The BEACON logic-based configurator is designed to ensure the correct and complete configuration of large computer systems at the time of order entry. BEACON is based on a semantic network, KNET, implemented in PROLOG. Unlike other configuration expert systems, BEACON uses a declarative, logic-based approach, as opposed to a data-driven production system or hybrid design. Among other virtues, this allows for a completely interactive ordering session which is guaranteed correct with respect to the underlying model, rather than a batch-mode order-correcting system. KNET allows for configuration by individuation of an abstract, generic representation in a manner that is analogous to the instantiation of logical variables in a PROLOG-style proof.

1. INTRODUCTION

Configuration was one of the first and most successful applications of expert-systems technology. The R1 project [20] and its follow-ons XCON and XSEL [21] established the utility of artificial-intelligence techniques for the class of applications typified by the complicated task of assuring that orders for large computers, which consist of long and widely varying parts lists, can be correctly assembled into complete, working systems. Typically, this requires the simultaneous satisfaction of a great many criteria, encompassing not only engineering requirements, but also corporate standards, marketing practice, logistics, and maintenance. To complicate matters further, the nature of the configuration problem can be quite dynamic in a large computer corporation, with a large and varied product line changing rapidly over time.
Wu et al. [26] distinguish between an engineering configurator, which produces proper parts lists and interconnections from an engineering perspective, and a sales configurator, which assists marketing representatives in assembling an order. XCON is an engineering configurator, while the more recent XSEL program [21] is designed as a salesperson's assistant which produces input for XCON. XCON and XSEL are separate programs used by different organizations [15]. We would add a third distinction, between configuration using parts as input and configuration in terms of requirements stated at some higher level than the actual parts to be ordered (e.g. sufficient disk space to support the needs of a typical 200-bed hospital). We call the latter functional configuration.

XCON is written as a forward-chaining production system. As such, it has a data-driven architecture, taking as input a list of parts, which triggers production rules that combine to allow the specification of connections or additional parts, or perhaps the detection of errors. Thus, it is a batch-mode system which in one sense can be characterized as a configuration checker, or parts-list validator [26]. Borrowing terminology from automata theory, it is essentially an acceptor rather than a generator. A forward-chaining production system may be the most appropriate architecture for a batch engineering configurator like XCON because, being data-driven, it would tend to adjust well to unordered input such as might be expected when the order-placing process is disconnected from the order-checking process. Similarly, a purely rule-based system can be quite successfully applied to functional configuration, because knowledge in this domain tends to be relatively unstructured and more oriented to “rule-of-thumb” facts, thus taking advantage of the well-known versatility of rule-based systems.

Simple production systems, however, do have shortcomings in dealing with large and complex tasks, and several other configurators have benefited from the use of more structured knowledge representation schemes [12, 24, 26]. One reason that has been cited for this is the greater efficacy of knowledge engineering in a structured (e.g., frame-based) representation, where “encoding is made closer to actual expert knowledge and hence easier to construct,” and where “maintenance of a knowledge base is made simpler by having modular sources of configuration knowledge” [26]. In fact, for applications the size of large system configurators, the structuring of knowledge is even more important because of the tendency of large production rule-based systems to be untenable due to the sheer size and complexity of interaction among the free-standing rules. Indeed, this has proven to be the case with XCON, where the difficulty of maintaining its thousands of rules has recently prompted a reimplementation in a more structured language system, RIME [23]. Others have developed a “hybrid representation” scheme for configuration, combining forward-chained rules with frames [26], which, however, appears to remain essentially procedural and batch-oriented.

Discontinuity between sales and engineering configurators has also been criticized [26]; rather than assisting the salesperson in creating a list of parts for later submission to a separate post hoc engineering configurator, it may be preferable to have a system that completely integrates the total engineering knowledge base with the ordering activity of the salesperson, in a true point-of-sale interactive configurator. In order to address this, as well as the need for a model-based approach where the systems being configured are represented structurally in a hierarchical knowledge base, we have implemented the BEACON configurator in PROLOG using a
semantic network formalism, KNET [11]. BEACON produces a truly interactive, model-driven configuration session which communicates and imposes the engineering rules of valid configuration at the time the choices are made by the salesperson. In fact, it supervises the session to the point that all the necessary information (and only the necessary information) is elicted from the user, and choices are presented such that the user never has the opportunity to make an invalid selection. This is possible because of a strict adherence to a formal knowledge representation scheme (with a few notable exceptions to be discussed in Section 5.4), and because of a paradigmatic approach to the processing of the knowledge base which is based in logic. This logic-based approach to configuration extends beyond the simple fact of its implementation in PROLOG, to include a strongly declarative orientation and an inference strategy that is in the spirit of, and draws many parallels from, PROLOG-style search.

2. LOGIC-BASED CONFIGURATION

Intuitively, a pure declarative approach to configuration holds great appeal. Consider the following example PROLOG program, which demonstrates how in a simple case configuration can be accomplished by PROLOG's procedural interpretation of factual statements about permissible combinations of objects:

```prolog
concept(b25computer,[Monitors,ProcessorModules,GraphicsModules]) :-
  roleSet(Monitors,b25monitor,[1,1]),
  constraint(colorGraphics,Monitors,Graphics Modules),
  roleSet(ProcessorModules,b25processorModule,[1,1]),
  roleSet(Graphics Modules,b25graphicsModule,[0,1]).

concept(b25processorModule,[CPUs,Memory]) :-
  roleSet(CPUs,b25CPU,[1,1]),
  roleSet(Memory,b25memoryBoard,[1,2]).

concept(b25monitor,'BW-CRT').
concept(b25monitor,'Color-CRT').
concept(b25graphicsModule,'Graphics').
concept(b25CPU,'CPU').
concept(b25memoryBoard,'256K').
concept(b25memoryBoard,'512K').

constraint(colorGraphics, ['Color-CRT'], Graphics Modules) :- !,
  Graphics Modules=[_]. % color requires exactly 1 graphics mod
  constraint(_,_,_).

roleSet([],_,[0]) :- !. % roleSet(Parts,Type,Range)
roleSet([],_[Min,_,0]) :- Min=0. % returns list of n Parts
roleSet([Parts|Rest],Type,[Min,Max]) :- % of given concept Type,
  concept(Type,Parts),
  NewMin is Min - 1, NewMax is Max - 1,
  roleSet(Rest,Type,[NewMin,NewMax]).
```

It will be seen that this represents a specification for a microcomputer (resembling the Unisys B25) consisting of a monitor (which may be black-and-white or color), a
processor module, which in turn consists of a CPU and one or two memory boards that come in 256K and 512K sizes; and a graphics module, which is optional unless a color monitor is used, in which case it is mandatory. (This is a vast oversimplification of an actual KNET model; the implementation of BEACON, also, bears little resemblance to this sample program.)

The predicate names are only illuminating in the context of the semantic network to be described later. For the moment, consider that concept represents a description of a component whose identifier is given as the first argument. The concept rule clauses are abstract descriptions, with the second argument being a list of variables which represent the subcomponents that define the identifier, and which the rule goals serve to instantiate. The concept ground clauses, on the other hand, are concrete descriptions of actual individual parts, represented in the second argument as an atomic string identifier.

The instantiating goals of concept are calls to the domain-independent predicate roleSet, which returns through its first argument a list of subcomponents of an atomic type given to its second argument. The cardinality of the list returned is restricted to an inclusive numeric range, specified in the third argument. (The predicate is naive, however, in that sets will be duplicated with list elements in every possible ordering.) In this implementation, roleSet establishes a recursive walk via the concept rules, gathering up individuals from the concept ground clauses.

The final feature of the specification is the constraint, of which there is one example. Its first argument is also an identifier, and its remaining arguments specify component variables which possess some additional interdependence beyond that captured by the simple structure given so far. (The final constraint clause permits success where constraints are not applicable.) Note that this particular constraint makes use of a PROLOG language feature to coerce the term structure of the variable GraphicsModules to a singleton list, ensuring that its later binding overrides the numeric range of the roleSet predicate (although it must be subsumed by the latter). This is not generalizable, and indeed, in this simple approach there is no elegant way to capture the full power of constraints as they are used in KNET and BEACON (see below); however, it does demonstrate the notion of constraints propagating their effects forward, rather than checking after the fact and acting by logical failure.

This program produces the behavior shown below:

```prolog
| ?- concept(b25computer,X).
X=[["BW-CRT'"],[["CPU"],["256K'"],[[ "Graphics'"],[ ]];
X=[["BW-CRT'"],[["CPU"],["256K'"],[[ "Graphics'"],[ ]];
X=[["BW-CRT'"],[["CPU"],["256K'"],[[ "Graphics'"],[ ]];
X=[["BW-CRT'"],[["CPU"],["256K'"],[[ "Graphics'"],[ ]];
X=[["BW-CRT'"],[["CPU"],["512K'"],[[ "Graphics'"],[ ]];
X=[["BW-CRT'"],[["CPU"],["512K'"],[[ "Graphics'"],[ ]];
X=[["BW-CRT'"],[["CPU"],["512K'"],[[ "Graphics'"],[ ]];
X=[["BW-CRT'"],[["CPU"],["512K'"],[[ "Graphics'"],[ ]];
X=[["BW-CRT'"],[["CPU"],["512K'"],[[ "Graphics'"],[ ]];
X=[["BW-CRT'"],[["CPU"],["512K'"],[[ "Graphics'"],[ ]];
```
It can be seen from the first query that this declarative description of what constitutes a valid B25 configuration is capable of generating all such valid parts lists, including all variations of memory number and type, and with and without graphics modules except where obligatory (i.e. with a color monitor). The remaining queries show that it also accepts or rejects proposed configurations, as appropriate. Thus, the generic concept of a B25 computer that is specified by the top-level rule truly captures the notion of all valid instances of that abstract B25, in a manner that is very familiar and natural to logic programmers. Note that if the roleSet predicate were written so that the user was consulted for the number and selection of concept ground clauses returned (cf. [14]), then a true interactive configurator would result.

While it will be seen that the design and behavior of the preceding example differ from BEACON in a number of important ways, this introduction should serve as a reference point in the descriptions that follow, by suggesting analogies between this form of proof and the action of BEACON. As noted before, these analogies will be based on the essentially declarative nature of the semantic network description of systems, and on the movement through the search space defined by this structure in a manner resembling the depth-first traversal of PROLOG. The KNET description of an abstract system is individuated in a process that fundamentally corresponds to the restriction and successive instantiation of logical variables in the example program above.

3. KNOWLEDGE REPRESENTATION IN KNET

KNET is a semantic network formalism developed by Michael Freeman [10, 11, 12], in the tradition of Brachman's KL-ONE [4, 6], but implemented in PROLOG. The primitive objects in KNET are the concept, the roleSet, and the constraint. In terms of frame representations, concepts can be said to correspond to frames, and roleSets to slots. To a certain extent, constraints resemble demons, or procedural attachments, but it will be seen that constraints are generally confined to a declarative interpretation referring directly to, and acting only upon, other objects in the network (specifically, roleSets). Concepts participate in a specialization (or class-subclass) hierarchy, and, via their roleSets, in an aggregation (or part-of) hierarchy. This is illustrated schematically in Figure 1. By convention, a concept is indicated graphically by a labeled ellipse, a roleSet by a small circled square with an attached label and numeric range, and a constraint by a labeled rhombus. Figure 1 also shows...
FIGURE 1. Schematic representation of a KNET fragment.

several kinds of arcs that can connect these nodes, to be described in succeeding sections.

3.1. The Aggregation Hierarchy

Rolesets designate components or attributes of concepts. A roleset is connected by a single line to its owner, that is, the concept of which it is an attribute or part. A roleset also has an associated type, indicated by an arrow to another concept. Thus, the roleset can be viewed as sitting astride a link in the aggregation hierarchy, connecting concepts. Finally, a roleset has a range, representing the minimum and the maximum possible cardinality of the collection of objects, selected from the type, denoted by the roleset. Roleset R3 in Figure 1 is owned by C4, has type C6, and has a numeric range from 0 to 1.

Rolesets, being attributes of their owners, can be said to collectively define these concepts. In a configuration application, they most commonly specify physical subcomponents that constitute the owner concept, but they can just as well specify attributes such as voltage, or even intangible characteristics that serve to somehow distinguish a concept. Constraints establish relationships among rolesets (which may be widely separated in the network) that would otherwise be difficult or exceedingly inefficient to capture using structure (i.e. concepts plus rolesets) alone. In Figure 1, constraint X1 is said to be housed in concept C4, to which it is thus connected by a line, but it is actually only concerned with the rolesets R1 and R3 of C4, to which it is connected by arrows. A constraint is housed at the concept which is the lowest common scoper on the aggregation hierarchy of all the rolesets to which it is connected. This is a powerful feature, because it serves to partition large and otherwise cumbersome rule sets in a way that focuses the inference mechanism on a smaller set of rules, whose scope is reflected by their level in the aggregation
hierarchy. Even more importantly, the KNET structure serves as a navigable organizing paradigm, so that designers and maintainers can expect to find constraints housed at some concept, which is appropriate to their action, rather than in a flat list of production rules.

3.2. The Specialization Hierarchy

A specialization link (sometimes called an "is-a" link) is drawn as a double arrow, pointing toward the more general concept. Thus, concept C4 would be said to specialize C1, in Figure 1. That is, C4 is a more specific concept than C1, but still falls within the general class of objects denoted by C1.

A fundamental feature of KNET, as well as most similar systems, is inheritance, which in KNET is defined only along the specialization hierarchy. Rolesets and constraints are inherited down specialization links, which is to say that anything which is true of a general concept must also be true of the more specific concept derived from it. However, rolesets may be narrowed in numeric range or type as they are inherited; that is, a roleset's range may fall inside (but never outside) the range of its more generic source, and similarly its type may be a specialization of the type of the source roleset. In Figure 1, the specialization of C4 from C1 entails a convergence of R2's numeric range and of R1's type. This feature of inheritance, by which ranges and type are required never to be more general as a roleset is inherited, is called subsumption. KNET strictly enforces subsumption. Inheritance permits an economy of expression and consistency analogous to normalization in relational databases, and subsumption checking further enhances the integrity of information in the knowledge base.

Specialization can be thought of as being induced by narrowing of range or type in component rolesets. Specialization can also be achieved by simply adding new rolesets or constraints, as is shown in Figure 1 for R3 and X1, respectively. In this sense, concept C4 is more specific than C1 by virtue of having an additional attribute and constraint (in addition to having had its inherited rolesets narrowed in range and type). The process of narrowing or converging a roleset is closely related to the notion of instantiation of a complex logic term. For instance, a logic variable in a term may be instantiated by unification with a complex term containing more variables, which themselves must become instantiated. With each unification, the term becomes more specific—that is, it can unify with a smaller number of terms. Similarly, a roleset can be looked on as a KNET "variable", which is progressively made more specific as its range is narrowed and as its type is specialized. Of course, the imposition of term structure on logic variables is only one way in which this kind of typing of variables can be accomplished, but it is apt as an analogy with the aggregation (roleset) hierarchy in particular, insofar as the entire KNET specification can be thought of as a deeply nested, complex list structure of such variables.

The notion of attaching constraints to logical variables has been examined separately as a way of constraining the search space in certain combinatorial problems [18]. Note that it should be possible for specialization to be induced by a narrowing of the action of an inherited constraint—for instance, if the action of a constraint is to narrow the range of a roleset, a specialized version of that constraint might propagate an even narrower range. However, the semantics of this are not yet
well defined in KNET for the general case. The combination of range, inheritance, and constraint supported by the KNET subsumption mechanism has proved to be a very powerful and flexible tool for describing various systems.

3.3. Individuation, Decomposition, and Differentiation

Since the process of configuration is in effect the creation of an individual from a generic description, it is important to look in a little more detail at how this is achieved in KNET. Figure 2 shows the creation of an individual concept I1 from the generic concept C1; this distinguished form of specialization is represented by a line running down the middle of the specialization arrow. In the process of specialization, the roleset R1 has had its numeric range completely converged to (10 10). In addition, an individual's rolesets specify individuals descended from their type to fill their "role", which are thus called fillers. If C2 itself had an individuation, then R1 could have that as its filler, and I1 would indeed be a proper individual. In KNET, fillers are indicated by an additional arrow from the roleset body to the filler concept, called a qua link [9].

However, C2 instead has two further specializations which are generic concepts, C3 and C4. These particular concepts together form a decomposition of C2, as indicated by the loops around their specialization arrows. This construct is useful for specifying groups of concepts which are a partition of their parent concept; for instance, a concept "printers" might be properly decomposed into "serial printers" and "parallel printers". C3 and C4 do have individuations I3 and I4 which are candidates for fillers of R1, being ultimate specializations of C2.

Since the individual role R1 has cardinality ten, it may refer to ten components, which, however, need not be all of the same filler type. When it is necessary for an

FIGURE 2. Individuation, decomposition, and differentiation in KNET.
individual role to refer to distinguishable components, e.g., two or more different printers, then differentiators are created. These are spawned rolesets (such as R1.1 and R1.2 in Figure 2, shown connected to the original roleset R1 by a dotted line), whose cardinality must sum to that of the roleset being differentiated. Qua links are then created to indicate the fillers of the differentiators, which in the example are I4 and I3, respectively.

3.4. Implementation

This essentially completes a description of the “user view” of KNET. KNET knowledge bases are implemented as PROLOG ground clauses, which would be fairly accessible to anyone familiar with this top-level description, with several notable exceptions: there are a number of relations which store information to speed up access to objects in the net, and there are the constraint bodies, which are PROLOG rules acting on KNET structure. Because the latter are (generally) required to confine their actions to remain within the KNET domain (obeying subsumption, for instance), they can be seen as simply means for dynamically altering KNET structure, and the strongly declarative orientation of KNET representation is preserved.

The complexity of KNET constructs in actual use is necessitated by the need for power and generality in “real-world” applications. However, it should be noted that KNET is a more streamlined representational system than many other approaches. Its expressive power has been carefully weighed against the need for efficiency and ease of use, since further development of KNET has been driven by the practical needs of the BEACON project, to be described below. In practice, it has proven to be a highly successful knowledge-structuring system for the essentially logic-based approach that BEACON embodies.

4. CONFIGURATION IN BEACON

In considering how the BEACON configurator makes use of an abstract KNET description of a system in order to create a configured individual, it is instructive to reexamine the example knowledge base given earlier. This is represented schematically as shown in Figure 3.

Note that this corresponds closely to the simplified PROLOG representation in which generic concepts were rules whose goals were rolesets, effectively named by logical variables, and constraints which could establish interdependencies between variables. Individual concepts, which were represented as ground clauses of the same concept predicates but holding atomic strings rather than lists of variables, can now be seen to correspond in KNET to decompositions of type concepts, where the number of decomposers is the number of clauses in the database. Thus, the earlier representation is perhaps deceptive, in that the decomposing specializations are “understood” in the second argument, distinguished as atoms, whereas in the rule form the second argument is a list of roleset variables.

As has been suggested, the process of configuring from the network representation consists of a PROLOG-style search, i.e. a depth-first, left-to-right traversal of the aggregation hierarchy. The first step in this process is to create an individuation of the top-level concept, b25computer, which inherits all of the structure below it.
FIGURE 3. A simplified KNET model for B25 configuration.

(This individuation is not shown in the figure.) This “shadow network” is then traversed. The first roleset encountered is Monitors, which already has its range converged; however, there are two possible fillers of its type, i.e. the decomposition of b25monitor into 'BW-CRT' and 'Color-CRT'. One of these must be chosen, and a qua link created to it from the Monitors roleset.

In the sample program, every possible combination of fillers was chosen by backtracking. In actual practice, what is desired is a more directed, or pruned, search. Whenever a decomposition of a “leaf” item is encountered, BEACON queries the user as to what choice should be made; this is the basis for its interactive mode of use. Similarly, whenever an unconverged numeric range is encountered in the traversal, BEACON queries the user as to how many of each item in the decomposition are desired. For rolesets with converged ranges and only a single possible filler, BEACON quietly makes that choice for the user.

When Monitors has its filler, the constraint colorGraphics is triggered. This is a piece of PROLOG code which, if 'Color-CRT' happens to be the filler of
Monitors, changes the shadow-network representation so that the numeric range of the roleset GraphicsModules is changed from (0 1) to (1 1). While this is accomplished by side effect in practice, at the conceptual level it can be treated as logical, since rolesets (and their numeric ranges) can be viewed as variables awaiting instantiation, and fair game for restricting in advance. Constraints can change not only numeric ranges, but also decompositions, and in fact are capable of arbitrary action on KNET structure within the limitations of subsumption. The state of affairs in the individuating network, at this point in the traversal, is shown in Figure 4.

Also in the spirit of PROLOG-style proof, the traversal of the aggregation hierarchy can be recursive. ProcessorModules is descended, creating an individuation of b25processorModule, and the same is done in turn for each of its rolesets. Had the number of ProcessorModules required been more than one, then the mechanism is capable of differentiating that roleset and creating multiple distinct traversals of the subhierarchy.

Also as might be expected, a form of backtracking is supported. This is necessary because a mistake might be made by the user in the process of configuration, or perhaps the effects of constraints might cut off desired choices later. The user can "fail" back to prior choice points using a command called undo. This is implemented at a meta level (rather than at the PROLOG level), and has features that
provide extra help to the user and cause it to skip over choice points which are insignificant to the user. An enhanced version of this has also been developed, to be described below.

To reiterate, the BEACON configurator is an application program that works on a generic KNET representation, emulating one form of logical proof by traversing the aggregation hierarchy and gradually individuating the network in a manner analogous to the instantiation of logical variables in PROLOG. It is important to note that the configurator itself is a relatively simple application program for walking the KNET network, and that the domain-specific knowledge directing configuration is embodied in the structure of the network and especially in the constraints. In fact, many different application programs can be run over the same structure, for different purposes. The generality of the structure can be maintained by using "typed" constraints to embody the different purposes of these different application programs. Thus, we have so far described only "configurator" constraints, but there are a number of other types, such as "database loader" constraints, which will be described in the next section. Thus, a uniform representation of the same complex system can be used for many purposes by application programs that consist of relatively simple traversal algorithms.

5. ADDITIONAL BEACON FEATURES

The following sections will describe various aspects of the practical matter of fielding an extremely large expert system founded on the theoretical approach described above.

5.1. The Browser / Editor and Constraint Language

A requirement imposed on the BEACON system is that a major portion of the modeling be done not by experienced PROLOG programmers or knowledge engineers, but by the designers of the objects being modeled, i.e., by the plant engineers who create the computers to be configured. Thus, the other major component of BEACON is the browser/editor, a tool to permit easy creation and modification of KNET structure. (BEACON, it should be noted, stands for "Browser/Editor and Automated configurator".) The browser/editor supplies commands for both global and local (hierarchy-directed) movement around the network, and for adding, modifying, and deleting objects. Note that the latter operations can be nontrivial in the middle of multiple hierarchies with inheritance, as well as the "speedup" relations alluded to above. In addition, there are various utilities, help functions, audit trails, etc., such as might be expected in an interface to a complex data structure.

Perhaps the most important element in making KNET modeling accessible to a wide audience is the presentation of constraints. While the creation of constraint bodies in PROLOG was quite natural to the developers, it was not deemed suitable for end users. Accordingly, a large effort was put into the development of an application-specific constraint language, with a syntax resembling more conventional
programming languages. A structure editor was created for this language, which was incorporated directly into the browser/editor, as well as an interpreter and compiler. The constraint corresponding to the running example would be written in this language as follows:

```
if _chromatics.type.name=color then
   _graphicsModules.min:=1;
end if;
```

While this form appears to have little to do with the underlying PROLOG implementation, it is much more palatable to novice users, and in fact the PROLOG translation of this constraint would not be illuminating even to experienced PROLOG programmers, so dependent is it on KNET implementation details. As an example of the expressive power of the constraint language, consider the following constraint which keeps track of the bus length of a B25 system as a certain kind of module is added:

```
if not has_differentiators(_diskExpansionModules) then
   if _diskExpansionModules.number > 0 then
      _busLength.filler :=
         _busLength.filler_or_type +
         _length.filler_or_type *
         _diskExpansionModules.number;
   end if;
end if;
```

Briefly, this constraint finds differentiators (i.e. those rolesets that do not themselves have differentiators) of the roleset _diskExpansionModules, and computes the bus length those modules require as the length required by each one of that kind of component, _length.filler_or_type.name, multiplied by the number of components of that kind, _diskExpansionModules.number.

The browser/editor supplies several other utilities to the modeler which make life easier. One of these is an elicitation constraint generator: a separate application program that traverses a model, automatically writing the relatively trivial constraints which consult the user for range and filler information when that has not already been determined by the action of previously fired constraints. Another standalone program assists the user doing knowledge-base amalgamation, which is the process of combining separate but related knowledge bases. Such knowledge bases are quite common because the peripherals for large computer systems are manufactured independently of the central system, and are reused by many systems in the product line, and thus should be modeled separately by the appropriate experts. These separate subsystems exist as distinguishable models, and in fact together form the basis on which new system models are built. This requires that the models be spliced together seamlessly, and separated and respliced as the subsystem models change. This involves such difficulties as recognizing and reconstructing “spanning constraints” that cross the boundaries of models, and reconciling the identities of overlapping concepts. The latter is an issue because KNET objects are actually specified by generated internal identifiers, so that their representation is independent of their (changeable) external string identifiers.
5.2. The Database Loader

Another important feature of BEACON relates to the problem of maintaining a knowledge base with a large amount of rapidly evolving data. It was realized early on that it would be inappropriate to try to store and modify information as volatile as pricing in a KNET structure; such information is better maintained in a conventional database management system. Accordingly, BEACON makes use of an external corporate relational database for frequently updated catalog information. This is done by making the lowest level of the aggregation hierarchy consist of generic catalog descriptions, e.g. “parallel printer”; these can be filled by any of a number of catalog items—which, however, do not have any further distinguishing specializations. The catalog items can then be entered into the network as decompositions, and in fact this is done by an entirely separate application program called the database loader. Thus, a baseline generic model can be maintained, which need seldom be changed, while the frequent changes to the catalog can be reflected by periodically uploading the database. The actions of the database loader are specified in the network by a distinguished set of constraints which are active only for that application; these constraints actually perform selects on the database information to load only catalog items that are supported by the model. The fact that the network is represented in PROLOG was a great advantage, since all that was required was a report generated from the relational database that prints the database tuples in the format of PROLOG ground clauses; these are then directly consulted, so that the external database is a virtual extension of PROLOG’s internal database. Analogously, at the level of KNET, the fields of the tuple correspond to the rolesets of the decomposed concept.

5.3. The Windowing Interfaces

For purposes of deployment all of these functions are provided to the modeler via a window-based visual interface that runs on a 25 × 80 character-oriented screen. Because this system was designed to be used at so many sites, there was a requirement that the interface be presented on widely available and economical hardware. Simple “flat” interfaces were judged to be too difficult to use, so it was necessary to create window-oriented versions of both the browser/editor and the configurator. Because so much special-purpose functionality was needed that was directly controllable from PROLOG, it was necessary to create a windowing functionality from scratch. This was done using the C language, which is much better suited to such I/O-intensive operations, using the foreign-language interface feature of Quintus PROLOG to allow direct control by and interaction with PROLOG. Figure 5 shows a representation of a typical display from the configurator. It provides a running order log, a spreadsheet-like menu selection window for the current choice, and a message window with instructions; all windows are scrollable. At each potential individual roleset in the traversal, the decomposers are listed, and the numeric range is enforced. The formats of the windows and elicitation message may vary according to the numbers of available fillers and ranges. Other flags in the menu window, such as PP, refer to a “packaging” feature that allows the user to keep track of parts that are bundled by Marketing for reasons of pricing policy or other special significance.
There are 5 types of serial printers for the master WSs. The total must be between 0 and 2. Enter a whole number and press Return for each choice. Then press Continue. If you do not want any serial printers simply press Continue.

### Product List

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>3250 20 CPS Daisy Wheel Printer</td>
</tr>
<tr>
<td>2</td>
<td>2595 55 CPS Daisy Wheel Printer</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1795 160 CPS Dot Matrix Printer</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>2395 200 CPS Dot Matrix Printer (serial/parallel)</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>2000 5500 1795 160 CPS Dot Matrix Printer</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>2395 200 CPS Dot Matrix Printer (serial/parallel)</td>
</tr>
</tbody>
</table>

### Configuration Log

- **B25-CPU**: 1755 80186 Processor w/256KB Memory
- **B25-266**: 575 256KB RAM Memory Board
- **B25-M1**: 1450 2x630KB Floppy Disk Module
- **B25-M3**: 2975 10MB Hard, 630KB Floppy Disk Module
- **B25-PS**: 200 B25 Power Brick
- **B25-EC**: 0 B25 Power Brick Cord

FIGURE 5. Schematic representation of a typical configurator display. Prices shown were fabricated for this example.
5.4. Ancillary Expert Systems

Just as it is appropriate to deviate from PROLOG when implementing functionality that is better suited to other languages (such as I/O routines in C), so it is occasionally advisable to leave the KNET formalism to program at the next level down, i.e. PROLOG. This is the case for certain aspects of the configuration of systems that have been modeled; for instance, there is a collection of rules describing how modules may be assembled in the B25, which are both local (e.g., a graphics module, if present, must go beside the CPU module) and global (e.g., the overall bus length of the system may not exceed a certain number of inches). Such rules are far more easily implemented as a straightforward PROLOG program, without reference to KNET, and in practice they are simply called as subroutines by special constraints at the appropriate point. A vastly more complex problem occurs in large mainframe systems such as the Unisys A15, where there are exceedingly complex packing and interconnection rules for specialized “cards” within “bases” within “cabinets”. This function is performed by a separate PROLOG expert system which is itself quite large, but which is again simply called by a constraint. This program gathers its framework structure and parameters from a KNET model, and returns modifications to that KNET structure based on the resulting packing.

6. SIZE AND PERFORMANCE STATISTICS

As shown in Table 1, the programs associated with BEACON are quite large by PROLOG standards. Furthermore, it should be borne in mind that the functional activity of configuration is actually directed by the constraint bodies, which are housed in the KNET models and not the BEACON programs; this is reflected in the sizes shown for the knowledge bases, of which nearly half the lines of code are in constraint bodies. We believe this will be one of the larger PROLOG systems ever written, and probably the largest application of a KL-ONE-style semantic network.

It will be noted, from the ratios of clauses to relations and lines of code to clauses, that the coding style tends to relatively large relations with few clauses. This reflects the fact that flow of control in this code is directed more within clauses, by internal disjunction and implication, than by multiple clauses. While this is not

<table>
<thead>
<tr>
<th>Program</th>
<th>Relations</th>
<th>Clauses</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contigurator</td>
<td>788</td>
<td>1372</td>
<td>9,924</td>
</tr>
<tr>
<td>Browser/editor</td>
<td>1033</td>
<td>2270</td>
<td>20,987</td>
</tr>
<tr>
<td>Basepacker</td>
<td>599</td>
<td>1255</td>
<td>7,631</td>
</tr>
<tr>
<td>Database loader</td>
<td>117</td>
<td>193</td>
<td>2,030</td>
</tr>
<tr>
<td>Window interface (C)</td>
<td>—</td>
<td>—</td>
<td>6,149</td>
</tr>
<tr>
<td>Total</td>
<td>2537</td>
<td>5090</td>
<td>46,721</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Knowledge base</th>
<th>Relations</th>
<th>Clauses</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>B25</td>
<td>32</td>
<td>2026</td>
<td>3,663</td>
</tr>
<tr>
<td>V300</td>
<td>32</td>
<td>7211</td>
<td>12,097</td>
</tr>
</tbody>
</table>

*aExcluding comment lines.*
TABLE 2. Size breakdowns of BEACON knowledge bases

<table>
<thead>
<tr>
<th>Node type</th>
<th>B25</th>
<th>V300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concepts</td>
<td>111</td>
<td>501</td>
</tr>
<tr>
<td>Rolesets</td>
<td>88</td>
<td>287</td>
</tr>
<tr>
<td>Constraints</td>
<td>118</td>
<td>294</td>
</tr>
<tr>
<td>Catalog items</td>
<td>194</td>
<td>692</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>511</td>
<td>1774</td>
</tr>
</tbody>
</table>

TABLE 3. Dynamic behavior of BEACON configurations

<table>
<thead>
<tr>
<th>Metric</th>
<th>B25</th>
<th>V-series with peripherals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Single</td>
<td>Cluster</td>
</tr>
<tr>
<td>Nodes visited</td>
<td>47</td>
<td>89</td>
</tr>
<tr>
<td>User queries</td>
<td>17</td>
<td>29</td>
</tr>
<tr>
<td>Constraints fired</td>
<td>58</td>
<td>110</td>
</tr>
<tr>
<td>Individuals created</td>
<td>29</td>
<td>47</td>
</tr>
<tr>
<td>CPU time (sec)*</td>
<td>35</td>
<td>62</td>
</tr>
</tbody>
</table>

*Excluding basepacker and report generation.

textbook PROLOG style, it appears to be somewhat more efficient, perhaps by minimizing unification across clauses as well as the overhead of lookups for the large number of auxiliary predicates that would otherwise be required. This concession to efficiency and manageability is more obviously reflected in the liberal use of cut and assert/retract in this code, which is a necessity familiar to programmers of any sizable PROLOG application.

Table 2 shows the sizes of typical KNET knowledge bases in terms of individual components. The B25 is a microcomputer, while the V300 is a minicomputer that comes in a range of sizes. The figures for the latter system refer to an amalgamated knowledge base that contains large generic models for peripheral subsystems, which account for a large portion of the total model. The A15 has also been modeled successfully, but its knowledge base is not much larger than that of a V300 because the subsystem models dominate.

Table 3 gives some statistics about the run-time behavior of the configurator, based on log-driven runs. These tests were run using Quintus PROLOG on a Unisys 5000/70, a 68020-based system running under the CENTIX operating system. User satisfaction with response times and overall performance has been very good during stress tests which ran as many as three configurations per processor simultaneously. The most remarkable observation on performance from the perspective of knowledge-based systems, however, is that the growth of time and space requirements for BEACON in going from very small to very large configuration systems is quite reasonable.

7. DISCUSSION

Even though BEACON is not, and in practical terms cannot be, implemented in "pure" logic, it is nevertheless noteworthy from a logic-programming perspective
for several reasons. First, it is a very large PROLOG program meant to be fielded widely in a “production” environment, and should contribute to the growing success of PROLOG and its evolution from a “research” language to mainstream computing. Also, although the built-in backtracking inference mechanism of PROLOG is subverted in the system as a whole, it is nevertheless used to good advantage locally and, we believe, contributes significantly to productivity.

Most importantly, the guiding principle in BEACON design is that configuration by individuation of an abstract semantic network representation is analogous to the instantiation of logical variables in PROLOG-style proof. For example, it has already been noted that constraints are generally positioned to be triggered as early as possible so as to propagate their effects forward, rather than setting up an *ex post facto* validity check (the logical extension of which would be a batch-mode configurator). In logic programming, one would not put clauses restricting the values of variables all at the very end of a long inference unless absolutely necessary, because that would lead to wasteful backtracking. One could place the restricting clauses immediately after the choice points, thus attempting to resatisfy only the relevant clause. However, such resatisfaction would be unacceptable in an interactive system, since it would amount to the user making a series of choices and being repeatedly told, “No, try again”. In both PROLOG and interactive configuration, it is preferable to restrict variables in advance wherever possible, to minimize backtracking to the utmost [18]. In BEACON, this means that inappropriate options are never even presented to the user.

Nevertheless, this forward propagation of constraints can lead to situations where the user follows a path that precludes a later choice that was actually desired, and this leads to the necessity for an undo capability to backtrack to the point that caused the undesired restriction. This may be undesirable, first, because the backtracking process may be lengthy and laborious for the user, and second, because a large portion of the branch of the search which is being backtracked over may remain valid, and will have to be reselected. Accordingly, one of the follow-on tasks just finished for BEACON is a “random access” undo capability, in which the user will be able to select from a menu of choice points to which to backtrack automatically; the choice at that point would then be changed manually, after which the system would automatically retrace the path over which it had backtracked to the extent possible (i.e. until encountering some pruning of the search space that occurred as a consequence of the altered choice). This will thus incorporate some of the notions of “selective backtracking” [22], in which an attempt is made to backtrack more efficiently by analyzing which variable instantiations caused the failure and then backtracking directly to the relevant point, and “intelligent backtracking” [7], which involves retaining subproofs (which would otherwise have been discarded) in order to avoid recomputing them. This is also reminiscent of work being done in the area of meta-level logic programming, for instance that of Bowen et al. [2,3], whose metaPROLOG has a predicate *demo(T, F, P)* which specifies that the formula *F* is derivable from the theory *T* via the proof *P*. *P* is thus a metavariable representation of a proof (or branch of a proof) which can be used to store information like that required for the undo feature we are implementing. Such meta-level treatments may also be instructive in dealing with the considerable problem of *customer add-ons*, another follow-on task. This task involves adding parts to a preexisting configuration (analogous, perhaps, to a stored proof in the
"demo" sense), in the case where the underlying knowledge base (corresponding to the "demo" theory) may have "changed out from under" in the intervening time.

Another interesting parallel with logic programming deals with the notion of sequential ordering of roledsets. In the abstract, there need be no particular ordering of the roledsets and constraints of a concept, any more than first-order logic intrinsically requires ordering of goals in a clause. However, it is necessary to specify an order for roledsets in practice because there is generally a preferred order of elicitation from the user. For instance, in the trivial example, it is better to ask whether the users want a color terminal before asking if they want a graphics module, for if they are offered and decline the graphics module first, they will not even be offered the color monitor. Even more serious difficulties can arise in other cases, where there is a strict one-way dependency. Analogously, logic programs in practice may require specific goal ordering to achieve efficiency, or even termination.

Another way of saying this is that search of a proof space should not be required to be depth-first left-to-right. In point of fact, BEACON does not absolutely require this, since it is actually constraints that govern the vertical descent of the search, and this is a useful mechanism for pruning. However, an additional fact standing in the way of any more general search strategy is that constraints as currently implemented are unidirectional in their action, i.e., they have a specified trigger and target(s). The notion of an omnidirectional constraint, for which any of the participant roledsets could be encountered in any order, holds great appeal but is difficult and costly to implement in practice. One can imagine approaches, similar to some meta-level control primitives that have been proposed for PROLOG [8,13,19], that would allow at least a relaxation from total to partial ordering on roledsets, and local omnidirectionality of constraints, creating greater flexibility.

It is interesting to examine the relationship embodied in KNET and BEACON between the semantic network and logic. Hayes and others [16,17,25] have argued persuasively that semantic networks (including inheritance) are merely notational variants of logic, insofar as the basic constructs of the former can be mechanically translated to the latter while retaining the full meaning. In a trivial sense, KNET is a proof by construction of this tenet. However, KNET represents the knowledge at a meta level, rather than directly in terms of logical predicate names, term structure, and implication; that is, it uses a logical representation of network constructs, e.g.

```
concept(b25monitor,'BW-CRT')
```

rather than a direct object-level representation in PROLOG,

```
b25monitor('BW-CRT')
```

such as would be implemented by the usual translation scheme [25]. This meta-level representation, of course, affords an additional level of control, which is necessary in implementing such features as forward-acting constraints and sophisticated user interaction, not to mention modeling tools. Nevertheless, as has been argued above, the action of BEACON is essentially logic-oriented, and the use of logic is not simply as a Turing-power programming language to implement the network. First-order logic has been proposed as an adjunct to semantic nets in KRYPTON, which uses
logic as an assertional component ("ABox") referring to a KL-ONE-style semantic network ("TBox") for its terminological, or definitional, framework [5]. This essentially views the semantic network as an extremely rich typing construct for the logic, and BEACON can also be seen as embodying this approach; a great deal of explicit typing is done, in fact, in the specialization hierarchy for catalog items. The semantics of this notion has been established more rigorously in LOGIN [1], a typed logic-programming system with its unification algorithm extended to support inheritance. For the application at hand, however, BEACON goes further than either of these systems by providing both a sophisticated control mechanism and a complete abstract data type captured in the network, including constraints treated as first-class, inheritable objects rather than adjunct logical sentences.

The authors wish to thank the members of the BEACON staff whose work is described here: Peggy Arico, Keith Cassell, Jerry Goldberg, Mark Lipshutz, Dr. David Matuszek, Robin McEntire, Floyd Miller, Kent Monthei, Dr. Chris Overton, Jon Pastor, Howard Rosenshine, Dr. Bill Scholz, and Cathy Smotherman. Special thanks are extended to Peggy Arico for gathering the statistics shown in the tables. In addition we thank our corporate sponsor, David Bogle, for his tireless patronage. A debt is also owed to earlier contributors who laid a foundation for the present work: Dr. Lynette Hirschman, Herb Jellinek, Chick Kozlaff, Don McKay, Dr. Martha Palmer, Karl Puder, and especially Dr. Michael Freeman, the creator of KNET and founder of the BEACON project.

REFERENCES


