AI-BASED PROCEDURES

William Bricken
October 1987

Our observations regarding methodology in general and the architecture in particular are that a large scale approach to Artificial Intelligence (AI) integration, using sophisticated, state-of-the-art AI techniques is inappropriate. In particular, we do not advocate the development of large scale expert systems (ES) empowered to act as design "experts". The development of such expert systems requires a somewhat static domain, as well as one that can be readily formalized. In addition, the development process inevitably requires a comprehensive knowledge engineering effort, with frequent and extended interactions between human experts and knowledge engineers. The design process, however, is sufficiently dynamic and informal (even subsequent to extensive formalization), and the necessary knowledge engineering effort so wide in scope that such a monolithic ES approach seems inappropriate. We do, however, recommend adoption of the, now standard, AI approaches to software development and knowledge representation, making use of iconic interfaces, windowing environments, object-oriented programming paradigms and frame based representation and inference schemes. We further suggest the development of at least one small ES (described below) of relatively narrow scope that can be, to some extent, engineered to be domain independent.

In the sections below, we describe knowledge bases and procedures applicable to the developing architecture. Where possible, we outline where and in what manner such procedures might be inserted into the architecture.

Concept-Based Information Retrieval

The current choice for the database management system, although powerful, constrains the designer to rather traditional keyword style searches. In addition, the data itself must be such that it can reasonably be stored in a form conducive to the requirements of such a database management system. Although these constraints are of little consequence to the designer in many instances, it is expected that circumstances will develop in which a rather less structured, *concept* based search involving full document search and retrieval is required. A simple example of such a requirement is the need to search through the abstracts of research articles in such a manner that user-defined *concepts* rather than externally defined *keywords* drive the search.

The concept-based information retrieval (CBIR) approach is to provide the user with a means by which search requests, posed in a query language capable of expressing the *concepts* in which the user is interested, can be used to locate relevant documents. In addition to simply retrieving documents, CBIR is able to rank the retrieved documents based upon their assumed relevance to the concept of interest. In addition, an explanation facility enables the

user to request logical, understandable and intuitive explanations as to *why* a particular document was included in the search result. All of these capabilities are present within an integrated retrieval environment which includes tools for rule browsing and editing.

The capability for concept-based search is provided by *rules* which describe relationships between topics of interest. These rules do *not* require that the user specify how the concepts of interest have been expressed in the document base, since the inference mechanisms encompassed within a CBIR system are capable of working with the raw rules themselves. A rule in CBIR takes on the following form:

A --> B g[a]

which means that if a document happens to encompass concept A, then with some certainty g[a], it should also encompass the concept B. Although a *certainty* takes on a real valued number in the interval [0,1], it should not necessarily be interpreted as a probability. Rules are, however, likely to be more complex than this, for example:

(A1 and A2) --> B g[b]

which means that if the document encompasses *both* concepts A1 and A2, then it is likely to encompass concept B with certainty g[b].

The difficulty in using CBIR rests with the process of defining concepts as sets of rules. Once the concept definition process has been completed (it need only occur once) subsequent requests for information retrieval may be carried out with ease.

A CBIR system is large and comprehensive and, by necessity, is given only a superficial treatment here. The outline below serves to provide a simplified overview of many of the components that make up the CBIR tool.

The Knowledge Acquisition Subsystem: This subsystem contains several tools which facilitate the construction and modification of queries.

The Rule Editor: This tool provides the user with the ability to create, modify and delete rules.

The Rule Browser: This tool allows the user to select individual rules or sets of rules for examination.

The Rule Analyzer: This tool performs analyses more complete than the simple syntax checking handled by the Rule Editor, enabling the user to check the rule base before attempting any data retrieval.

The Retrieval Subsystem: This subsystem performs the evaluation of documents with respect to those queries formed by the user via interaction with the Knowledge Acquisition Subsystem. Three tools are contained here:

The Preprocessor: This tool essentially expands the user's query into a fully formed search request to be sent to the rule evaluator.

The Evaluator: The evaluator controls the traversal of the search tree generated by the preprocessor, computing the value of the query to be associated with each document in the document base.

The Performance Analyzer: This tool enables the user to display the results of the search request. Histograms and lists of retrieval scores can be presented, and the user can view selected documents in such a way that the text expressions which successfully matched the document are highlighted.

The Guide Subsystem: This tool provides a generalized help capability to the user.

The Rule-Base Management Subsystem: This enables the user to select and peruse the existing rule bases.

The Database Management Subsystem: Functionally consisting of two components:

The Preprocessor: This takes as input the free format text of the documents in the document database and transforms them into files which CBIR can readily access and manipulate.

The Word Matching Mechanism: This is a collection of procedures which implement the many word matching capabilities of CBIR retrieval language.

To summarize, CBIR is a complete, powerful, yet easy to use intelligent document retrieval mechanism by which full document retrieval can take place via the search for documents whose contents are relevant to concepts supplied by the user. A retrieval mechanism such as this is much more powerful than the simple keyword search facilities supplied by other data retrieval tools and, as such, can prove invaluable when complete (yet *precise*) searches are desired.

A CBIR tool has the potential to become a valuable component in the database management system arsenal of tools. Given that database search requests are likely to stem from a variety of different sources, and given the CPU-intensive nature of the retrieval task, it seems likely that a CBIR tool should reside on the system host. Copies of the Rule Editor and Rule Browser, however, could easily reside on the individual workstations.

Knowledge Base Editing System

Much of this approach makes use of the concept of a hierarchical network of frames. Some databases, most notably the lessons learned database, are also well suited for representing in such a framework. Similarly, if small expert systems are introduced into the methodology, their knowledge bases will inevitably be represented as hierarchical networks of frames. Given the ubiquitous nature of this hierarchical structure, a means by which such structures can be conveniently browsed and edited seems crucial.

An existing system was developed to provide a simple but extensible knowledge base browsing and editing system. It provides functions for defining, accessing, displaying, editing and saving a knowledge base. The basic knowledge base access and definition functions are written in COMMON LISP and do not require other tools to be loaded. This system is thus a self contained, portable package for knowledge representation. It is divided into modules with the following capabilities:

1. Organization of knowledge into a tree of definitions, with attributes and values attached to each node. Attributes may be inherited from parent nodes. The knowledge base may be partitioned into different knowledge sources, with a distinct tree for each source.

2. Browsing and editing of the knowledge base. The overall structure is shown as a tree display, and operations on the display allow creation and deletion of nodes and links. Nodes may be displayed and edited in a tabular format to add or remove attributes and to alter their values.

3. Reading and writing of the knowledge base to disk in a text-editable format. During the read process, information from several saved knowledge base files may be merged. Filename prompting and directory management is provided.

4. A sub-system for entering and editing mathematical formulas is provided. Type checking is provided during the editing process and at run-time. A library of pre-defined knowledge base entries supply operators for mathematical, list manipulation, and knowledge base access operations.

Constraint Management System

Constraints pervade the entire design process and thus, in many ways, constraint management (the process by which the consistency of the overall design is tested whenever design changes, additions or refinements are made) is a natural tool for design. The constraint management process described below rests firmly upon the assumption that an appropriate representation scheme (a hierarchy of frames) is used to represent the design process. The text below describes some of the functionality of a constraint management

system, and suggests appropriate sites for such a tool within the architecture.

At a minimum, a reasonable constraint management system should serve to alert the designer when constraints are violated and should be flexible enough that constraint alerting can be customized based upon the "hardness" of the constraint, the level/seniority of the designer, and the level of the design (prototyping/exploring vs. fine-tuning). In addition, it is necessary that the constraint management system be designed in such a way that appropriate pruning of the constraint search path be enabled, so as to reduce the amount of search necessary to alert the designer to constraint violations. Pruning can take place via such mechanisms as *if-changed demons* and *timeliness checking*.

Attractive enhancements to a constraint management system allow for a mixed-initiative style *explanation facility*. Such a mechanism would be capable (via inheritance) of providing the designer with a description of the process by which the constraint violation was detected. Such a facility would enable the designer to either more easily re-specify the proposed design or even to override the violation altogether.

Another useful enhancement involves assisting the designer via a simple mixed-initiative style dialogue. A rudimentary system could be designed via the process of constraint *inversion* in which ranges of acceptable design parameters (the range determined by the constraints imposed by other design features) are presented to the designer.

To summarize, there a variety of ways in which constraints can be processed. Utilizing demon invocation and constraint categorization facilitates the reduction of search to the extent that constraint processing becomes manageable, and constraint inversion and tracing support suggestion and explanation mechanisms which can be established in a mixed-initiative framework.

The constraint management system, as described above, is essentially an inference system applied to the frame-based representation of the design process. As such, the most appropriate location for the constraint management system is at the level at which the design itself is managed. If individual workstations support their own copies, then the workstations should also administer the constraint management tool.

History & Referencing System

The appropriate implementation of a frame based representation can make the hierarchical representation of the design process particularly conducive to the development (via inheritance and tree traversal) of a powerful history mechanism and a suggestive referencing mechanism. Such mechanisms would

enable a designer, upon creating (or returning to) a design frame, to request a historical trace of all of the design decisions which led to the creation of the current frame. If the representation was implemented on a bit-mapped display with mouse-sensitive capabilities, the designer could simply point to frames of interest, and obtain a detailed description of the frame's contents (slot-values, constraints, heuristic rules etc.). The ability to determine which tools and databases were accessed from a given frame could prove to be particularly profitable.

In addition to using the mechanism of inheritance for a design history tool, it can be used to develop a suggestive referencing tool in which a designer, when approaching a design task, could request information which described which tools and databases might prove useful at the current stage in the design. A trace through the design tree could identify those tool and databases which were accessed by frames above the current frame in the design tree. The desired information could be displayed quite conveniently in a spreadsheet format in which the rows (tools/databases of interest) were mouse-sensitive and could thus be expanded to reveal more detailed information about what frames accessed the tools/databases, and in what context.

Just as the designer could pose questions such as "What tools are appropriate to my current task?", a related capability of the suggestive referencing mechanism could enable the designer to select a potential tool and pose questions such as "What other tasks used this tool?". A simple traversal of the design tree could locate the appropriate design frames, which could then be graphically displayed on the workstation screen.

In general then, the representation of design decisions as a hierarchical network of frames can, with the use of both appropriate inference mechanisms and a sufficiently accessible user interface, prove to yield a powerful tool for both design and audit trail maintenance.

The history and referencing tool is intimately tied to the design itself and as such should reside at the level within the architecture at which the design itself is administered.

Tool Selection & Data Retrieval Expert System

Although the availability of a wide variety of analysis tools offers considerable potential as a mechanism for extending the power and capabilities of the designer/analyst, there are obvious hurdles to overcome. In particular, such tools must be *learned*. An analysis or database tool, regardless of its efficacy or power, is of little use if its interface needs are sufficiently complicated to inhibit use by the designer. The difficulty in using a compendium of analysis tools is further exacerbated when the inputs/outputs and order of execution of such tools is of some importance.

Towards this end, we recommend the development of a small expert system. We envision such an ES acting as a buffer between the designer and the tool itself, assisting with both the preparation of a tool for execution and the subsequent retrieval of analysis results.

Specifically, we propose the development of an ES which would provide the user with a frame-like template for the specification of the appropriate parameters required by the requested tool. Where possible, acceptable parameter ranges (based upon previous design constraints) will be presented and, as slots in the parameter template are filled, ranges for the other slots dynamically change as appropriate. In addition, when a particular tool is requested, a quick search (via the history and referencing mechanism described previously) could determine if similar requests to the tool had been previously issued, and the location of similar requests could greatly aid the specification of parameters, particularly for a complicated tool. If the specification of some tool parameters are obtainable only via the execution of other tools, a simple recursive backtracking mechanism can be used to dynamically set up a piping scheme for a sequence of tool executions. Once the appropriate parameters have been specified, the ES system is responsible for preparing the appropriate calling sequence and actually executing the job.

Once a tool is executing, the ES is responsible for retrieving results and displaying such results in a form suitable to the designer. Many complicated tools generate massive quantities of output, much of which is often of little use to the user. Via a mechanism involving the initial specification and subsequent location of an output template, the ES can process retrieved output and present it to the designer in an attractive and parsimonious representation. In addition, records of the tool request, execution and search result can be automatically inserted, thus simultaneously facilitating maintenance of an audit trail and the ease with which subsequent calls to the tool may be made.

Such an ES seems best suited for residence on the analysis workstation.

Interface Prototyping System

The rapid prototyping of interfaces is a necessary aspect. In order for design workstations to be effective their interfaces must be readily accepted by the designers. The interfaces must be easily understood, easily manipulated and, where possible, easily customized. It is also important that, to the extent possible, a common metaphor for the designer interface extend across all workstations. The AI iconic and window approach to user interface design has met with growing acceptance outside of the AI community and we recommend its adoption for the workstation designer interface. The interface prototyping process for displays is at once more critical and more

difficult. A wide variety of disparate display technologies is available for selection and thorough testing both through simulation and man-in-the-loop evaluation is crucial.

An object-oriented approach to display design and testing offers a number of advantages. The object-oriented metaphor is a natural choice when the domain to be modeled itself consists of a number of interacting components (objects). Furthermore, when the objects to be modeled can be classified into some hierarchical scheme, the potential for parsimonious knowledge representation via inheritance mechanisms becomes available. Consider, for example, a simple histogram display. One can easily imagine such a display object belonging to (and thus inheriting from) the general class of frequency distribution displays. In turn, a horizontal bargraph display object could inherit from the histogram display object and thus particular instantiations of horizontal bargraphs could be modeled with ease.

Of course the simple generation of static candidate displays, object-oriented or otherwise, is insufficient for display development needs. The candidate displays must be able to be tested, and this naturally requires that the candidate displays must be dynamic. A natural approach to the testing problem is to extend the object-oriented metaphor to include a message passing programming paradigm. In this type of system, objects (more accurately, *classes* of objects) are defined in terms of both their visual characteristics and the kind of actions they may take subsequent to the reception of appropriate messages. To return to the histogram example introduced above, one can imagine a single bar in a histogram belonging to a class of histogram bars which, among other things, can respond to a *grow n-units* message.

The metaphor of a candidate display built up as a hierarchy of objects, each of which can change its behavior subsequent to the reception of appropriate messages is a powerful one. Simulations can be readily built (the concept of object-oriented simulations is addressed in the next section) to drive candidate displays, and the hierarchical nature of the display definition readily supports a rapid prototyping display development environment.

A number of commercially available display development tools support the object-oriented metaphor. We has had a good deal of experience, for example, with a product known as the *Object-Oriented Graphical Modeling System* (OOGMS). This product, based upon the *SmallTalk-80* programming paradigm is written in C and runs on a variety of popular workstations (VAX, SUN, Apollo etc.). Accompanied by a powerful drawing package, OOGMS allows the user to design and classify complex hierarchies of graphical objects. The objects themselves can be assigned behaviors and via the reception of appropriate messages, the displays can be animated and controlled in real time.

Object Oriented Simulation

The prototyping and development of displays and human-computer interfaces, as described in the previous section, is potentially a very difficult and time consuming task. A crucial aspect to the interface testing process involves the ability to test displays via simulation. Treating the display elements themselves as addressable *objects* argues for an object oriented approach to simulation. The traditional approach to simulation design and construction create several problems for simulation development and maintenance. The procedurally-oriented programming paradigm with which simulation developers typically model their domain ensures that even carefully specified assumptions, dependencies and behaviors will become obscure as levels of detail are added to the developing simulation. Concomitantly, procedural simulations tend to be monolithic, unwieldy creations that are slow to build, awkward to modify and difficult to interpret.

These difficulties are, to some extent, resolved by an object-oriented, knowledge-based paradigm for simulation design. Domain experts often find the object-oriented metaphor to be particularly intuitive, especially when the domain to be modeled consists of numbers of autonomous, interacting entities.

Development, maintenance and verification of knowledge based simulations is facilitated by the separation of domain knowledge from control structure and by the use of inheritance as a mechanism for the parsimonious representation of knowledge. The message passing metaphor for simulation driving permits sophisticated tracing and debugging during development and provides a convenient mechanism for animating the simulation at run time.

Although object-oriented knowledge-based simulation is a powerful methodology, it is not without its problems. Current exemplars of the approach indicate that simulation development is a rather tedious undertaking. For example, the "natural-language" style of the ROSS language, although an attractive feature for the novice, becomes somewhat of an impediment as expertise accumulates. Other problems include the modeling of non-intentional events, the issue of grain size, the delegation of responsibility across objects, dynamic modification, and the problem of execution speed.

We have adopted a *systems-based* modeling approach which extends the object metaphor. Systems are objects with processing capabilities, such as an input-agenda and prioritization mechanism, an internal process with self-definition and message handlers, and a model of the environment. The systems-oriented approach solves many of the deficiencies of static and procedural objects by providing more computational power local to an object. Control of the simulation, rather than being a global characteristic, can be distributed to the values/priorities of the system-objects. Composite systems can be formed that permit control of the grain-size and abstraction level of the simulation. Dynamic modification of systems is the rule rather

than the exception, since systems are inherently dynamic. This approach permits a strong form of parallelism, distributing computational resources and responsibilities over the object-systems in the model. We have developed formal transformational procedures that permit principled construction of networks of systems. Theories and world models expressed in Predicate Calculus can be mapped onto these networks so that the semantics of the model is preserved while the control of inference is distributed. We are also committed to the dynamic visual display of system-objects and their interactions. We are currently developing a purely visual display language which represents computation as transformations on planar maps and networks.

In summary, we has developed a theory and an infrastructure to implement a distributed knowledge-based simulation environment based on dynamic objects (systems), with a parallel inference engine, a visual programming language and a network-oriented development interface.

The natural location for an object oriented simulation tool is alongside the interface/display prototyping system on the Display Development workstation.