannotti, F., Pedreschi, D., Sacca, D., and Zaniolo, C. (1991). Nondeterminism in deductive databases. *Proc. 2nd Int. Conf. Deductive and Object-Oriented Databases.*

Hopper, D.E. (1990). Rattling SABRE—New Ways to Compete on Information. *Harvard Business Review*, May-June, 118–125.

Kiernan, G., de Maindreville, C. and Simon, E. (1990). Making deductive database a practical technology: a step forward. *Proc. 1990 ACM-SIGMOD Conf. Management of Data* (pp. 237–246).

Kifer, M. and Wu, J. (1989). A logic for object-oriented programming (Maier's O-logic revisite). *Proc. 8th, ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Management (PODS).*

Krishnamurthy, R. and Naqvi S. (1988). Non-deterministic choice in datalog. *Proc. 3rd Int. Conf. Data and Knowledge Bases.*

Krishnamurthy, R., Naqvi, S. and Zaniolo, C. (1989). Database transactions in *LDL*. *Proc. Logic Programming North American Conf.* (pp. 795–830). MIT Press.

Morris, K., et al. (1987). YAWN! (Yet Another Window on NAIL!). *Data Engineering*, 10, 28–44.

Moss, C. (1986). Cut and paste—defining the impure primitives of prolog. *Proc. Third Int. Conf. Logic Programming* (pp. 686-694). London.

Naqvi, S.A., Tsur, S. (1989). *A Logical Language for Data and Knowledge Bases.* W.H. Freeman.

Pechura, M. and Martin, J. (eds.) (1991). *Mapping the Brain and its Functions.* Washington, DC: National Academy Press.

Phipps, G., Derr, M.A., and Ross, K.A. (1991). Glue-Nail: a deductive database system. *Proc. 1991 ACM-SIGMOD Conf. Management of Data* (pp. 308–317).

Ramakrishan, R., Srivastava, D., and Sudarshan, S. (1992). *CORAL*: a deductive database programming language. *Proc. VLDB'92 Int. Conf.*

Rowe, L. and Shoens, K. (1979). Data abstraction, views and updates in RIGEL. *Proc. ACM-SIGMOD Int. Conf. Management of Data*, Boston, MA.

Saccà, D. and Zaniolo, C. (1990). Stable models and non determinism in logic programs with negation. *Proc. 9th, ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems* (pp. 205–218).

Schmidt, H., et al. (1989). Combining deduction by certainty with the power of magic. *Proc. 1st Int. Conf. Deductive and O-O Database*. Kyoto, Japan.

Schmidt, J. (1977). Some High Level Language Constructs for Data Type Relation. *ACM TODS, 2.*

Silverschatz, A., Stonebraker, M. and Ullman, J. (Eds.) (1990). Database systems: achievements and opportunities. The Lagunita Report of the NSF Inv. Workshop on the Future of Database Systems Research, Held in Palo Alto, CA, *SIGMOD RECORD, 19.*

Stonebraker, M., Rowe, L. and Hirohama, M., (1990). The Implementation of POSTGERS. *IEEE Journal on Data and Knowledge Engineering, 1,* 125–143.

Tryon, D. (1991). Deductive computing: living in the future. *Proc. Monterey Software Conf.*

Tsur S. (1990). Deductive databases in action. *Proc. 10th, ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems* (pp. 205–218).

Tsur S. (1990). Data Dredging, *Data Engineering, 13.*

Ullman, J.D. (1989). *Database and Knowledge-Based Systems.* Vols. I, II. Rockville, MD: Computer Science Press.

Ullman, J. and Zaniolo, C. (1990). Deductive Databases, Achievements and Future Directions. *SIGMOD Record, 19,* 77–83.

Van Gelder, A., Ross, K.A., and Schlipf, J.S. (1991). The Well-Founded Semantics for General Logic Programs. *Journal of ACM, 38,* 620–650.

Wahl, D. (1991). Bill of Materials in Relational Databases—An Analysis of Current Research and Its Applications to Manufacturing Databases. Digital Equipment Corp. Internal Report.

Warren, D.H.D. (1983) An Abstract Prolog Instruction Set. Tech. Note 309, AI Center, Computer Science and Technology Div., SRI.

Zaniolo, C. (1984). Prolog, a database query language for all seasons. *Proc. 1st Int. Workshop on Expert Database Systems*, Kiawah Island, SC.

Zaniolo, C. (1989). Object identity and inheritance in deductive databases: an evolutionary approach. *Proc. 1st Int. Conf. Deductive and O-O Databases*. Kyoto, Japan.