Intensionally Integrating
Database and Expert Systems

Tore Risch

HP Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94303

This work was done when the author was at
Syntelligence, Inc., P.O.Box 3620, Sunnyvale, CA 94088

ABSTRACT: We describe a new system architecture that draws on concepts from several different programming paradigms. This architecture, called an active functional system, shares certain characteristics with database systems, expert systems, data flow languages, and spreadsheet systems, and yet is very different from any of these. It is based on a uniform use of side-effect free functions as the means for representing facts and knowledge in a non-procedural programming system. The system evaluates functions in a data-driven, bottom-up way, which requires certain optimizations for efficient execution. The system also provides a clear, uniform method for handling unknown, inexact, and default values.

1 Introduction

Database systems handle large amount of data with relatively simple structure and little inferencing. Expert systems handle moderate amount of data with complicated structure and much inferencing.

There has recently been some interest in coupling database and expert systems [4,11]. The most straight-forward way to do this is by loosely coupled systems in which the database module acts as a server for the expert system module [1]. However, there are also many applications that place equal demands on both the expert system and the database. For these applications, a tightly coupled approach would seem more desirable.

The system described here, Syntel[15,9] 1, is an example of a tightly coupled expert and database system, specially designed for financial applications

---

1The name 'Syntel' is trade-marked by Syntelligence, Inc. This work was done while I was employed at Syntelligence, and benefited greatly from the contributions of my colleagues there.
[9,15], but whose architecture can be used also for other kinds of applications. It has been fully implemented and in commercial use since mid 1986. It uses declarative functions to represent both knowledge and facts in an expert system. The uniformity of the results generated by functional representations leads to a corresponding uniformity in database and knowledge-base operations. We call the system 'active' because it is data driven. To be more specific, changes in the distributions of the factual input data are propagated through the knowledge base to update the distributions of the derived output data.

Viewed as a programming language, Syntel is a non-procedural functional data-flow language [2], sharing certain properties with Lucid [19] and Lucas and Risch's definitional equation interpreter [13]. Among the better known data-driven expert systems shells or languages, Syntel is related to Prospector/KAS [8], OPS5 [5] and Oncocin [18]. From a database point of view, Syntel is related to the DAPLEX [16] functional data model, which also has been generalized into an object oriented data model [10].

2 Knowledge Representation in Syntel

The concepts used to represent data and knowledge in Syntel are extensional functions (also called value tables), intensional functions, types, and forms. The language is non-procedural because the functions used are side effect free, no control primitives are available to the programmer, and the system decides in which order to evaluate them.

2.1 Extensional functions

Syntel represents both changing, case-specific data and persistent data using a uniform data structure called value tables, which basically are representations of extensional definitions of functions. For example, Fig. 1 gives the extensional function (or value table) Expenses[Year,State]. The columns Year and State identify each row in the value table and are called the parameters of the value table. Value tables can hold both numeric and symbolic data. We refer to a single row in the value table as an instance.
<table>
<thead>
<tr>
<th>Year</th>
<th>State</th>
<th>Expenses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1986</td>
<td>'CR'</td>
<td>$20,000</td>
</tr>
<tr>
<td>1987</td>
<td>'AZ'</td>
<td>$15,000</td>
</tr>
<tr>
<td>1988</td>
<td>'AZ'</td>
<td>$26,000</td>
</tr>
<tr>
<td>1988</td>
<td>'UT'</td>
<td>NIL</td>
</tr>
<tr>
<td>1989</td>
<td>'WA'</td>
<td>$13,000</td>
</tr>
<tr>
<td>1989</td>
<td>NIL</td>
<td>$20,000</td>
</tr>
<tr>
<td>NIL</td>
<td>NIL</td>
<td>$10,000</td>
</tr>
</tbody>
</table>

Fig. 1 A value table

The value of an instance can also be the special symbol NIL, (as for Expenses[1988, 'UT']) meaning that the value is currently unknown. If a parameter in the value table is NIL (as for Expenses[1989, NIL] and Expenses[NIL, NIL]), the value is a default value to be used if no exactly matching instance for a given parameter value exists. In particular, the default value when all the keys are NIL is called the prior value. For example Expenses[1989, 'CA'] will be looked up as $20,000 and Expenses[1990, 'OR'] as $10,000, the prior value.

The value of an instance may also be a probability distribution [9]. In this way Syntel has a natural way to handle the inexactness originating in missing data and incomplete information that is common in expert systems [8,5,18].

2.2 Intensional functions

Syntel uses a family of side-effect free primitive functions to operate on value tables. Mathematically each primitive function is an intensionally-defined mapping from one or more value tables into a single value table. Thus, the primitive functions are second order functions.

The primitive functions are used to define derived value tables in terms of other value tables. For example, we might want to define a derived value table Income[Year, State] given value tables Revenue[Year, State] and Expenses[Year]. We use an equational format for this:

\[
\text{Income}(\text{Year, State}) \leftarrow \text{Difference} (\text{Revenue}(\text{Year, State}), \\
\quad \text{Expenses}(\text{Year}))
\]

Note that this equation is strictly definitional. It expresses the extensional function Income as an intensional function Difference of the two extensional function Revenue and Cost.
The intensional functions (e.g. **Difference**) normally are applied instance by instance to define derived extensional functions. Syntel functions are referentially transparent, and so can be composed to arbitrary depth.

Syntel provides some 65 primitive intensional functions. They may be classified into several classes, as arithmetic, logical, and string manipulating functions. Other functions are used to represent expert judgement, such as judgements that involve comparing and weighting the importance of diverse factors leading to some conclusion. These functions play a role analogous to rules in traditional expert systems, such as Mycin [17]. Other functions transform value tables and perform aggregations over value tables. Those functions correspond to relational algebra operators in database systems. All primitive function also have to handle inexact as well as exact values, which is done by using the theory of induced probability distributions [9].

A limited form of recursion over the finite, extensional functions is also supported.

Value tables are similar to database relations with a single non-key column. The operation performed by primitive functions such as **Difference** is similar to the relational database operator **join** followed by an arithmetic operator and a projection. In addition the system must take special care to maintain correct default values of derived value tables.

### 2.3 Types

Syntel functions are typed with programmer definable types using a hierarchical abstract type system. The types are also used for defining user interaction behavior, validity checking of user input, etc..

### 2.4 Forms

Syntel uses the **business form** as its basic display metaphor. All user interactions involve **display objects**, which include screens, primitive boxes that display single values, and groups of display objects. The display objects and the links that connect them to the value table instances are specified by the knowledge engineer using a non-procedural screen definition language. The end user is restricted to the set of display objects that are currently visible, which is determined non-procedurally by Syntel functions.

### 3 Control Strategies

In general, the equation network defines functional constraints among the values in value tables. All of these constraints are satisfied initially, and the primary control task is to make sure that these constraints continue to be satisfied. After the end user has entered one or several new values, the inference engine
must update all dependent values and refresh information presented to the user. As with all data-driven systems, this requirement can lead to serious efficiency problems, especially since we are dealing with thousands of value tables up to 120 levels deep.

The three principal techniques we use to achieve efficient data-driven operation are called breadth-first propagation, incremental propagation, and screen-limited propagation. In addition we use demand-driven propagation when applied for. All of these techniques are motivated by the principle of minimizing the amount of recomputation done in response to changes.

3.1 Breadth-first propagation
The equation network is a directed acyclic graph. From any set of input nodes, there are typically multiple paths through the network that eventually terminates in one or more output nodes. The propagation is performed breadth-first through the network in a bottom-up fashion, using a compile time partial sorting of the equations based on their distance to a root node. The breadth-first propagation guarantees that a function never will be recomputed more than once in a given round of propagation.

3.2 Incremental propagation
The value tables are normally parameterized and propagation often creates or deletes new instances as well as changes their values. The simplest way to propagate changes in instances of value tables would be to recompute the entire value tables for each change, performing the joins over and over, which would be extremely expensive. Instead, we use a form of differential evaluation [12,14] of joins, a form of continuous view materialization [7], where we continuously maintain the join of the right hand side value tables of each equation.

3.3 Screen-limited propagation
A third way to apply the principle of updating those and only those items that need to be updated is to exploit the fact that at any given time the user can only see what is visible on the screen. After the user has made changes, the only value tables that need to be recomputed are those that influence the values visible on the current screen. By a compile time analysis of how each screen is linked to the value tables, we can compute for each screen the set of value tables that influence the screen. The propagation algorithm is optimized so that only instances of those value tables are recomputed that are member of this set for the currently displayed screen, which results in screen-limited propagation. [7] also propose a static pruning as a way to optimize monitoring of data.
3.4 Demand-driven propagation

We use demand driven, top-down propagation for data that is retrieved by demand only. For example, dynamic explanation texts can be specified using Syntax equations. The explanation text is normally retrieved by an explicit demand by the user, and thus functions used only to construct explanation texts are propagated demand-driven.

3.5 Alerts

It is often important that the system be able to alert the user when certain special or abnormal conditions exist. We call such a triggered message an alert. The normal computational mechanisms can be used to test for the alert condition, and to concatenate strings to compose the alert message. Thus, alerts are actually treated as strings and the general data-driven evaluation makes the implementation of alert messages almost trivial.

4 Summary and conclusions

Active functional systems provide a data-driven, functional approach to integrating database and expert systems. The uniform view of representing data as extensional functions and representing knowledge as derived functions leads to great conceptual simplicity. The view that prior distributions are generalized default values, and the incorporation of prior instances into the basic design, provides a clear, uniform method for handling unknown and inexact values.

The fact that the system is completely non-procedural yields several advantages. End users can enter, change or override data in any order. ‘What-if’ experimentation is easily supported. Explanation and alerts are facilitated because outputs depend only on the values of user inputs, not on sequence or side effects. Knowledge engineers specify the relations between variables without concern for control issues and thus the knowledge base can be specified and designed from a data flow standpoint [6].

We have described several techniques that were used to improve efficiency. In particular, program differentiation and screen-limited propagation proved effective in limiting recomputation to those and only those items that must be recomputed.

References


