A Knowledge Base Design and Application Prototyping Tool Based on an Enhanced Functional Data Model

Satya Prabhakar
Systems and AI Department
Honeywell (SSDC)
3660 Technology Drive, Minneapolis, MN 55148

Shamkant Navathe
School of Computing
Georgia Institute of Technology
Athens, GA, 30332

Abstract

Significant research is being done in the area of knowledge representation and modeling to reduce the semantic gap between reality and its representation. Most models, while intuitively appealing, lack a formal basis to be of practical use. In this paper we describe a knowledge base design and prototyping tool, Crayon, based on an enhanced functional data model implemented in PROLOG. The objective of this exercise was to explore use of this data model for knowledge base management. We discuss the advantages of the model (formal representation, semantic richness and simplicity) from the perspective of two knowledge representation guidelines and the advantages of using PROLOG as the implementation language for knowledge base prototyping. We propose and discuss theoretical enhancements to the functional data model in the form of knowledge-oriented constraints. The design and functioning of Crayon are described with an expert system shell as the application using the knowledge base management system as a server.

1: Introduction and motivation

The development and definition of constructs, mechanisms and techniques to more effectively represent the real world continues to be one of the active areas of research in the fields of semantic data modeling, object-oriented data management and knowledge representation. The goal of these efforts is to reduce the semantic gap between reality and its representation in a way that makes it easier for computers to process the information efficiently and correctly. However, reality is dependent on our perception and the way we understand and process objects and events we see every day. To reduce the semantic gap between reality and its representation in computers, the representation mechanism must be rich enough to capture our perception of reality. This is a formidable task for two reasons: (1) at this time, our ability to formalize all the knowledge representation mechanisms we employ in our minds is quite limited; (2) the more complex the representation, the more difficult it becomes to reason with and process the information using a computer. If the representation framework is complex with numerous orthogonal concepts and constructs, it becomes intractable and useless.

Most of the recent work in computer representation employs the notion of an “object” to represent real-world entities and their interrelationships. While intuitively appealing, many of these models lack a formal foundation, making efficient working implementations difficult to realize. The relational data model [1] owes much of its success to its well-defined semantics, but the limitations imposed by the model have been well-documented [2–5]. The need for future data and knowledge base management systems is a formal, well-defined data model that captures richer abstractions in addition to data. The functional data model is a good example of such a model, blending both formal definition and richer semantic constructs. This paper describes a knowledge base prototyping tool based on the functional data model.

The paper consists of two parts. The first (Sections 2 and 3) is a theoretical discussion of the functional data model, its merits, and a few enhancements to capture additional knowledge-oriented semantics. This part also discusses the advantages of using PROLOG as the implementation language. The second part (Section 4) describes the implementation of a knowledge base prototyping tool using the enhanced functional data model as well as PROLOG. This part also describes an expert database system framework that makes use of the knowledge base prototyping tool as a server.

The objectives of this exercise were (1) to build a flexible, yet powerful, prototyping mechanism to explore the use and advantages of the functional data model for knowledge base applications, and (2) to explore the synthesis of an expert system with this knowledge base tool to realize what is commonly referred to as an “expert database system” [6].

0-8186-2300-4/91 $01.00 © 1991 IEEE
2: Selection of the data model and the implementation language

This section presents arguments for choosing the functional data model as the representation model and the use of PROLOG as the implementation medium.

2.1: Functional data model

The need for semantically richer data models that can capture more complex real-world information has been well documented. Research in the areas of object-oriented data management, semantic data modeling and knowledge representation has been, in part, directed toward formalizing models for representing semantically complex data interrelationships [7,8]. This research has sometimes resulted in overly complex data representation frameworks (e.g., Krypton) that were unmanageable and impractical for regular use. Levesque and Brachman [9] and Smith [10] have outlined the following guidelines for any representation scheme:

- **Tradeoff between semantic richness and tractability:** Levesque and Brachman [9] argue that there is a tradeoff between the representational power of a knowledge representation schema and the ability to reason efficiently and correctly using that schema. In other words, if a knowledge representation schema contains several different orthogonal constructs to represent different aspects of the domain, then the ability to reason efficiently (tractability) and correctly (provability) will be correspondingly difficult.

- **Knowledge representation hypothesis:** Smith [10] outlines the knowledge representation hypothesis (KR Hypothesis) as follows:

  Any mechanically embodied intelligent process will be comprised of structural ingredients that (a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and (b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behavior that manifests that knowledge.

  This hypothesis underscores two important aspects of knowledge representation. First, any knowledge representation mechanism must contain (data) structures that present a (propositional) account of the knowledge incorporated in the system. Second, in addition to and independent of the knowledge the structures present to external observers, they should also play an important role in the intelligent behavior the system is required to manifest.

  The decision to use an enhanced functional data model [11,12] was based on the two theories described above.

  Even though these theories are only guidelines rather than quantitative litmus tests, the functional data model, in our opinion, meets the requirements to a significant extent. (A brief overview of the functional data model can be found in section 3.) Specifically, the following are benefits of using the functional data model:

  - **Expressive power:** The functional data model provides a semantically rich framework with several orthogonal constructs such as aggregation, generalization, association, identification and derivation. Specifically, the treatment of derived data using the notion of composite functions in the same way actual data is used is one of the distinguishing features of the functional data model.

  - **Theoretical Foundation:** Kerschberg and Pacheco [11] used the well-defined concept of a function as the basis of the functional data model. All the orthogonal constructs of the functional data model—aggregation, generalization, association, identification and derivation—are defined as variations of one concept, the function.

  - **Simplicity:** Unlike other knowledge representation schemes, the expressive power of the functional data model was not achieved at the expense of simplicity. As mentioned above, all the orthogonal concepts of the model were developed using the concept of the function. Because of its simplicity, the functional data model scores well against the tradeoff test described above.

  - **Removal of sharp distinction between data and rules:** The functional data model supports the notion of a virtual entity (also called a “derived entity”) through the composition of the functional definition of primitive entities. The virtual entity can then be used like other entities. This feature erases the boundary between definition and derivation.

2.2: PROLOG

PROLOG was chosen as the implementation language for this knowledge base prototyping tool. The reasons and corresponding benefits of selecting PROLOG are as follows [13-15]

- **Representational diversity and flexibility:** A PROLOG predicate or a combination of predicates can be used to represent at least four different knowledge structures: facts (or tuples), declarative structures, procedures and rules [13].

- **Representational closeness to the functional data model:** The notation of the functional data model is remarkably similar to the declarative structures of PROLOG. This simplifies the programming task and enhances the readability of programs.
• **High-level specification:** In PROLOG, in marked contrast to other programming languages, the specification of the problem quite often embodies the solution; in other languages, the procedural mechanics of the solution have to be specified.

• **Data/rule homogeneity:** Much like the functional data model, PROLOG does not distinguish between facts and declarative predicates. This closeness also contributes to programming and modification ease.

We chose PROLOG to be the implementation medium for the above reasons.

3: **Enhancements to the functional data model**

3.1: **Overview of the functional data model**

The chief constructs of FDM are entities and functions that relate these entities to one another. All the entities can be organized as a hierarchy under the root entity ENTITY, which is system defined. Other predefined functions are STRING, INTEGER, BOOLEAN and REAL. The syntax of FDM is provided by Shipman [16].

**Person** can be defined as an entity using the functional notation without any argument, as follows, supporting the property of **Identification**:

\[ \text{Person()} \rightarrow \text{ENTITY} \]

The functional data model also supports the features of **generalization** and **inheritance** by allowing entities to be defined as subclasses of other entities as follows:

\[ \text{Student()} \rightarrow \text{Person} \]

Properties of an entity can be defined as a mapping from the domain of **Person** to the attribute domains, supporting **aggregation**:

\[ \text{Name(Student)} \rightarrow \text{String} \]
\[ \text{SSN(Person)} \rightarrow \text{Integer} \]

The functions can be multivalued to provide for attributes that can have multiple values. This is represented by a double-headed arrow. By virtue of being a subclass of **Person**, the class of **Student** inherits the attribute of **SSN**.

To capture the notion of **association**, the functional data model maps a relationship between two entities as follows:

\[ \text{Course(Student)} \rightarrow \text{Course} \]

Functions can have multiple arguments or arguments that are composite. To identify the location of the courses a student is taking, the following function can be defined:

\[ \text{Location(Course(Student))} \rightarrow \text{String} \]

**Grade** can be defined as an attribute of the relationship between the two entities **Student** and **Course**:

\[ \text{Grade(Student,Course)} \rightarrow \text{Char} \]

Any of the arguments can be a function, too. For example, **Grade** can be defined over only legitimate combinations of **Student** and **Course**:

\[ \text{Grade(Student,Course(Student))} \rightarrow \text{Char} \]

A distinctive feature of the functional data model is its ability to support the representation of derived data within its data definition language. Given the following definitions:

\[ \text{Instructor(Course)} \rightarrow \text{Instructor} \]
\[ \text{Course(Student)} \rightarrow \text{Course} \]

a new relationship can be derived as follows:

\[ \text{Instructor(Student)} \rightarrow \text{Instructor(Student)} \]

This new virtual function may now be used as if it were primitively defined. It can be used in queries even though no physical data is stored for this function relation. This derived data can now also be used to derive further new information, adding significantly to the expressive power of the model.

Using a simple and well-understood functional notation, the functional data model supports many abstractions absent in conventional data models: identification, aggregation, generalization, inheritance, association and derivation [17].

3.2: **Enhancements**

While the functional data model supports several orthogonal abstractions, it is deficient in its ability to express constraints satisfactorily [18]. The ability to specify constraints is important both to present the user with a true picture of the domain (so that he or she can better understand the dependencies between entities) and to enforce these constraints so that the integrity of the database is maintained [19,20]. Where a data model fails to capture these constraints, the application programs that use the database must implement these constraints individually. This is not only highly inefficient but is likely to lead to serious inconsistencies when even one program fails to enforce the constraints.

Most of the enhancements proposed here are related to the specification of constraints. (Due to space considerations, only a brief overview is provided.) Constraints can be classified as inherent, explicit and implicit. Inherent constraints are the structural specification mechanisms of the data model (DBL) used to capture the restrictions of the domain. Explicit constraints are those that have to be specified externally to any entity definition in the data model. Implicit constraints are those that are implied by the inherent and explicit constraints. All three categories have been enhanced to achieve greater expressive power. Only inherent and explicit constraints are described here.

Due to space limitations, the BNFs for the enhancements are not presented here. The BNFs can be found in Reference [21].
3.2.1: Inherent Constraints: The additional inherent constraints that have been defined to supplement the basic constraints of the original functional data model can be categorized as follows:

- **Attribute value constraints**: These constraints define the restrictions on the types of values an attribute of an entity can take. The functional data model specified basic data types such as integer, string, real, etc. The additional data types that have been defined are:
  - List of Values (e.g., Readings(Experiment) ----> list of reals)
  - Enumerated Sets (e.g., Type(Car) ----> ["Hatchback," "Sedan," "Convertible"])
  - Range of Values (e.g., SSN(Employee) ----> [000000000-999999999])
  - Compute Type (e.g., Bonus(Employee) ----> Salary * 0.10)
  - Rule Type (e.g., Type(Stock) ----> BlueChip if (Yield(4.0) && (B,1.1))

- **Cardinality and participation constraints**: In addition to the type constraints on the attribute values, there could be constraints that specify the number of values an attribute can take. For example, a section of a course could be constrained to have no less than 10 students and no more than 30 students. Another instance of such a constraint is the restriction that the social security number attribute of an employee can take one and only one value. To satisfy the specification of such a constraint, an additional pair of values is attached to each attribute of an entity. This constraint, which can be referred to as the min-max constraint, is specified as a pair of values for each attribute:

\[ \text{SSN} \text{(Employee)} \rightarrow [00000000-999999999], [1,1] \]
\[ \text{Student(Section)} \rightarrow \text{Student} [10,30] \]

The same min-max pair specification can be used to represent the cardinality restriction in an entity relationship. Relationships between entities have certain constraints, limiting the possible combinations of the entities. The (min, max) pair can be used to restrict the number of relationships between two entities. Together, the cardinality and participation constraints represent the structural constraints of a relationship.

While all constraints represent knowledge inherent in the domain, the COMPUTE and RULE types define functional dependencies between data objects and derive or infer knowledge as part of the data definition language [22].

The above enhancements subscribe to the KR Hypothesis discussed in section 1 for the following reasons:

- Specification of the above constraints provides a more accurate propositional account of the domain (it is representing) to an external observer.
- These constraints play a causal role in engendering the intelligent behavior that is expected of a knowledge base system. In other words, when a user tries to modify the database in violation of any of the constraints, the constraints "trigger in" and the user is prevented from committing the changes.

3.2.2: Explicit constraints: In addition to structural constraints, constraints are needed that are not specific to an entity but to the database as a whole. These constraints are defined externally to the data definition of any entity; these constraints ensure that the database is in a consistent state against manipulations. The explicit constraint language specified here provides the mechanism for specifying both static and dynamic constraints. A similar structure has been proposed by Kerschberg [23]. The body of the constraint is segmented as follows:

- List of relevant entities,
- Triggering operation,
- Constraint or precondition,
- Explanation response,
- Post-condition.

The first element of the constraint specifies the list of entities that are involved in the constraint specification. Whenever an operation is attempted on an entity, the system identifies the constraints that are relevant to the entity by scanning this list. The second part of the constraint is the trigger condition specifying the operation that should trigger this constraint. The precondition part of the constraint specifies the condition that has to be satisfied for the transaction to be allowed. If this condition is not met, the constraint triggers and takes further action. So, for an explicitly defined constraint to be defined, the first three parts have to be satisfied. The violation response defines the reaction of the system, in the event of a violation. This might include a rollback mechanism with an explanatory capability informing the user of the violation. The last segment specifies all the conditions that have to be satisfied after the operation is completed.

A more detailed explanation of the enhancements with supporting examples can be found in Prabhakar [21].

4: Tool architecture and functionality

This section describes the tool (named "Crayon" for reference purposes) based on the enhanced functional data model and implemented in PROLOG. Crayon was designed and implemented to support the following major functions:

- **Knowledge base management system**: Define the knowledge base schema, populate with data and then access/manipulate the data;
- **Interactive browser with limited graphic capability**: An interactive graphic interface to browse schema definitions and actual data;
Expert database system shell: Specification of an expert system shell (using native PROLOG inference mechanisms) that makes use of the knowledge structures of the knowledge base management system.

In designing and developing Crayon, we paid close attention to modularity and user friendliness. With respect to modularity, all modules were developed using the application program interface (API) approach whereby each module provides access (to the modules it interacts with) via a well-defined interface. For example, the interface between the ES and KB Manager modules is a set of function calls. Other implementation details of the KB Manager are irrelevant to the ES module. With respect to user friendliness, the following were provided: (1) a dynamic windowing mechanism, (2) constant on-line help, (3) an ad hoc context switching facility whereby the user can switch between contexts such as incremental query formulation and knowledge browsing.

4.1: Module design

The design of Crayon is described here in a top-down fashion. As shown in Figure 1, Crayon consists of two functionally distinct modules:

- Knowledge Base Manager (KB) module: This module can function in two modes: as a server supplying required information to the expert system module (Figure 2), or as an independent module providing knowledge base management functions to users via an independent, interactive user interface (Figure 3).

- Expert System (ES) module: The expert system module is dependent on the knowledge base management module for providing access to the knowledge that drives the inference engine. This module only needs to be aware of the knowledge retrieval interface, i.e., the set of procedures to invoke to retrieve different types of information, and not the internal implementation of the KB Manager.

Using the commonly used terminology, the coupling between the knowledge base and deductive component is loose. The ES and KB modules are distinct modules with the only interaction occurring through the call-level API. In a tightly coupled expert database system, the information stored in the KB is a natural extension of the information stored in the ES, and both are tightly integrated. The significant advantage of designing a loosely coupled architecture is that the KB can be replaced with an existing database management system that can somehow support the call-level API. Both these modules are further defined below.

The main options provided by Crayon are shown in Figure 4.

4.1.1: Knowledge base manager module: This module supports the creation, access, manipulation and browsing of knowledge defined within the framework of the enhanced functional data model. We use the term "knowledge" rather guardedly. While we are aware that there is neither a clear understanding nor a general consensus as to what constitutes knowledge, and that the arbitrary use of the term is rather pretentious, we use the term to distinguish the information managed by Crayon from the information managed by conventional database management systems. The term "knowledge" has been used in this manner by other researchers [23]. In addition to managing structured data, Crayon also supports higher levels of abstraction such as generalization, inheritance, derivation, rules, computation, etc., as part of the data definition language.
The major components of the KB module are:

- **Interactive user interface manager (UIM):** This module manages the interface with the user. The interface uses the pop-up window and menu paradigm to interact with the user.

- **Coordinator:** This is the heart of the KB. The UIM interacts with the coordinator (again using a well-defined API) and passes along the user request. Accordingly, the coordinator invokes the appropriate functional module, such as the Schema Definition and Manipulation module, the Knowledge Retrieval and Manipulation module, etc.

- **Schema definition and manipulation module:** This module allows the user to define a schema using the functional data model constructs and to manipulate existing schemas if necessary. Each schema generally
has a knowledge base (instantiations) populated using the structures defined by the schema. The various options and the control flow of this module are shown in Figure 5.

- Knowledge retrieval and manipulation module: This module provides access to the instantiated knowledge of the schema. The three functions supported by this module are general access, addition/deletion and modification of knowledge structures.

- Consistency checker module: This module contains the data definition restrictions of the enhanced functional data model and the constraints defined for different schemas. The consistency checker module serves the following purposes:
  - At schema creation time, the consistency checker enforces the restrictions placed by the functional data model. For example, if the user defines an entity as a subclass of two other super entities, the consistency checker traps the error as multiple inheritance is not supported by the functional data model.
  - When the knowledge base is being modified, the consistency checker uses the constraints specified for the corresponding schema to ensure the overall integrity of the knowledge base. For example, in the case of an entity with an enumerated attribute, if a value is added to that attribute that does not belong to the enumerated set, the consistency checker identifies the error and warns the user.

- Secondary storage manager: This module handles the storage and retrieval of data from the secondary storage device making use of the indexed file access facilities provided by the PROLOG language.

4.1.3: Expert system module: This is a normal expert system shell implemented in PROLOG that uses a forward-chaining inference engine. The inference engine contains (API) calls to the KB module to retrieve whatever information it needs to answer the queries of the users. A Investment Advisor expert system was created to advise on investment based upon user profile. This expert system makes use of information about 100 different stocks and bonds managed by the KB module. Due to space considerations, the ES module is not explained in great detail here. More information can be found in Reference [21].

5: Conclusions

In summary, the design and development of Crayon has been a fruitful exercise in terms of understanding the applicability of the functional data model and the PROLOG language for knowledge base applications. Specifically, the following are conclusions:

- Functional data model: The data model lends itself well to representing complex semantic constructs, especially with additional constraints that have been
defined; while the simple constraints specified in the functional data model were easy to deal with, complex dynamic (external) constraints posed problems of tractability and closure.

- **PROLOG:** Programming Crayon in PROLOG proved to be easier for design, prototyping and modification; features of PROLOG like recursion, backtracking and natural inferencing proved especially useful to implement features such as interactive browsing, inheritance propagation, etc.; however, the forward-chaining inference engine of PROLOG may cause problems.

- **Usefulness:** The tool proved useful for building fast prototypes of relatively simple knowledge base schemas; the schema browsing and incremental query formulation facilities were especially appealing to naive users; however, regular use of the tool for complex domains proved difficult due to the weakness of PROLOG in maintaining large databases.

### 6: References


