A FUNCTIONAL APPROACH TO INTEGRATING DATABASE AND EXPERT SYSTEMS

A new system architecture shares certain characteristics with database systems, expert systems, functional programming languages, and spreadsheet systems, but is very different from any of these.

TORE RISCH, RENÉ REBOH, PETER HART, and RICHARD DUDA

Advances in computing sometimes flow from the recognition of fundamental similarities among disparate theories or system architectures. Once recognized, these similarities can be exploited to design a new architecture that combines selected aspects of its predecessors, opens new areas of applications, and may even lead to new insights into underlying theories.

A new system architecture, called an *active functional system*, shares certain characteristics with database systems, expert systems, functional programming languages, and spreadsheet systems, but is very different from any of these. It is based on a uniform—one might even say rigid—use of side effect–free functions that represent facts and knowledge in a nonprocedural programming system. Database objects are represented by arbitrary extensional functions, i.e., tables, while domain knowledge is represented by side effect–free intensional functions composed from a suitable library. Both default and inexact information are accommodated by treating values of database objects as random variables with associated probability distributions. The uniformity that results from functional representations leads to a corresponding uniformity in database and knowledge-base operations. We call the system "active" because it is data driven; more specifically, changes in the distributions of the factual input data are propagated through the knowledge base to update the distributions of the derived output data. These concepts are embodied in the Syntel® programming system [23], which has been fully implemented and in commercial use since mid-1986.

Syntel is a registered trademark of Syntelligence, Inc.

DATA-INTENSIVE AND KNOWLEDGE-INTENSIVE PROBLEMS

Database and Expert Systems

In recent years the complementary nature of database and knowledge-base applications has been recognized. Database applications typically involve a relatively small number of relations or files holding a large number of records; relatively simple representational structures are used to store voluminous amounts of data. By contrast, expert systems usually involve a large number of relations (or rules, frames, etc.), each holding a small amount of judgmental knowledge; complex representational structures are used to store moderate amounts of data.

Inference styles and methods are also complementary. Database inference is usually restricted to either precise boolean combinations of modest complexity or simple numerical comparisons. Expert systems often employ complicated reasoning procedures, frequently involving incomplete or uncertain numerical and symbolic data. In addition, the factors that are relevant and the sequence in which they are considered usually varies greatly from run to run.

The complementarity of database and expert systems has led to considerable interest in architectures that combine the two [5, 17, 34]. The most straightforward synthesis is to create a *loosely coupled* system in which the database module acts as a server for the expert system module [1, 20]. This architecture allows reference data to be freely accessed by the expert system module while the database system acts as a case server. However, some important families of applications place equal demands on expert system and database system

facilities, rendering a loosely coupled design inappropriate. Instead, a tightly coupled, unified approach would be more desirable. The following example illustrates this approach.

A Financial Application

Many financial applications call for a combination of database and expert systems methods. An example from banking involves evaluating the credit risk of a firm that wishes to borrow money. A banker engaged in this task—and therefore a system that provides decision support to the banker—will need to deal with a number of issues that are common to most financial applications.

First, both quantitative and qualitative data must be analyzed to assess the firm's ability to repay a loan. Some of this data, like financial statements, occurs naturally in tabular form, but other data, like information relating to quality of management or competitive position, is not so regularly structured. The amount of tabular data typically varies from several hundred case-specific items to several hundred thousand reference-data items. Case-specific, non-tabular data is comparable in size to case-specific tabular data. Taken together, the amount of data accessed by a single user is likely to be large enough to require a significant data management facility.

Bankers operate in a world of business and financial forms, so it seems obvious that the user interface of the system should emulate this familiar milieu. A banker using the system will insist on retaining the initiative in any interactive dialog, as spreadsheet systems allow; the sequential "question-and-answer" mode characteristic of goal-directed expert systems is unlikely to be acceptable. We can also anticipate a need to support repeated "what-if" explorations, and be able to restore the system to the state it was in before the what-ifs were performed. Finally, the banker is a professional who expects to use the system as a tool; among other things, this implies that the end user must be able to override any system-generated output without introducing logical inconsistencies.

Assuming we have satisfied the requirements for dealing with considerable amounts of case-specific and reference data, and have provided the user with a familiar and usable interface, we still need to support the style of reasoning appropriate to this application domain. It is clear at the outset that quantitative and qualitative reasoning are tightly integrated; in other words, there is no obvious, clean partition between "the numbers side" and "the subjective side" of the task. Typically, the analysis proceeds in response to the particular situation and does not follow any fixed sequence. Understanding how various elements of an analysis are *related* is more important than knowing an optimal analysis sequence.[1] Finally, credit assessment is inherently imprecise. Notwithstanding the substantial amount of information that bankers receive about a borrower, there inevitably are gaps, leading to the need for inexact reasoning.

[1] For example, an important piece of judgmental knowledge in this domain might relate the marketing strength of a company, the maturity of the market it is in, and the business strategy of the company. The knowledge lies primarily in understanding this relation, rather than in deciding when the relation should be used during the analysis process.

A UNIFYING DESIGN

Five design elements that unify the database and expert systems components need to be considered. First, we need to design to the data model for both case-specific and reference data. Second, we need to define a set of primitive operations over the data objects that are sufficient to express domain knowledge. Third, we need a means for expressing and reasoning about inexact data and knowledge. Fourth, we need a user interface that replicates the world of business forms, and that allows the user to retain the initiative. Fifth, we need an interpreter (or in expert systems terms, an *inference engine*) that governs the flow of control.

Before proceeding with details, we should at least mention the general criteria we were seeking to satisfy. We anticipated that we would be dealing with volumes of data that are moderate by database standards but large by expert system standards. Preliminary studies also suggested that the required inferences would be moderately complex by expert system standards and very complex by database standards. We expected, and later proved, that we would have to define relations (whether by rules, functions, etc.) among several thousand decision variables or factors. Our design choices were made with these considerations in mind, keeping an eye toward elegance. Specifically, we hoped that it would be possible to design a tightly integrated interface between the expert system and database components; indeed, we hoped to eliminate the distinction between these two components.

Extensional Functions

We use extensional functions or *value tables* to represent both changing, case-specific data and persistent reference data. A value table contains any number of columns with independent keys, and exactly one non-key column. Accordingly, a value table can be regarded as a relation with a single non-key column. From a functional point of view, a value table holds an extensional definition of a function. The name of that extensional function is the name of the value table. For example, Figure 1 gives the value table `Revenue` for several different years and states; functionally, it shows the extensional function `Revenue[Year, State]`. Depending on whether we are thinking of a value table as a relation or as a function, we refer to `Year` and `State` either as *keys* or as *formal parameters*. A single row of the table is an *instance*, and the part of the row containing just the parameter values is the *parameter instance*. Thus, in Figure 1, the tuple

```
(1987  'AZ'  $10,000)
```

is an instance of `Revenue` and the tuple `(1987 'AZ')` is the corresponding parameter instance.

Actually, this description of value tables is slightly oversimplified. The values stored in the non-key col-

Year	State	→	Revenue
1986	"CA"		$50,000
1987	"AZ"		$10,000
1987	"CA"		$60,000

FIGURE 1. A Value Table with Keys *Year* and *State*, and Value *Revenue*

umn need not be exact, but can be probability distributions, and a value table also includes additional information, such as whether or not the value has been overridden or footnoted; this information can be thought of as being stored in additional non-key columns. Basically, however, value tables are extensional functions.

Value tables are also typed, that is, both the keys and the value must have a type, which can be a system-defined type (such as `String`, `PosNumber`, or `Dollar`) or a knowledge-engineer-defined type. In addition to protecting against programming errors, the type system supports input/output by providing validity checks, system-generated selection menus, and flexible formatting.[2]

A parameter or a value can be the symbol `NIL` that means that the instance exists, but that its value is currently *undefined.* If a key in the value table is `NIL`, the associated value is the *default* value which is used if no exactly matching instance exists. When all the keys are `NIL` the default value is called the *prior value.* Figure 2 shows a value table containing undefined and default values. The instance with the parameter instance (`1987 'AZ'`) exists, but its value is undefined. In 1986, for any state other than `'CA'` the default value is `$1,000`; for any year except 1986 and 1987, the default value is `$500` in `'CA'` and `$0` (the prior value) in all other states.

Year	State	→	Revenue
1986	"CA"		$50,000
1986	NIL		$ 1,000
1987	"AZ"		NIL
1987	"CA"		$60,000
NIL	"CA"		$ 500
NIL	NIL		$ 0

FIGURE 2. A Value Table Containing Undefined and Default Values

Unlike standard database systems, Syntel allows the programmer to specify default values of extensional functions to avoid null values in database operations. Syntel combines null values with complete or partial default values, and dynamically maintains correct default values for derived value tables. (See [11] for a proposal for the specification of static default values for each relational column.)

[2] The type system also includes an inheritance hierarchy not discussed in this article.

Value tables can store symbolic as well as numeric data. A knowledge base serving an end user like our banker may well require 1,000 or 2,000 value tables to represent qualitative and quantitative information about one case. Although some of the value tables contain entered data describing a case, most hold derived data that is maintained by the inference engine. These tables constitute a personal copy of case-specific data when the case is loaded, and are stored centrally when the case is saved.

Intensional Functions and Equation Networks

Syntel uses a family of side effect–free primitive functions as the single, uniform representation for operations on value tables. Mathematically, each primitive function is an intensionally-defined mapping from one or more value tables into a single value table.[3] Programmatically, the value of a function depends solely on the values of its arguments; furthermore, no primitive function can directly determine the flow of control. For these reasons, Syntel is a pure functional or applicative language.

Primitive functions provide the basic means for computing derived value tables—value tables not obtained from the user or from other external sources. For example, we might want to define a derived value table for `Income` given value tables for `Revenue` and `Cost`:

```
Income[Year, State]
    ← Difference(Revenue[Year, State],
                 Cost[Year, State]).   (1)
```

This equation is not an assignment statement, but is strictly definitional. It expresses the extensional function `Income` as an intensional function `Difference` of the two extensional functions `Revenue` and `Cost`. It also implies a direction of dataflow. In particular, whenever the value of an instance of either `Revenue` or `Cost` changes, this equation calls for a change in the value of the corresponding instance of `Income`. Were an instance of `Income` to be changed in some other way (through a user override, for example), no changes would be made to corresponding instances of `Revenue` or `Cost`.[4]

Since Syntel functions are referentially transparent, they can be composed to arbitrary depth. We might, for example, wish to define `NetIncome` as `Income` less `Overhead`:

[3] Since Syntel employs both extensionally and intensionally defined functions, there is considerable opportunity for confusion. We use the term *value table* for the former and shorten the latter to *function.* We consistently use the term *formal parameter* to refer to an argument of an extensional function—i.e., value table—when we wish to emphasize the functional view; we use the term *key* to refer to an argument of a value table to emphasize the database view; and we use the term *argument* to refer to the argument of an intensional function. It is important to bear in mind, however, that the principal objects mathematically are functions.

[4] In principle, constraint satisfaction techniques could be used to relax this rather strong restriction on dataflow direction [32]. Syntel employs probabilistic mechanisms to accommodate uncertainty. The combination of general constraint analysis with uncertainty would greatly increase overall system complexity and severely impact run-time efficiency.

```
NetIncome[Year, State]
    ← Difference(Income[Year, State],
                 Overhead[Year, State]),   (2)
```

where Income is defined by (1). Syntel functions may be self-referencing (i.e., recursive). For example, Syntel can estimate the revenues for some year, given the revenues of a previous year. Aside from this self-referencing aspect, functional composition in Syntel can be represented graphically as a directed acyclic graph called an *equation network*. A node in an equation network represents a single value table and the associated function that computes it. Arcs point to the associated function from its arguments. From a database standpoint, an equation network is comparable to a database schema. Figure 3 shows the simple net corresponding to Equations (1) and (2) in which nodes correspond to value tables and derived nodes (such as NetIncome and Income) are intensional functions of argument nodes. Names of the value tables are shown above the names of the intensional functions.

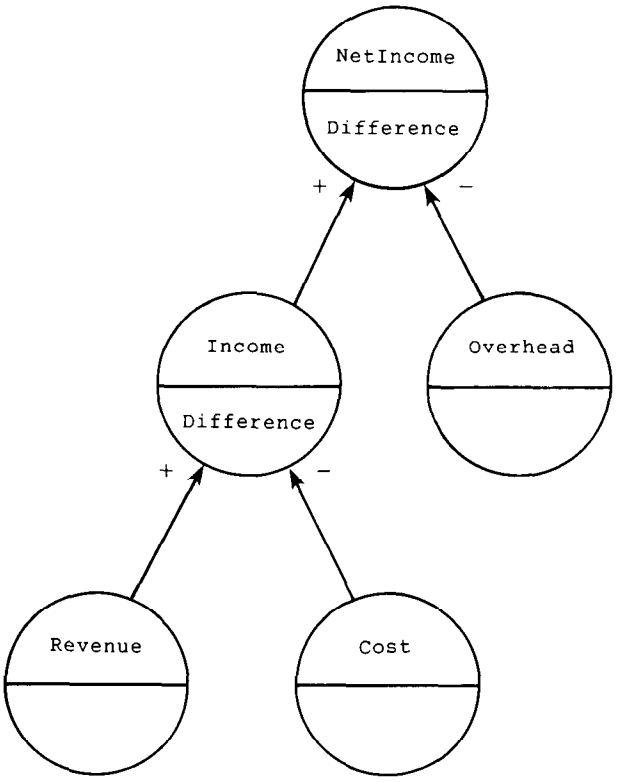

FIGURE 3. A Simple Equation Network

Intensional Functions and Judgmental Knowledge

The Syntel system provides the knowledge engineer with 65 primitive intensional functions. In addition to a variety of numerical, logical and string-manipulation functions, there are special families of functions for transforming and aggregating value tables and representing expert judgement. Here we discuss a central problem in expert systems, the representation of judgmental knowledge.

Representing Judgmental Knowledge

Developers of expert systems draw a distinction between classification (or diagnosis) problems and assessment (or estimation) problems. "Classification" emphasizes selecting an entity from among a finite number of alternatives. On the other hand, assessment emphasizes assigning a value on some continuum to a variable. The IF-THEN production rule, first used in expert systems in Mycin [27], continues to be the most widely used means for representing judgmental classification knowledge, and Syntel provides a primitive intensional function of equivalent expressiveness.

Unfortunately, IF-THEN rules are poorly suited to expressing the judgmental assessments that are a critical part of financial expertise because the IF-clause cannot combine sub-assessments into an overall assessment. Since each sub-assessment is by definition a point on a continuum, many rules are required to specify the conclusions to be drawn under various combinations of their values.

As a simplified example, the overall credit assessment of a firm might depend upon individual assessments of the financial strength of the firm, its market strength, and its management strength. Financial experts are skilled at performing "apples-to-oranges" evaluations of this type and can describe how the individual factors are to be weighed. Therefore, a functional representation that expresses relative weights, rather than a rule-based one that emphasizes selection, is clearly more natural and economical. Syntel provides a primitive function that first maps sub-assessments into a common continuum called *votes* and then sums them:

$$\text{OverallAssessment} \leftarrow V_0 + V_1(\text{Assessment}_1) + \ldots + V_n(\text{Assessment}_n).$$

The voting functions V_i are piecewise linear in form, and are specified in detail by the knowledge engineer. Summation presumes independence of individual assessments. The special case in which the voting functions are linear arises frequently in statistical inference as linear regression.

Voting functions are widely used in Syntel knowledge bases to compute assessments and dovetail smoothly with IF-THEN functions which are typically used to identify and select analytical sub-cases. In that role they are far more compact than a comparable rule-based representation would be, facilitating knowledge base design and maintenance and leading to perspicuous machine-generated explanations of reasoning.

While Syntel voting functions are an important example of why functional representations are a natural choice for solving a broad class of estimation problems, they are not the only such example. More generally, functional representations support a rich family of methods that allow knowledge engineers to combine not only predicates—as is often done in rule-based

representations—but also to combine arbitrary, nonboolean variables. Rule-based systems, in contrast, lack this intrinsic richness and typically rely on "procedural escapes" to perform comparable operations. Such escapes complicate the introduction of uniform mechanisms for dealing with inexact reasoning about nonboolean quantities. As we show next, functional representations are readily extended to provide this critical capacity.

Inexact Reasoning

As already noted, applications like credit evaluation introduce inexactness in two principal ways: missing data and imperfect knowledge. To deal with the inexactness that results from missing data, we generalize the value from the exact and default values already mentioned to include probability distributions as well. In addition, Syntel provides limited support for representing imperfect knowledge through special intensional functions for manipulating probability distributions.

Year	State →	Revenue
1986	"CA"	$50,000
1987	"AZ"	$10,000
1987	"CA"	$60,000
1988	"AZ"	$11,000
1988	"CA"	($55,000, $1,000)
1988	"NY"	NIL
1988	NIL	($25,000, $3,000)

FIGURE 4. A Value Table that Includes Inexact Data

For a simple illustration of how probability distributions can account for missing data, let us expand the value table in Figure 1 to hold projected 1988 revenues in addition to the recorded historical revenues. Suppose that the system was used in 1987 to estimate projected 1988 revenues. Since projections are by nature uncertain, we may wish to represent the projected revenues by a mean and a standard deviation, as shown in Figure 4. The first number in a list is the mean and the second the standard deviation of a probability distribution. This figure indicates that the projected 1988 revenue for Arizona is exactly $11,000. For California, the distribution of projected 1988 revenue has a mean of $55,000 and a standard deviation of $1,000. For New York, the value (or distribution) of 1988 revenue is undefined. Finally, for all states other than Arizona, California, and New York, the default distribution of projected 1988 revenue has a mean of $25,000 and a standard deviation of $3,000. This represents the *prior distribution* of revenue for the year 1988.

The introduction of probability distributions considerably complicates the notion of function evaluation. Now the value of each instance of a derived value table is a probability distribution that depends on the distributions of the corresponding instances of its arguments. In general, the computation of derived distributions (often referred to as the propagation of probability distributions) cannot be computed from the separate distributions for the arguments without rarely-available information about joint distributions. Even when this information is available, the required computations are often very costly. However, we can make the situation tractable by making several simplifying assumptions.

Typically the arguments of the function are assumed to be *statistically independent.* Thus, the joint probability distribution is merely the product of the distributions for the arguments. The situation can be further simplified if we agree that it is unnecessary to describe probability distributions completely. In Figure 4, the distributions over Revenue are specified only by their means and standard deviations; in general, we represent distributions over continuous values by these second-order statistics. Distributions over discrete values are represented by the complete probability mass function.[5]

A fuller treatment of these issues is given in [13], which includes a discussion of the statistical independence assumption from a knowledge engineering viewpoint. From the system viewpoint, these assumptions and simplifications allow the system to compute the distributions of the values in the derived value tables from the distributions of the arguments. A different method must be used for each of the 65 primitive functions. For a primitive function like `Plus`, the method for computing derived distributions is elementary. For most other primitive functions the methods are more complex. The `Equal` predicate, for example, requires a comparison of the "closeness" of two probability distributions. In general, we have used the formal theory of functions of random variables as a guide [24], implementing exact solutions where feasible and approximate solutions otherwise. The level of difficulty often encountered in these computations is the chief obstacle to allowing knowledge engineers to enlarge the set of primitive functions.

The most important use of probability distributions is to represent default knowledge. To return to Figure 4, suppose we needed to know the 1988 revenue for Texas. Given no further information, the bottom row shows (by default) that the prior distribution has a mean of $25,000 and a standard deviation of $3,000. This distribution would be used until new information becomes available, perhaps from an informed user or possibly computed elsewhere in the equation network.

Semantics of Equations

Syntel equations always have the form `LHS ← RHS`. From the knowledge engineer's viewpoint, these equations define how instances of a single value table on the left-hand side are derived from instances of one or more value tables on the right-hand side. However, the system must also account for the possibility of values being entered (or overridden) by the end-user, being

[5] Discrete values frequently are used to represent assessments whose values might be, e.g., any of {`Poor, Fair, Average, Good, Excellent`}. The small number of possible values makes using the complete distribution feasible.

defaulted to prior distributions, or being left undefined. The system *always* computes the actual value of an instance of an LHS value table as follows:

```
If  user-entered-value ≠ NIL
  then user-entered-value
elseif RHS ≠ NIL
  then RHS
else prior-distribution
```

This defines the precedence of user-entered values, derived values, and default values. In the remainder of this section we focus on the derived values, without forgetting that there are two other ways that a value table can obtain its values.

Two things have to be considered when computing the contents of a value table on the left-hand side of an equation given a set of value tables on the right-hand side:

1. How is the value of each instance computed? As discussed earlier, each primitive function has its own method; e.g., `Plus` adds the values of each instance.
2. What are the relevant instances of the LHS value table? For most primitive functions (except a few structure-transforming ones) it is determined by a new kind of join, called the *default join*, of the value tables on the RHS.

The Default Join

Roughly speaking, the keys of an LHS value table are the natural join of the keys of the RHS value tables, while the values are determined by the particular primitive function. However, as others have noted [11], the conventional natural join can lose data in certain circumstances. Specifically, if the RHS value table contains a key that is matched by no other RHS value-table key, that instance will be lost. To overcome this problem, various kinds of "outer joins" that put null values in non-matching columns of joined relations have been proposed [10, 11].

The default join performs a loss-free join over the RHS value tables while, in contrast with outer joins, maintaining the correct default values of the LHS table given the default values of the RHS value tables. Its semantics resembles a natural join with the equality operator replaced by a match operator. To describe the semantics we use functional language rather than relational database terminology, primarily because the need for the default join arises from the requirements of computing-derived value tables.

Suppose that X and Y are two value tables (extensional functions), and let FP_X and FP_Y be the sets of formal parameters for X and Y, respectively. For example, for X[*I*] and Y[*I*, *J*] we have $FP_X = \{I\}$ and $FP_Y = \{I, J\}$. We always assume that these parameters are *free*, i.e., that they can be assigned any value that is present in their columns in the value tables. Given a set of parameter instances of X and a set of parameter instances of Y, we want to define a meaningful set of parameter instances of an arbitrary function of X and Y. For example, suppose that we want to form the sum Z[*I*, *J*] ← X[*I*] + Y[*I*, *J*], and that the following instances of X and Y are defined:

I	→	X
7		14
NIL		−1

I	J	→	Y
7	a		3
NIL	NIL		11

Two instances of Z are obvious: Z[7, a] = 14 + 3 = 17 and Z[NIL, NIL] = −1 + 11 = 10. But sometimes we know that *I* is 7 (and hence that X = 14) when we don't know *J*; since we have a default value for Y, we can also include the instance Z[7, NIL] = 14 + 11 = 25.

Another way to consider this situation is to imagine that X is actually parameterized by both *I* and *J*, but it just happens that we never know a value for *J*. Then both X and Y can be thought of as being identically parameterized, with value tables as shown below:

I	J	→	X
7	NIL		14
NIL	NIL		−1

I	J	→	Y
7	a		3
NIL	NIL		11

→

I	J	→	Z
7	a		17
7	NIL		25
NIL	NIL		10

Instead of talking about knowing or not knowing parameter values and using defaults, we can say that the *parameter-instance pattern* (7 NIL) in X matches the parameter instance (7 a) in Y to yield the parameter instance (7 a) in Z; similarly, (7 NIL) matches (NIL NIL) to yield (7 NIL); (NIL NIL) matches (7 a) to again yield (7 a); and (NIL NIL) matches (NIL NIL) to yield (NIL NIL). The set of parameter instances of Z is formed from these results.

This line of thought can be generalized and formalized as follows. Let Z be a combination of two extensional functions X and Y, so that the set FP_Z of the n formal parameters for Z is the union of FP_X and FP_Y. Define the *extension* of X (denoted by $\overline{X}$) as the value table formed from X by appending the parameters unique to Y and using NIL for their parameter values. Let the extension of Y be similarly defined, so that both $\overline{X}$ and $\overline{Y}$ have n identical formal parameters. With no loss in generality, assume that the corresponding parameters in $\overline{X}$ and $\overline{Y}$ appear in the same column positions. Let $P_{\overline{X}}$ be a parameter instance of $\overline{X}$ and $P_{\overline{Y}}$ be a parameter instance of $\overline{Y}$, with

$$P_{\overline{X}} = (p\bar{x}_1 \ldots p\bar{x}_n)$$

and

$$P_{\overline{Y}} = (p\bar{y}_1 \ldots p\bar{y}_n).$$

We treat NIL as a wild card in matching $P_{\overline{X}}$ and $P_{\overline{Y}}$. To be specific, we say that $P_{\overline{X}}$ *matches* $P_{\overline{Y}}$ if for every i either (a) $p\bar{x}_i = p\bar{y}_i$, (b) $p\bar{x}_i$ = NIL, or (c) $p\bar{y}_i$ = NIL.

If $P_{\bar{X}}$ matches $P_{\bar{Y}}$, the resulting parameter instance $P_Z = (pz_1 \ldots pz_n)$ has

$$pz_i = \begin{cases} p\tilde{x}_i & \text{if } p\tilde{x}_i = p\tilde{y}_i \quad \text{or } p\tilde{y}_i = \text{NIL} \\ p\tilde{y}_i & \text{if } p\tilde{x}_i = \text{NIL}. \end{cases}$$

The set of parameter instances for Z is found by removing duplicates from the results of matching all of the pairs $(P_{\bar{X}}, P_{\bar{Y}})$ of parameter instances from $\bar{X}$ and $\bar{Y}$.

The duplicates that result from different ways to match parameter instances of $\bar{X}$ and $\bar{Y}$ do not affect the default join, but when we want to compute a value for Z we must select specific values from X and Y to combine. In our example where both (7 NIL) and (NIL NIL) in $\bar{X}$ match (7 a) in $\bar{Y}$ to yield (7 a), it is clear that we want to use the value of $\bar{X}$ associated with the more specific parameter instance (7 NIL) rather than the complete default value associated with (NIL NIL). In general, when more than one pair of instances match, we want to combine the values associated with the most specific pair. If $(P_{\bar{X}1}, P_{\bar{Y}1})$ and $(P_{\bar{X}2}, P_{\bar{Y}2})$ both match to yield the same P_Z, we say that $(P_{\bar{X}1}, P_{\bar{Y}1})$ is more specific than $(P_{\bar{X}2}, P_{\bar{Y}2})$ if there are fewer NILs in the first match.[6]

The following examples illustrate the properties of the default join. Consider first the equation

```
TotalIncome[Year]
    ← Difference(TotalRevenue[Year],
                 TotalCost[Year]).
```

Since we always include the prior instances, at any given time, there will be one or more instances of TotalRevenue and one or more instances of TotalCost. The parameter instances of TotalIncome are the default join of TotalRevenue and TotalCost. The corresponding values come, of course, from differencing the most specific instances of TotalRevenue and TotalCost:

Year	→ TotalRevenue
1985	$60,000
1986	$75,000
NIL	$0

Year	→ TotalCost
1984	$10,000
1985	$12,000
NIL	($5,000, $2,000)

Year	→ TotalIncome
1984	−$10,000
1985	$48,000
1986	($70,000, $2,000)
NIL	(−$5,000, $2,000)

[6] When there is more than one pair with the minimum number of NILs, the choice is arbitrary; if we let P_Z correspond to a binary number N in which a bit is one where a NIL was involved in the match, then the current implementation prefers the pair with the smallest value for N.

Note that there are instances of TotalIncome for 1984, 1985 and 1986, although there is no 1984 instance of TotalRevenue and no 1986 instance of TotalCost. Had there been no default value for TotalRevenue, the value of TotalIncome would have been undefined (NIL) for 1984, but the instance would still exist. Also, the combination of missing 1986 data and a prior distribution for TotalCost results in a distribution for the 1986 instance of TotalIncome; this is typical of the way that prior distributions substitute for missing data and introduce inexactness into derived results.

In general, propagation occurs whenever an instance is created, a value is modified, or an instance is deleted. Thus, if the user were to create a 1987 instance of TotalRevenue, the system would also create a 1987 instance of TotalIncome. Deleting the 1985 instance of TotalRevenue would change the value of the 1985 instance of TotalIncome from $48,000 to −$12,000; if the 1985 instance of TotalCost were also deleted, the system would delete the 1985 instance of TotalIncome. Accordingly, the number of rows of a value table changes dynamically at run time. Note that all value tables have at least one row, which holds the prior value.

The second example illustrates the combination of value tables that are differently parameterized:

```
W[Year, State]← Plus(X[],
    Difference(Y[Year], Z[State])).
```

X
1

Year	→ Y
1984	5
NIL	10

State	→ Z
'AZ'	4
'CA'	9
NIL	2

Year	State	→ W
1984	'AZ'	2
1984	'CA'	−3
1984	NIL	4
NIL	'AZ'	7
NIL	'CA'	2
NIL	NIL	9

In general, the presence of multiple disjoint parameters leads to the cartesian product of the component instances. While this example shows that the dimensionality of the results can be increased, the dimensionality of a value table can also be reduced. For example, if Revenue is parameterized by *Year* and *State*, then Revenue[*Year*, 'CA'] is parameterized by *Year* only. Also, several intensional functions such as Sum and Max aggregate over value tables to produce results of lower dimensionality.

The User Interface

In the Syntel system, the user interface not only plays the conventional role of handling communication with the end user, but controls the process of computing derived value tables as well. Syntel uses the business form as its basic display metaphor. All user interface actions involve display objects, which include primi-

tives values, text strings, forms, and groups of display objects. The display objects and the links that connect them to objects in the equation network are specified by the knowledge engineer using a nonprocedural forms language.[7]

Since the system is data driven, the end user is free to go to any form, and to view, enter, modify or delete any data items in it. However, this can be bewildering when there are hundreds of possible forms, most of which might be relevant only in particular situations. To solve this problem, Syntel allows display objects to be *conditionally visible.* The knowledge engineer can use predicate nodes in the equation network to control the visibility of a part of a form, an entire form, or an arbitrary set of forms. By restricting the user to the set of forms and display objects that are relevant, the system provides much of the guidance and focus of a goal-driven system while still providing data-driven operation.

The example business form shown in Figure 5 deals with the business environment of a prospective borrower. The end user can enter one or a sequence of values either through the keyboard or through choice menus that appear when an active region is cursor-selected. Outputs computed by the system can be displayed symbolically or graphically as shaded bars called *meters.* Output meters for two inexact assessments are shown in Figure 5. The position of the dark bar indicates on the minus-to-plus scale the degree to which the assessment is favorable. The width of the bar indicates the uncertainty in the assessment and provides a graphical expression of the underlying probability distribution.

The items shown in Figure 5 are particularly straightforward because none of them is parameterized; the corresponding value tables have only a single instance, and it is not necessary to specify the instance to be displayed. Figure 6 shows a form dealing with the cash available to cover debt payments. The boxes on this form are more typical, because most of them are drawn from instances of parameterized value tables. The parameters for the dollar amounts shown are `StatementType (ANNUAL)`, and `Date (December-1984)`. The values entered for these parameters determine which of the many existing instances will be displayed.

[7] The details of this language, which roughly resembles document formatting languages such as SCRIBE or TEX, are beyond the scope of this article.

```
SCREEN          FILE          EXIT          MODE
INDEX           COMMANDS      FILE          EVALUATE

Screen overview: [ ] F      Business Environment                Page 4

Industry:        [MANUFACTURING]        Asset size:  [10-50MM]
Industry group:  [FABRICATED METAL PRODUCTS]
SIC category:    [HAND AND EDGE TOOLS, NOT OTHERWISE CLASSIFIED]
SIC code         [3423]
Business type:   [NATIONAL]

Industry stage:          [MATURE]
Competitive structure:   [MONOPOLISTIC]
Competitors trend:       [STABLE]

Competitive environment  - [ ][ ][ ][■][ ][ ][ ] +

Industry cyclicality:    [AVERAGE] F
Product positioning:     [ ]
Threat of substitutes:   [ ]
Regulatory risk:         [ ]

Industry risk            - [ ][▒][■][▒][ ] +

Industry net sales trend:      8.0%

03/26/1987    02:33 PM    Neal Herman
```

FIGURE 5. An Example Form

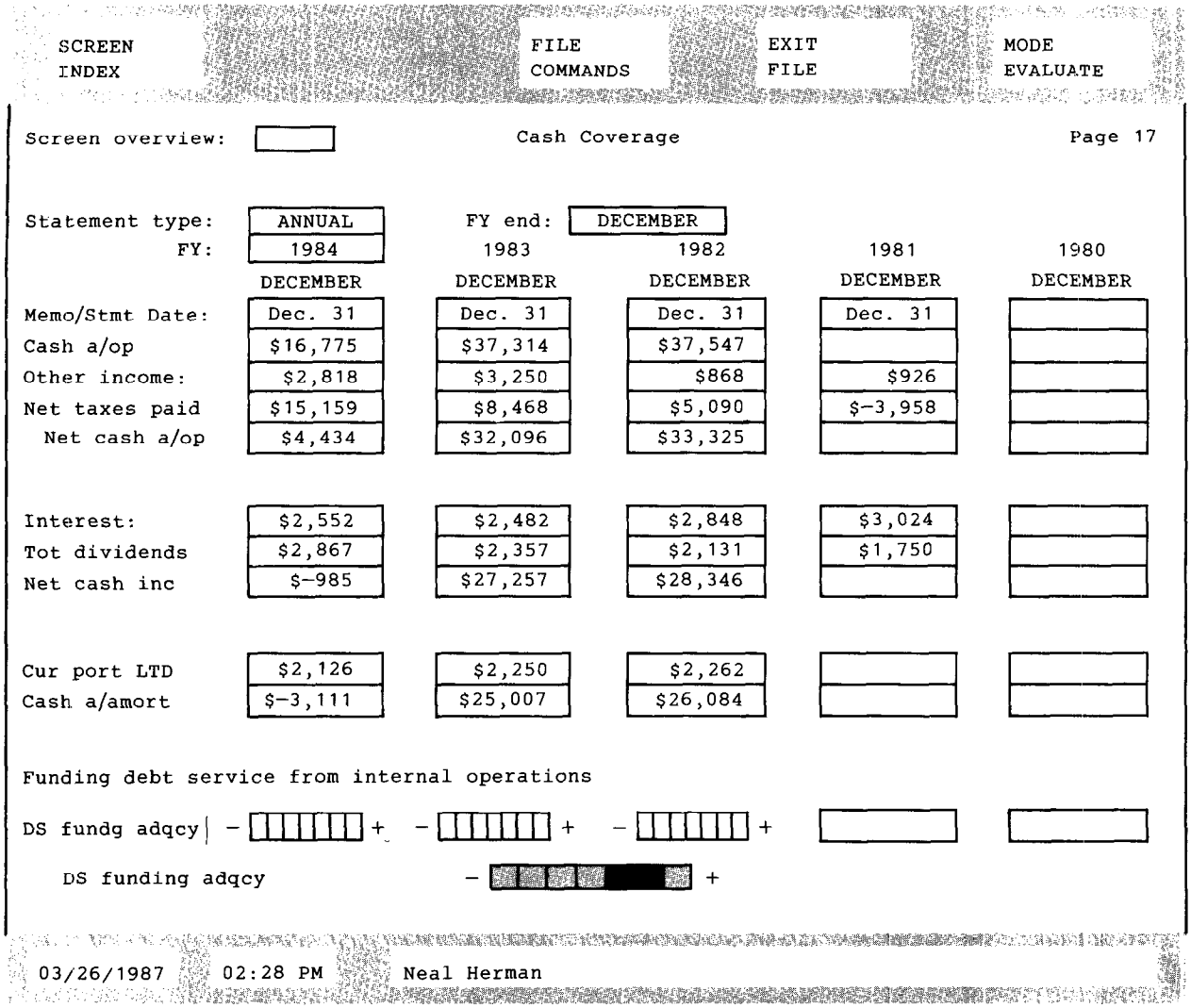

FIGURE 6. Display of Parameterized Value Tables

In general, the visual appearance of the form is specified by a layout description containing information about the size and place of boxes and text. The behavior of each box, and of the form itself, is determined by bidirectional *links* to nodes in the equation network. Each link connects a specific box on a form to a dynamic subset of instances of some value table determined by selectors; the connection is represented by an extensional function expression. For example, in Figure 6 the input box labelled "`Interest:`" is linked to the expression

```
Interest[SelectedStatementType,
         SelectedFiscalYear].
```

The nodes used as arguments in a link, `SelectedStatementType` and `SelectedFiscalYear`, are used to select the correct instance from the extensional function `Interest[`*`Type, Year`*`]`, and are therefore called *selectors*. The boxes labelled "`Statement type:`" and "`FY:`" are linked to these selectors so whenever the user enters a value for `Interest:`, the system will assert the instance of the value table `Interest` currently selected by the two selectors; conversely, the value to be displayed in a box is equal to the value of the linked functional expression.

Additional display information can be obtained by linking to other nonprimary values of value tables. For example, the user can request a clarification of box contents. The clarification string, which can be static or can be dynamically computed when requested, is another non-key column in the value table, and is accessed using its associated link. The small "`F`" in Figure 5 alongside `AVERAGE` denotes a footnote, and is an example of another class of displayable information called *annotations*. Other possible annotations include overrides (the user has directly entered a replacement for a computed value) and alerts (the system has de-

tected a condition specified by the knowledge engineer to be anomalous). Annotation information is also held in other non-key columns in the value table, and is again accessed through links.

The design of the forms system leads to a clean separation of a knowledge engineer's two principal tasks: representing and structuring domain knowledge, and designing and encoding an appropriate user interface. The technical apparatus of extensional and intensional functions (i.e., the equation network) supports the first task, while the forms system supports the second. The flexible and uniform mechanism for linking these components considerably simplifies the design and implementation of large knowledge bases.

INTEGRATION AND CONTROL

Control Issues

The principal objects in Syntel are functions and display objects. Accordingly, the principal control issues are when to compute a function and when to refresh a display object.

As we mentioned earlier, the Syntel run-time system is basically data driven. It maintains consistency between variable values by recomputing derived values in response to user-initiated changes in input values. Most display objects are refreshed immediately after the user has entered one or several input values and relinquished control. As with all data-driven systems, this can lead to serious efficiency problems. For large equation networks containing thousands of value tables with tens of thousands of instances, response times can be unacceptably long.

Typically, only a few of the output display objects will have new values when the user enters new data. Therefore, we cache all computed values, whether display or not, and reevaluate only the ones that have changed and thus need recaching. This reevaluation is done incrementally and bottom-up. The result is a form of program differentiation [21] that we call *propagation.*

It is interesting to compare propagation to the method known as *lazy evaluation* [9], a demand driven, top-down approach in which a minimal set of expressions are evaluated each time an output value is requested. By contrast, propagation is a data-driven, bottom-up approach in which computed values are saved between evaluations and the system reevaluates and recaches the differences that result from limited user updates of input values.

Syntel, however, does not use propagation exclusively. In particular, some display objects (such as dynamic explanation texts) are computed only when the user explicitly requests them, in which case, the system uses demand-driven, top-down evaluation. Each function is analyzed at compile time to determine whether or not it participates in demand-driven evaluations only, and should not be evaluated bottom-up.

As important as they are, consistency and efficiency are not the only considerations that shape the design of the inference engine. The system has to be able to control the forms or parts of forms visible to the end user. It should allow the user to override system-derived values, and later to remove any overrides as desired. It should also be able to alert the user when important situations are detected. Finally, it should support explanations of how derived values are obtained.

The three principal techniques used to achieve efficient operation—breadth-first propagation, incremental calculation of the default join, and screen-limited propagation—are used to minimize the recomputation done in response to changes.

Breadth-first, Bottom-up Propagation

As we mentioned earlier, an equation network is a directed, acyclic graph. From any input node, there are usually many paths through the network that eventually terminate in one or more output nodes. For this reason, direct breadth-first propagation will incur serious recomputation penalties. To avoid this recomputation, Syntel does a compile-time analysis of the equation network to create a partial ordering of nodes according to their *level number.* At run time, this ordering is used to maintain a *pending array* `P[L]` whose `L`th element is the set of node instances at level number `L` whose values have changed, but for which the effects of those changes have yet to be propagated. Initially, this array contains only the nodes changed by end-user input. Each time a node is propagated, it is deleted from `P[L]` and its successor nodes are added to `P[L]` if a change in their values occurred; the propagation stops if no change occurred. Starting with the largest level number and working in descending order, the inference engine propagates nodes in `P[L]` until it is empty. This results in propagation that proceeds upward level-by-level in a breadth-first fashion. Incidentally, our design heavily favors efficient run-time performance at the expense of compile-time analysis, just the reverse of the design of typical interpreted spreadsheets. The same compile-time analysis also supports the screen-limited propagation technique described later.

Instance Propagation and the Incremental Default Join

Changing a single instance of one argument can produce multiple changes in the values of other instances. For example, suppose

$$Z[I, J] \leftarrow \mathtt{Plus}(X[I], Y[J])$$

and suppose the following instances exist:

I	→	X
7		2
NIL		−1

J	→	Y
a		3
b		7
NIL		11

	I	J	→	Z
	7	a		5
	7	b		9
→	7	NIL		13
	NIL	a		2
	NIL	b		6
	NIL	NIL		10

COMPARING SYNTEL WITH OTHER SYSTEMS

The development of Syntel drew upon work in expert systems, database systems, spreadsheet programs and nonprocedural programming languages. Perhaps the best way to place it in perspective is to compare it with well-known systems in each of these areas.

Expert Systems

Among the better known data-driven expert system shells or languages, Syntel is most closely related to Prospector/KAS [12], OPS5 [6] and Oncocin [28]. None of these systems has integrated database primitives, and none employs pure functional representation methods. Prospector's inference network resembles Syntel's equation network. In particular, it propagates distributions for propositional variables through its inference networks (its relative, Hydro, propagates more general probability distributions [12]). The propagation algorithms, based on Bayes' Rule and histogram techniques, are substantially different, however.

OPS5 resembles Syntel primarily through being a general data-driven language for expert systems, but it is based on production rules, not functions. It differs in its recognize/act architecture, high-level primitives, lack of support for inexact reasoning, different run-time features, and lack of a user interface definition facility.

Oncocin is really a medical expert system, rather than an expert system shell or language, but many of its external characteristics are similar to those of Syntel. In particular, unlike its famous ancestor Mycin [27], Oncocin is data-driven and uses a forms-oriented interface that allows end users to enter information and to override results. However, it is implemented as a rule-based rather than a functional system, has different ways of expressing inexactness by certainty factors, and entirely different control mechanisms.

Programming Languages

Viewed as a programming language, Syntel is a nonprocedural functional data-flow language [2, 25] that shares characteristics of Lucid [33] and Lucas and Risch's equation-based system [19]. Those systems, however, are both demand-driven (evaluated top-down), and thus cannot actively monitor computed values. Access-oriented programming languages like Loops [30] support monitoring through "active values." In Syntel, however, function activation is invoked uniformly by the system, rather than by programmer-provided explicit triggers.

Spreadsheets

The heavy emphasis on data-driven control makes it tempting to compare Syntel to spreadsheets, possibly the only systems in widespread use that share this control paradigm. This superficial resemblance is heightened because Syntel applications often contain arrays of numbers like the one shown in Figure 6, but concluding that Syntel and spreadsheet programs are alike would be as misleading as concluding that all demand-driven programs are alike.

The differences between Syntel and spreadsheets are many and deep. First, in contrast to spreadsheet programs, Syntel separates the user interface definition from the knowledge base defined by the equation network, allowing us to define a data model for the knowledge base unconstrained by the requirements for display. An immediate consequence is the association of a value table and its related machinery with what in a spreadsheet would be a single cell. More significantly, instances of a value table can be created or deleted at run time, a capability that in spreadsheets would be roughly comparable to creating or deleting a cell, row, or column at run time. Spreadsheets lack this ability for fundamental architectural reasons.

There are other significant differences between Syntel and spreadsheet architectures. Even advanced spreadsheets [16] cannot reason probabilistically except through Monte Carlo simulation—an extremely expensive computational approach. More importantly, spreadsheets lack a systematic, well-founded method for dealing with default data, a capability often essential for major classes of expert system applications. Also, spreadsheets lack Syntel's structured methods for drawing inferences from symbolic data.

A further critical distinction between Syntel and spreadsheets centers on the question of efficiency. Syntel knowledge bases are typically too extensive and complex to run effectively even on large mainframes unless a great deal of attention is paid to efficient execution. The extensive compile-time analysis of the knowledge base and the variety of interacting mechanisms described earlier overcome problems of scale that would overwhelm conventional interpreted spreadsheets even when supported by powerful hardware.

Databases

From a database point of view, Syntel generally fits the functional data model [14, 26, 29]. These systems also treat binary relations as functions and use the naturalness of function composition in their data manipulation languages [26, 29], but they differ from Syntel in many ways (e.g., by lack of inexact values, default values, alerts, and overrides). Where other functional models use multi-valued functions, the functions in Syntel are always single valued. Hammer and McLeod point out the importance of derived data in semantic data bases [15], and Shipman stresses the use of derived functions for this purpose [26]. However, the mechanisms in data manipulation languages like DAPLEX [26] are designed more for data extraction and manipulation than for expressing derived, inexact, nonprocedural computations.

The important problem of representing "inexact information" (i.e., unknown, uncertain, and default values) in databases has been addressed by a number of mechanisms, including static default values [11], null values [10, 11], and the use of fuzzy sets (see, for example, [22] for an entry point into that literature). Using probabilities for any of these purposes has been much less investigated. To the best of our knowledge, Syntel is the first system that combines unknown, uncertain, and default values within a uniform representation founded on formal probability theory.

Several techniques have been proposed for continuously monitoring derived data. One approach combines a global recomputation strategy with static pruning mechanisms to recalculate derived data when needed [8], but it is inefficient when the amount of monitored data is large. Another approach is to continuously maintain derived data in a functional data model by using "program differentiation," where for each data operator there are predefined differentiation operators describing what to compute when data changes [18]. A third method uses an incremental view-maintenance algorithm to maintain derived relations containing Select, Project, and Join operators [4]. Our propagation approach is a variant of these techniques, where we employ the user interface to prune the data to monitor, and where we also support default values. Finally, we should also mention the trigger mechanisms of System R [3] in which relational database manipulation code is executed when data is updated, and the POSTGRES extension of INGRES [31] that includes the RETRIEVE-ALWAYS primitive for continuously refetching derived data.

If the user changes just the one instance X[7], three instances of Z must be changed: Z[7, a], Z[7, b], and Z[7, NIL]. Furthermore, if the user creates one new instance of X or Y, several new instances of Z may result, with corresponding removals occurring upon deletion of instances.

The simplest safe way to implement the inference engine would be to completely recompute the default join and the values of all instances every time an argument instance of a function changed. While logically correct, such a solution would be extremely inefficient. Instead, an incremental form of view materialization [4, 18], limits the recomputation to only those instances that must be recomputed.

If the value of an existing instance of an argument is changed, it is straightforward to identify all of the instances of the result that must be updated: they are the instances for which the extended parameter instance of the argument matches the existing parameter instances of the result, and for which the common parameters match exactly. The problem is a bit more complicated when an instance of an argument is created or deleted, since that changes the default join. Our solution is to compute the default join incrementally. Returning to the above example, suppose that a new instance of Y is created: Y[c] = 2. The corresponding extended parameter instance (NIL c) matches two extended parameter instances in X: (7 NIL) and (NIL NIL). Thus, creating a new instance of Y leads to two new instances of Z, but none of the other instances has to be updated.

I	→	X
7		2
NIL		−1

J	→	Y
a		3
b		7
c		2
NIL		11

	I	J	→	Z
	7	a		5
	7	b		9
	7	c		4
→	7	NIL		13
	NIL	a		2
	NIL	b		6
	NIL	c		1
	NIL	NIL		10

Screen-Limited Propagation

Another way to apply the principle of minimal updating is to exploit the fact that, at any given time, the user can see only what is visible on one screen. After the user has made changes and relinquished control, the only nodes that must be updated are the ones that influence the values visible on the current screen. This observation leads to a method for minimizing computation that we call *screen-limited propagation.*

A node is said to *support* a screen if it is either (1) linked directly to the screen or (2) located on a path through the equation network between two nodes that are directly linked to a screen. Let Support(S) denote the set of nodes that support a screen S. When the user changes the values of nodes that appear on screen S and relinquishes control, the inference engine follows a modified form of the breadth-first, bottom-up propagation algorithm. The basic idea is to propagate a node in the pending array if and only if it is in Support(S).

Because most of the nodes in the pending array are not in Support(S), we actually maintain two pending arrays, a local array for nodes in Support(S) and a global array for all other nodes.[8] Thus, when propagation of nodes in the local array stops, all of the nodes that appear on S will have been updated, but the global array will usually contain unpropagated nodes that support other screens. When the user decides to view another screen S', the inference engine moves the nodes in the global array that support S' into the local array and propagates them. Although this may result in some delay before the next screen can be viewed, it cannot increase (and it usually dramatically decreases) the total time required for propagation.

Overrides

Because Syntel functions are side effect–free it is easy for the user to override any derived values without introducing logical problems. Recall that user-entered values, if present, always take precedence over derived values. Thus, a user-supplied value can be entered the same way it would be for an input node, and will be used in place of the derived value for all subsequent computations.

Basically, an override is removed by deleting the user-entered value and recomputing the derived value. While it might seem that recomputation could entail a recursive chain of recomputations, the referential transparency of the intensional functions means that we only have to look up the current values of the arguments. Once recomputed, the node is entered into the pending array, and the effects of its restored value are propagated through the rest of the equation network.

Alerts

It is often important that the system be able to bring something to the user's attention when certain 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. The only real questions are practical ones of how to present that message to the user.

In some cases, the knowledge engineer can reserve an area of the screen for short messages and make them become visible when the alert condition is satisfied, but alert messages can be long, and screen area is usually in short supply. Our solution is to associate the alert message with some visible node on the screen. When the alert condition first becomes satisfied, in addition to popping-up the message in a scrollable and closeable window, the system displays the "!" annotation character next to the associated node's box. Special interface

[8] This solution was recognized and proposed by one of our colleagues, Jonathan Seder.

code allows the user to go back to any node flagged with the "!" and reread the alert message. Thus, the mechanisms for issuing and handling alerts are incorporated in the interface code, and although they seem to involve control, they place no additional requirements on the inference engine.

STATUS

Active functional systems provide a data-driven, functional approach to integrating database and expert systems. The Syntel embodiment of this approach has been fully implemented since mid-1986, and has been used to develop two families of commercial products, the Underwriting Advisor™ and the Leading Advisor™ systems which were initially developed on Xerox 1100-series workstations. Both the inference engine and the user interface were written in InterLisp; they have been delivered using an inference engine written in PL/I that runs on IBM System/370 mainframes under MVS/XA using CICS, with the interface system written in C that runs on PC/AT and PS/2 workstations. The equation networks for these applications contain several thousand nodes each with inference chains more than 200 functions deep and reference data relational tables that contain over 100,000 values. While there is no direct relation between functions and rules or frames, these are large expert systems by any measure.

Notwithstanding the large knowledge base size and the very considerable depth of (probabilistic) computation, performance in typical end user environments has been acceptable. Actual response times in time-shared environments depend, of course, on many factors outside Syntel. As a rough indication, however, most user transactions are executed in a few seconds or less.

From a more theoretical viewpoint, the uniform view of representing data as extensional functions and representing knowledge as a network of intensional 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. Such uniformity greatly simplifies the design of other subsystems, such as the explanation system and the knowledge engineering programming environment, that operate on the knowledge base, providing capabilities that are very difficult to create for multi-paradigm systems.

That the system is completely nonprocedural and free from side effects yields several advantages. End users can enter, change, or override data in any sequence. "What-if" experimentation is easily supported. Explanations and alerts are facilitated because outputs depend only on the values of user inputs, rather than sequence or side effects. Cases can be saved by saving user inputs, and restored by repropagating inputs. Knowledge engineers specify the relations between variables without concern for control issues and thus the knowledge base can be specified and designed from a dataflow standpoint [7]. We believe that this will also greatly simplify the maintenance phase of the knowledge-base life cycle as well, although this remains to be demonstrated.

Two issues that confront active functional systems are generality and efficiency. Generality really concerns the breadth of the class of problems for which the data-driven, functional approach is natural. For example, while one can solve problems that require graph searching in Syntel, such problems are much more naturally solved with a procedural approach. However, we believe that the active functional systems are applicable to a much broader class of problems than just the financial risk-assessment applications that have been addressed to date. It certainly includes estimation and assessment problems in general, as well as many moderate-size database problems. In particular, we have used Syntel internally to build a powerful database editor to help knowledge engineers develop and maintain reference databases.

We have described several techniques that were used to improve efficiency. In particular, program differentiation ideas proved highly effective in limiting recomputation to items that must be recomputed. The methods of breadth-first, bottom-up propagation, incremental computation of the default join, and screen-limited propagation all exploit this principle. Conditional visibility also proved to be very valuable in guiding the user and improving the efficiency of operational use without sacrificing the freedom of data-driven control, but since what should be visible depends on the semantics of the application, its effectiveness and logical correctness depend on the skill of the knowledge engineer. The development of additional ways to allow the knowledge engineer to use meta-knowledge to influence control without sacrificing the advantages of the functional approach is an important area for future research.

Acknowledgments. We are indebted to many colleagues at Syntelligence whose efforts converted the concepts described in this article into a full-scale operational system. Although we cannot identify them all by name, we want to express our appreciation for their contributions. We also want to thank the referees for the effort they devoted to reviewing and commenting on the original manuscript. Their numerous suggestions helped us to clarify both our thoughts and our presentation of them.

[6] Underwriting Advisor is a registered trademark of Syntelligence, Inc.
[6] Leading Advisor is a registered trademark of Syntelligence, Inc.

REFERENCES

1. Abarbanel, R., Tou, F., and Gilbert, V. KEE Connection: A bridge between databases and knowledge bases. In *AI Tools and Techniques*, M. Yazdani, and M. Richer, Eds. Ablex, Norwood, N.J. (1988, to be published).
2. Ackerman, W.B. Data flow languages. *IEEE Computer 15*, 2 (Feb. 1982), 15–25.
3. Astrahan, M.M., et al. System R: A relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.
4. Blakeley, A.J.A., Larson, P.A., and Tompa, W. Efficiency updating materialized views. In *Proceedings of the 1986 ACM-SIGMOD Confer-*

ence on the Management of Data (Washington, D.C., May 28–30). ACM, New York, 1986, pp. 61–71.
5. Brodie, M.L., and Mylopoulos, J., Eds. *On Knowledge Base Management Systems*. Springer-Verlag, New York, 1986.
6. Brownston, L., Farell, R., Kant, E., and Martin, N. *Programming Expert Systems in OPS5*. Addison-Wesley, Reading, Mass., 1985.
7. Bull, M., Duda, R. Port, D., and Reiter, J. Applying software engineering principles to knowledge-base development. In *Proceedings of Expert Systems in Business 87* (New York, Nov.). Learned Information, Medford, N.J., 1987, pp. 27–37.
8. Buneman, O.P., and Clemons, E.K. Efficiently monitoring relational databases. *ACM Trans. Database Syst. 4*, 3 (Sept. 1979), 368–382.
9. Buneman, O.P., Frankel, R.E., and Nilchil, R. An implementation technique for database query languages. *ACM Trans. Database Syst. 7*, 2 (June 1982), 164–186.
10. Codd, E.F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4 (Dec. 1979), 397–434.
11. Date, C.J. *Relational Database: Selected Writings*. Addison-Wesley, Reading, Mass., 1986.
12. Duda, R.O., and Reboh, R. AI and decision making: The Prospector experience. In *Artificial Intelligence Applications for Business*. W. Reitman, Ed. Ablex, Norwood, N.J., 1984, 111–147.
13. Duda, R.O., Hart, P.E., Reboh, R., Reiter, J., and Risch, T. Syntel: Using a functional language for financial risk assessment. *IEEE Expert 2*, 3 (Fall 1987), 18–31.
14. Gray, P. *Logic, Algebra and Databases*. Ellis Horwood/John Wiley and Sons, New York, 1984.
15. Hammer, M., and McLeod, D. The semantic data model: a modeling mechanism for database applications. In *Proceedings of the 1978 ACM-SIGMOD Conference* (Austin, Tex., May 31–June 2). ACM, New York, 1978, pp. 26–35.
16. Holsapple, C.W., and Whinston, A.B. *Manager's Guide to Expert Systems Using Guru*. Dow-Jones-Irwin, Homewood, Ill. (to be published).
17. Kerschberg, L., Ed. *Expert Database Systems*. Benjamin Cummings, Menlo Park, Calif., 1986.
18. Koenig, S., and Paige, R. A transformational framework for the automatic control of derived data. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, France, Sept. 9–11, 1981), pp. 306–318.
19. Lucas, P., and Risch, T. Representation of factual information by equations and their evaluation. In *Proceedings of the 6th International Conference on Software Engineering* (Tokyo, Japan, Sept. 13–16). IEEE, New York, 1982, pp. 153–167.
20. Missikoff, M., and Wiederhold, G. Towards a unified approach for expert and database systems. In *Expert Database Systems*. L. Kerschberg, Ed. Benjamin Cummings, Menlo Park, Calif., 1986, 383–399.
21. Paige, R., and Koenig, S. Finite differencing of computable expressions. *ACM Trans. Prog. Lang. Syst. 4*, 3 (July 1982), 402–454.
22. Raju, K.V.S.V.N., and Arun, K.M. Fuzzy functional dependencies and lossless join decomposition of fuzzy relational database systems. *ACM Trans. Database Syst. 13*, 2 (June 1988), pp. 129–166.
23. Reboh, R., and Risch, T. Syntel: Knowledge programming using functional representations. In *Proceedings of AAAI-86* (Philadelphia, Pa., Aug. 11–16). Morgan Kaufman, Los Altos, Calif., 1986, pp. 1003–1007.
24. Ross, S. *A First Course in Probability* (3d ed). McMillan, New York, 1988.
25. Sharp, J.A. *Data Flow Computing*. Ellis Horwood/John Wiley and Sons, New York, 1985.
26. Shipman, D.W. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst. 6*, 1 (Mar. 1981), 140–173.
27. Shortliffe, E.H. *Computer-Based Medical Consultations: MYCIN*. Elsevier, New York, 1976.
28. Shortliffe, E.H. Medical expert systems—knowledge tools for physicians. *The Western Journal of Medicine 145*, 6 (Dec. 1986), 830–839.
29. Sibley, E.H., and Kershberg, L. Data architecture and data model considerations. In *Proceedings of the AFIPS National Computer Conference* (Dallas, Tex., June 13–16, 1977), pp. 85–96.
30. Stefik, M.J., Bobrow, D.G., and Kahn, K.M. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software 3*, 10 (Jan. 1986) 10–18.
31. Stonebraker, M. Triggers and inference in database systems. In *On Knowledge Base Management Systems*. M. L. Brodie, and J. Mylopoulos, Eds. Springer-Verlag, New York, 1986, 297–314.
32. Sussman, G.J., and Steele, G.L., Jr. Constraints—a language for expressing almost hierarchical descriptions. *Art. Intell. 14*, 1 (Aug. 1980), 1–39.
33. Wadge, W.W., and Ashcroft, E.A. *Lucid, the Dataflow Programming Language*. Academic Press, New York, 1985.
34. Wiederhold, G. Knowledge and database management. *IEEE Software 1*, 1 (1984), 63–73.

CR Categories and Subject Descriptors: D.1.1 [**Applicative (Functional) Programming**]; H.2 [**Database Management**]; H.2.3 [**Languages**]; I.2 [**Artificial Intelligence**]; I.2.4 [**Knowledge Representation Formalisms and Methods**]; I.2.5 [**Programming Languages and Software**]
General Terms: Design, Languages
Additional Key Words and Phrases: Active databases, active functional systems, data-flow languages, default logic, decision support, equation networks, expert system tools and techniques, functional languages, fuzzy and probabilistic reasoning, inference networks, nonprocedural languages, program differentiation

ABOUT THE AUTHORS:

TORE RISCH is a member of the technical staff at Hewlett-Packard Laboratories. At Syntelligence, he was a principal architect and developer of the Syntel system. Prior to joining Syntelligence, he contributed to both the Prospector and the Hydro projects at SRI International, and, while he was at the IBM Research Center in San Jose, developed a functionally-based knowledge representation language intended for financial and business applications. His research interests include functional languages, database technology, and expert systems. Author's present address: Tore Risch, Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94303.

RENÉ REBOH is the Director of Systems Development at Syntelligence, where he is a principal architect and developer of the Syntel system. He came to Syntelligence from SRI International, where he was the Project Director for SRI's expert system group, the principal architect and developer of the Prospector and Hydro systems, and the creator of the KAS system for knowledge acquisition. He has also done research in AI languages, intelligent database systems, and automatic theorem proving.

PETER E. HART is Vice President for Research and Development at Syntelligence. Prior to that, he founded and was the first director of the Schlumberger/Fairchild AI Laboratory in Palo Alto, before which he was the director of the Artificial Intelligence Center at SRI International. Among his many activities at SRI, he started and directed the development of the Prospector expert system for mineral exploration. A fellow of the IEEE, he has published numerous papers in artificial intelligence, and is a coauthor of the book *Pattern Classification and Scene Analysis*. Authors' present addresses: René Reboh and Peter Hart, Syntelligence, 1000 Hamlin Court, P.O. Box 3620, Sunnyvale, CA 94088.

RICHARD O. DUDA is a Professor of Electrical Engineering at San Jose State University. During the four years he was at Syntelligence, he was concerned with knowledge representation and inference under conditions of uncertainty. Prior to that, he was active in research in pattern recognition, machine vision, and expert systems at Fairchild and SRI International, and was a principal contributor to the Prospector system. A Fellow of the IEEE, he is a coauthor of the book *Pattern Classification and Scene Analysis*, and a member of the editorial boards of *IEEE Expert*, the *IEEE Transactions on Pattern Analysis and Machine Intelligence*, and *Artificial Intelligence*. Author's present address: Richard O. Duda, Dept. of Electrical Engineering, San Jose University, One Washington Square, San Jose, CA 95192.