BRUCE I. BLUM and VINCENT G. SIGILLITO

AN EXPERT SYSTEM FOR DESIGNING INFORMATION SYSTEMS

This article discusses the possibility of building an expert system that can design information systems, thereby automating the software development process. This can be done by means of what we term the Environment for System Building. First we describe the software process and identify those areas in which knowledge can be formulated algorithmically and those in which knowledge can best be processed by an expert system. Next we present an overview of our approach to implementation and give the current status of our research.

In the mid-1940s, "computer" was the job title of a person who used an electronic calculator. Today, that older meaning has become an anachronism. Moreover, computers are no longer restricted to the domain of mathematical calculation; information systems have grown out of data processing, and artificial intelligence is showing us ways to process (symbolic) knowledge.[1]

Along with the dramatic successes of computer technology have come problems in implementation because systems are more complex, the pool of experienced developers is limited, and the needs of many users are poorly articulated. Consequently, it is common for new books on software engineering or design methods to have a chapter called "The Software Crisis" or to explain why all who have not read that particular book are using the "wrong approach." Elsewhere in this issue of the *Johns Hopkins APL Technical Digest*, the promise and potential of artificial intelligence have been presented. This raises the question, Can the physician heal himself? That is, can we system designers build a system that captures our expertise in order to automate the software development process?

We think that the answer is yes, and we call our solution the Environment for System Building. In order to understand how it is to operate, we must first describe the process of developing the software. From this generic model we can identify those areas in which we can formulate our knowledge algorithmically and those that can best be expressed in a form that can be processed by an expert system. Using this framework, we then can define the Environment for System Building structure and the status of our research.

THE SOFTWARE PROCESS

If we are to address the issue of system development, we must first understand what we mean by the software process, i.e., the computer software life cycle. Our view of the process will define to a large extent the vocabulary that we use in controlling or directing the process. The most commonly cited view of the software process represents it as a cascading sequence of discrete activities (with feedback and iteration). This is the so-called waterfall diagram that implies a specific implementation approach derived from experience with equipment development. Figure 1 shows a generic model of the software process that is implementation independent.

Three transformations are identified:

1. The first transformation maps a set of needs in the real world onto a set of requirements called the problem statement (or specification). It is important to recognize that there are many possible mappings and that the problem statement is never complete. As Turski[2] points out, a complete specification would imply that all correct solutions are isomorphic to it and that the users fully understand their needs. Thus, he uses the term "permissive" to indicate that the problem statement specifies only the essential features and is otherwise incomplete.

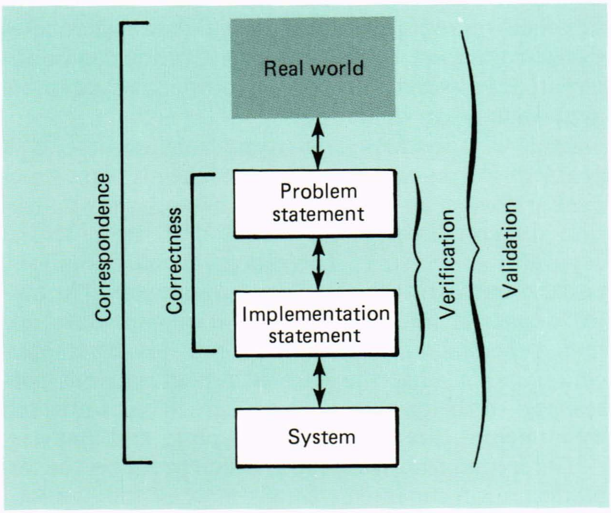

Figure 1—A generic model of the software process.

2. The second maps the problem statement into a complete and translatable implementation statement. The transformation assumes the presence of permissive specifications; it adds behaviors not defined by the problem statement. Many mappings are available. Over 90 percent of the software development cycle is concerned with this transformation and its associated decomposition of the problem into smaller, more manageable units.
3. The third produces the system from the implementation statement. This part of the process—called compilation, assembly, etc.—is well understood and fully automated.

Once the system exists, it is embedded in the real world, thereby modifying the real-world environment (and perhaps the validity of the problem statement). Thus, the figure represents only one iteration in the software process.

On the left of the figure are two concepts that measure the utility of the system. Correspondence measures how well the system satisfies (corresponds to) the needs of the real-world environment. It can be evaluated only after the system has been installed. Correctness, on the other hand, measures how well the implementation statement satisfies the problem statement. Many different systems can be equally correct with respect to a given problem statement. Correctness and correspondence are independent of each other; however, management of the software process links them.

On the right side of Fig. 1 are the two processes that control correctness and correspondence. Verification provides the measure of correctness and is initiated with the first formal specification; its goal is to assure that all derived products are consistent with the higher level statements. Validation serves as an estimator for correspondence. It begins before there are any formal statements and continues until the implementation statement is complete. Validation is the process by which agreement is reached on the problem statement and the permissible additions to the implementation statement are accepted. Stated another way, verification removes objective errors and validation eliminates subjective errors. Clearly, only verification can be automated. However, automated tools can assist in the management of validation.

Each of the three transformations represents a problem-solving activity. The first, the problem statement, relies on an understanding of both the application domain (in the real world) and the potential capability of a system. The process is called analysis, and the activity is essentially a cognitive one. The major danger is that the problem statement does not match the real-world needs. We call this the "application risk"—that the correct system will not correspond to the environment's needs. Risk is lowered by modeling, simulation, prototyping, and analysis.

The second transformation, which produces the implementation statement, entails most of the development life-cycle activity: preliminary design, detailed design, code, and test. It can be represented as an iterative sequence of development steps with the output of each step at another linguistic level,[3] e.g., module description, detailed design, Program Design Language, and, finally, code. Much of the process can be supported by software tools. The elements of risk here are that noncorresponding behavior will be introduced (application risk) or that, because of technical, management, financial, and/or schedule constraints, it will not be possible to produce the desired product (implementation risk).[4] The final transformation, production of the system, is fully automated. (A broader and more accurate definition would include issues of training, installation, documentation, etc., which this model avoids.)

Figure 2 displays the state of the art in terms of the two elements of risk. Projects high in both dimensions of risk are beyond the state of the art; simplifying assumptions must be made. When viewed from a problem-solving perspective, it can be seen that there are qualitative differences between projects that are high in one dimension of risk. High application risk projects involve acquisition of knowledge of the real world and the structuring of that knowledge to guide the software process. This typically involves considerable social interaction, and natural language is the key medium for communicating and formalizing knowledge. High implementation risk projects, on the other hand, begin with well-defined statements of the problem, e.g., many of the system interfaces are defined. In this case, problem solving relies more on technical knowledge, and there is a greater reliance on the abstractions, symbology, and jargon of the area of technology being targeted.

There are significant differences between products at the extremes of the two dimensions of risk. It would serve little purpose, therefore, to consider the problem of system development without distinguishing between the two. Since projects of high implementation risk are deeply embedded in the technological knowledge of their field or domain, development considerations will be closely bound to similar issues that are

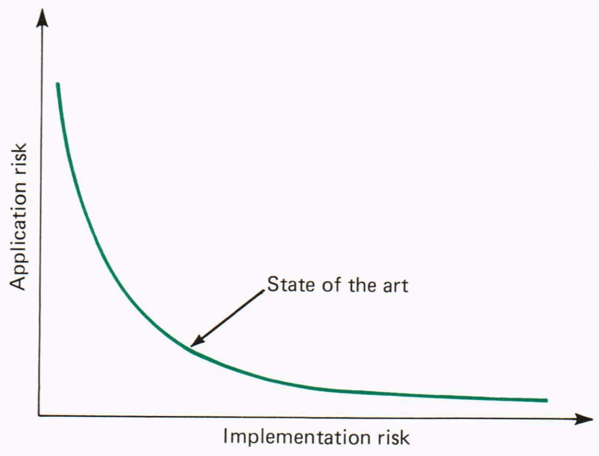

Figure 2—Two dimensions of risk.

restricted to that domain. Consequently, in what follows, we shall limit our discussion to the development of projects in which application risk is a significant factor. In particular, we shall consider interactive information systems. Such systems are characterized by their reliance on a database, a dependence on user inputs and actions, the absence of severe real-time or computational constraints, and an implementation as an organic (as opposed to embedded) system using off-the-shelf equipment and software tools. Their development can be characterized as the application of a well-understood technology and a poorly formulated methodology.

EXPERT SYSTEMS AND KNOWLEDGE DOMAINS

In the early days of artificial intelligence, there was a hope that the problem-solving process could be understood and formalized in a general problem solver.[5] In time, it was recognized that the "common sense" that humans have accumulated and seemingly use so effortlessly is far too complex to capture with our current state of understanding. Consequently, attention shifted to deeper and narrower knowledge domains that could be manipulated by an expert system.[6]

Figure 3 is a diagram of a simple expert system. At the lowest level are a knowledge base and a global database. The knowledge base contains the knowledge of the expert domain. In a production rule system,[7] the knowledge is represented as production rules:

$$\text{if } \langle \text{antecedent}_1, \ldots \text{antecedent}_i \rangle \text{ then } \langle \text{consequence}_1, \ldots \text{consequence}_j \rangle .$$

Other types of expert systems model the knowledge differently as, for example, a semantic network.

The organization of the knowledge base depends on the problem domain, the state of understanding, the complexity of the objectives, etc. Whatever its structure, however, the knowledge base contains the total domain knowledge available in the system. The global database, on the other hand, contains all information about the problem currently being solved. Thus, the knowledge base is problem independent and static, while the global database is problem specific and dynamic.

The inference engine is designed to search the knowledge base using information in the global database. It contains all control decisions. That is, unlike procedural programming, expert systems attempt to separate the knowledge from the way in which it will be used. For example, if there is a rule in a production rule system that

$$A_1 \text{ and } A_2 \rightarrow C_1 ,$$

and if A_1 and A_2 are true in the global database, then the inference engine will establish that C_1 is also true. It will be added to the global database for use as a fact in subsequent iterations. The inference engine usually has mechanisms to select among competing rules, deal with incompleteness, and improve search efficiency. It works by searching the knowledge base directed by the state of the global database until it reaches one of its goals (i.e., finds a desired answer) or determines that a solution cannot be found. In the latter case, the system may ask the user for more information.

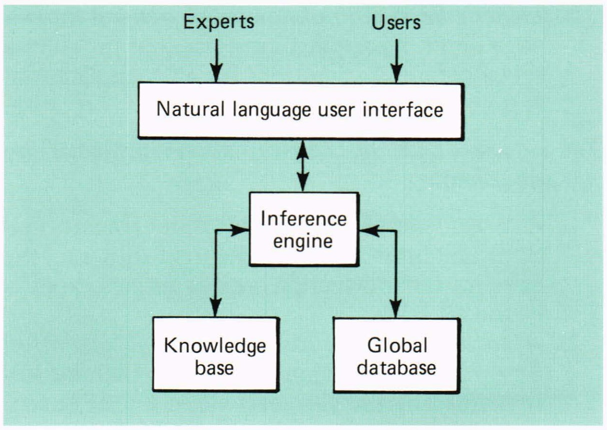

Figure 3—A simple expert system.

Because most expert systems store knowledge in symbolic form, there is general agreement that an expert system also should be able to explain its reasoning and interact with users through a natural language interface. With the present state of the art, however, natural language processing is a major research topic in its own right, and few, if any, expert systems support language processing beyond simple explanation facilities. Consequently, most interactions are managed by prompting scenarios that allow the experts to build the knowledge base and the users to enter data into the global database and interact with the inference engine.

Given this brief introduction to expert systems, let us now consider the knowledge domains appropriate to information systems development. Two may be identified:

- *Application knowledge,* about the application environment and its needs,
- *Transformation knowledge,* about the software process and the transformations used to produce correct and corresponding systems.

Note that these two domains overlap. Drawing on his experience in cognitive science, Curtis points out that "programming skill is specific to the application being considered."[8] That is why developers who build compilers may do a poor job at building an accounts receivable package and vice versa. Thus, it is useful to divide application knowledge into two subdomains:

1. *Application-specific knowledge,* knowledge that defines the specific application and sets it apart from all others. It is normally formalized as a requirements specification.

2. *Generic application knowledge,* general knowledge about an application class that provides a set of default assumptions for each application. It is normally considered experience or expertise.

One can also divide the transformation knowledge into two subdomains:

1. *Heuristic design knowledge,* about transformations that must be applied heuristically, i.e., the rules for transformation are not defined explicitly for all cases.
2. *Algorithmically prescribed transformation knowledge,* about transformations that are fully defined by the current state, e.g., the source code establishes a state that fully defines the transformations to be used by a compiler in producing the object code.

Given this decomposition of knowledge, it is clear that application-specific knowledge is very product dependent. It would be viewed as the global database by an expert system. The knowledge of algorithmically prescribed transformations, on the other hand, is well defined and understood. There is no need to include this knowledge in an expert system except for reasons of efficiency.

The remaining two knowledge subdomains, generic application knowledge and heuristic design knowledge, represent the types of knowledge ideally suited to an expert system's knowledge base.

1. Each is relatively narrow in scope;
2. No closed, formal representations of the knowledge already exist;
3. There is a degree of uncertainty in the use of the knowledge;
4. The knowledge base is incomplete and can be built and refined over time.

The following section describes how the four subdomains are being integrated into an environment to build information systems.

THE ENVIRONMENT FOR SYSTEM BUILDING

The previous two sections reviewed the software process and the architecture of expert systems. Three transformations and two categories of knowledge were identified. They may be combined as follows:

1. *Problem statement definition,* a transformation from the application-specific knowledge domain into a specification. It is a problem-solving activity that varies with each project.
2. *Implementation statement definition,* a transformation from an initial specification into a specification that can be implemented as executable code. There is generality across applications, and the design judgments depend on generic application knowledge and heuristic design knowledge.
3. *Implementation transformation,* a transformation of the implementation statement into executable code. It may be done by an assembler, a compiler, or a program generator. The higher the linguistic level for the input, the more complex this transformation and the less complex the previous transformation. In any case, the transformations are well defined in the domain of algorithmically prescribed knowledge.

Clearly, the first two transformations are domain dependent, and any approach to automating the process will require different environments for different application classes.

The Environment for System Building (ESB) is being developed for a single application class: the interactive information system. This target class was selected because (a) the transformations used in implementation are relatively well understood,[9] (b) there is a good understanding of database technology[10] and modeling,[11] (c) an implementation tool with a high-level specification language is available to manage the implementation transformation,[12,13] and (d) most products have a high application risk, a fact that makes them well suited to a dynamic, interactive development environment.

As the previous analysis suggests, ESB is organized as three modules:

1. *The definition module,* tools to capture the specification. These tools are an example of an interactive information system and should be implementable with ESB.
2. *The transformation module,* an expert system to transform the specification into an executable specification.
3. *The generation module,* a program generator to transform the executable specification into an operational program.

The definition module captures the knowledge of the application domain that establishes the behavior of the target product. It is viewed as a conceptual modeling tool for designers and analysts. The transformation module is an expert system that uses the database created by the first module as its global database. Its knowledge base is limited to the technology domain as augmented by some generalized application domain knowledge from the interactive information system. The generation module relies on an existing tool with a specific very-high-level language.

Figure 4 shows how these modules are integrated in ESB. The designer interacts with the definition module to produce an initial specification. As segments of the specification become complete, they are processed by the transformation module. This is an expert system that uses a transformation knowledge base to act on the knowledge structured in the application knowledge base; the output is an executable specification. The generation module translates the executable specification into an operational product. The initial generations serve as prototypes that provide feedback to the

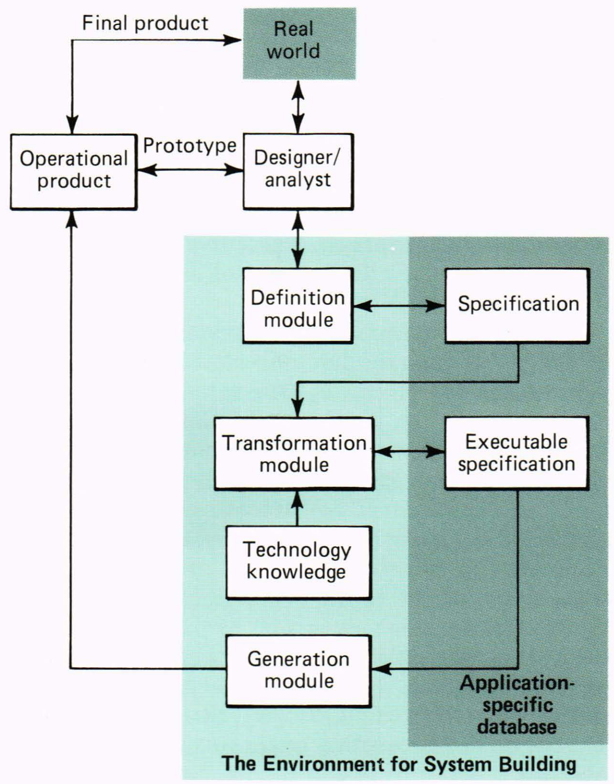

Figure 4—Functional diagram of the Environment for System Building.

designer. (Not shown in the figure is backtracking to resolve issues of incompleteness or contradiction.) A validated version of the generator output serves as the operational product.

Note that the user is consistently referred to as a designer or analyst. This implies that the goal of ESB is to produce interactive information systems that are more complex than those generated by the application development products intended for naive users. (Since there are many commercial products that support the latter process, there are few research issues associated with such a limited goal.) The use of the terms designer and analyst also implies that ESB is not concerned with programming. All ESB user activities involve the building or evaluation of an application-specific knowledge base. Programming is relegated to the generation module. In this sense, ESB differs from expert systems designed to act as a programmer's apprentice.

We now examine each module in further detail.

The Definition Module

The purpose of the definition module is to capture the users' understanding of what the target system is to do. It is a passive module in that it does not perform any decision making; it simply supports the user during the definition process. To be effective, the definition module must

1. Maintain a structure that contains a natural (with respect to the application class of an interactive information system) model of the target system;
2. Provide easy-to-use tools for entry or perusal of information about the target system;
3. Impose consistency checks and minimize the need for redundant entry.

In order to understand how the definition module performs these tasks, one must first consider how the target application is modeled.

Each target system involves three levels as shown in Fig. 5. The top level is descriptive. This implies that it is informal and not machine interpretable. The knowledge contained in the descriptive level is intended to assist the designers in organizing their thoughts; feed back preliminary system descriptions to other users; and establish a road map to the system that can be used during design and, more importantly, during evolution (i.e., maintenance). The second level contains conceptual models to be used in creating the target system. These represent the formal specifications in which the knowledge of the target system is organized as conceptual objects; where objects overlap, the designer is provided different views of the object for each context. The third, and lowest, level contains the implemented objects. As shown, there are algorithms and data structures within programs as well as data structures within the database. Also indicated in the figure is the fact that all interactive information systems must manage user interactions and concurrency.

At the descriptive level, three objects have been identified:

1. *Requirements,* a statement of what the target system is to do;
2. *Data groups,* generic descriptions of the major entities that are to be represented by data stored in the database;
3. *Processes,* generic descriptions of user interactions, data transformations, or system outputs.

Most interactive information system development methods assume that the requirements are prepared first and that the data groups and processes are derived from them.[14] In high application risk projects, however, the requirements may not be known. In many cases, development begins with only some of the general system objectives established. In those situations, it may be appropriate to bypass the requirements and begin to define the data groups and processes in the application domain. While such an approach may be anathema to the conventional software engineering process, it nevertheless represents a realistic approach for binding the designer's understanding of the application domain.[15] It is also consistent with the use of a fully automated environment for managing the application knowledge.

The requirements, data groups, and processes are represented as hierarchies with the nodes ordered at each level. This facilitates their listing in outline form. The goal of defining the requirements, data groups, and processes is not to establish the system's structure; rather, it is to provide an informal method for identifying potential system components. Each item may

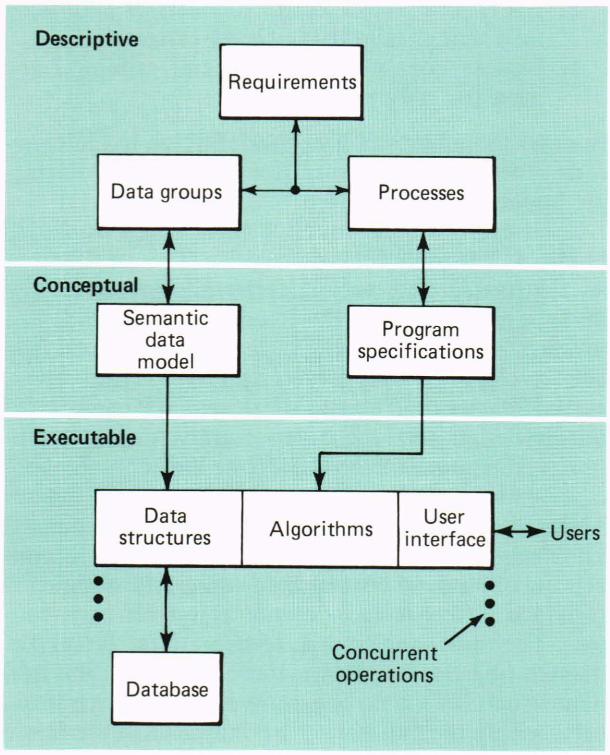

Figure 5—Representations of an interactive information system.

be linked to items in another category. For example, data flow is represented as interactions between data groups and processes. Requirement tracing is performed by tracing links from a requirement to data groups and processes; every data group and process (or its antecedent) must be linked to a requirement node. Obviously, the full network of hierarchies and links is too complex to list; however, local views are available on demand.

Because the descriptive statements are informal text statements, a more precise definition is required to produce an application knowledge base that can be converted into an operational system. This is the function of the conceptual level of the model. Again, two implementation objects are defined: data and programs. Current thinking about conceptual models suggests that there are three basic approaches to the development of "descriptions of a world/enterprise/slice of reality which correspond *directly and naturally* to our own conceptualizations of the objects of these descriptions."[16] These are:

- Knowledge representation (artificial intelligence)
- Semantic data models, e.g., the entity-relationship model[17]
- Program specification, e.g., abstraction

The choice of technique will depend on how well structured our knowledge of the domain is.

In the case of the interactive information system, the activities of the target system can be viewed as a set of interactions with the users and the database. The structure of the database should be derived from (a) a model of entities for which it will provide surrogates and (b) the externally defined interfaces that the system must satisfy. The processes then can be abstracted as (c) algorithms that transform data (e.g., compute a FICA deduction), (d) interactions with users (e.g., dialog management, screen generation), or (e) non-procedural definitions of complete outputs (e.g., reports, graphic displays). These five conceptual objects define a generalized model of an interactive information system.[18]

Definition of objects at the conceptual level requires navigating through the descriptive level in order to provide traceability. However, once at the conceptual level, the designer can—and usually will—produce (permissive) details that were not implied at any higher level. Consequently, ESB must provide an environment in which the designer may record any information about the behavior of the target system at any level of detail without regard to the uniformity of descriptions across levels. In the context of a top-down design discipline, this may appear to be chaotic, but in the ESB view, the definition/modeling process is seen as a binding of real-world desires and constraints. Consistent with the ESB philosophy, this binding is seen as a cognitive process that ought to apply a cognitive, and not a data processing model.[19]

The Transformation Module

The transformation module is an expert system. It differs from most expert systems in that it is not designed to interact with users to guide them in making more expert decisions. Rather, it interacts with the specification (i.e., application knowledge base) constructed by the definition module. It uses the expert knowledge residing in the transformation knowledge base to produce executable specifications and feedback requests for additional specification information. PUFF is an example of an operational expert system that performs in this closed loop fashion; in this case, the inputs are physiological performance data from sensors, and the output is a pulmonary function report.[20] ESB is far more complex than PUFF because its global database is considerably larger and it must be retained over a much longer period, years rather than minutes.

The definition module collects both the system's conceptual data model and its processing flows. It is organized, however, to allow the designer to consider the system in small, local views. It is the responsibility of the transformation module to confirm consistency, determine global requirements, and resolve internal clashes. For example, in business systems, employees are viewed as different entities by the payroll department, the benefits office, and their managers. For each view, the designer may produce different definitions in the semantic data model. These definitions may reuse existing terminology, e.g., EMPLOYEE.NUMBER and EMPLOYEE.NAME, but they will not be constrained by alternate views. If there is consistency, e.g., all views require an employee number, name, age, sex, job title, and years of service, then there will be a high confidence in the resulting entity definition. On the other hand, where there are attributes associated with only one view, there is a strong indication that an entity (or group of attributes) must be defined for that specific view (or set of functions).

The availability of global knowledge also aids in the definition of an efficient prototype. To illustrate, assume that we have an EMPLOYEE attribute that is keyed by NUMBER. Moreover, we have processes that include the statement

Find EMPLOYEE using NAME.

These facts, when guided by the proper heuristics, imply the need for:

- An inverted file on EMPLOYEE by NAME
- A file management program to maintain the inverted file
- A program to accept a partial name, scan through the inverted file, and, on successful completion, return the NAME and NUMBER

The availability of global knowledge also allows for global consistency checks. For example, chains of utilization may suggest that EMPLOYEES and WORKERS are the same (or similar) entities. Questions would then be framed to the designer so that additional information would be added to resolve the uncertainty.

The output of the transformation module is a formal specification that can be used by a program generator. The format is consistent with that of the semantic data model and program specifications. It differs from them in that it is complete with respect to implementation. That is, permissible behaviors have been added and all implementation decisions have been made at this point; the operational product will be determined fully by the executable specifications. Some of the executable specifications will be transparent translations of the user's definitions; other portions of the specifications will have been created as the result of applying the rules in the knowledge base.

The Generation Module

ESB uses TEDIUM*, which is an environment for implementing interactive information systems, as the program generator. It has been described and evaluated elsewhere.[12,13] It will be sufficient to note here that it has been in use since 1980 and has been used to implement several sophisticated clinical information systems, as well as itself. The largest system developed with this tool operates at the Johns Hopkins Oncology Center, where it runs on networked PDP-11/70s, supports 60 clinical terminals, manages a database of a quarter million therapy days, and actively assists in the clinical management of 2000 patients at any one time.[21,22,23] The generated system contains 5000 programs and the data model is defined by 1000 relations.

There have been several evaluations of TEDIUM.[13,24,25] It has been demonstrated that TEDIUM applications are about four times more compact than those written in MUMPS and 20 times more compact than those in COBOL. A frequently cited one-semester problem (requiring from 325 to 771 hours of effort) was implemented in 28 hours using TEDIUM. Other studies have shown that the TEDIUM specification language is expressive and stable; i.e., it requires very few iterations to arrive at the desired design.

If TEDIUM is an efficient, robust environment for designing interactive information systems, then what is the need for ESB? Although TEDIUM is an order-of-magnitude improvement over some development environments, the user still must make all design decisions. In the ESB framework, however, the user is asked to view the definition only in the application domain. The expert system (transformation module) will resolve the lower level design decisions and produce TEDIUM specifications. This should result in more consistent designs, less manipulation of detail, fewer errors, and a formalization of the design decision process. All of this is outside the scope of TEDIUM.

*TEDIUM is a registered trademark of Tedious Enterprises, Inc.

STATUS AND CONCLUSION

Work on the ESB project started at the end of 1984. The initial work included concept formalization and manual simulations. Several papers[4,18,19,25] were prepared that defined the software process, the conceptual modeling of systems, and the structure of ESB and the tools it will use. A prototype of the implementation module was also produced.

We are now refining the implementation module and working on prototypes of the transformation module (i.e., the expert system). We expect that ESB will soon be complete and robust enough to generate nontrivial interactive information system applications. If we are successful in that, we will have produced a research environment for testing and evaluating alternate paradigms for software development. While it is premature to speculate about where such ESB research might lead, we note that, if the concepts can be demonstrated in the interactive information systems application area, they may be tested on other application classes such as mission-critical embedded systems.

In summary, we again emphasize that ESB is a research project. At the very least, it provides an environment in which to learn more about the software process and design heuristics. However, it also offers a potential tool that may improve productivity in software development dramatically. Work to date has been encouraging, but the project is far too young to make any predictions of its ultimate success.

REFERENCES

[1] P. H. Winston, *Artificial Intelligence,* 2nd ed., Addison-Wesley (1984).
[2] W. M. Turski, "Completeness and Executability of Specifications: Two Confusing Notions," *Software Process Workshop,* IEEE Comp. Soc. Press, pp. 155-156 (1984).
[3] M. M. Lehman, V. Stenning, and W. M. Turski, "Another Look at Software Design Methodology," *ACM SIGSOFT SEN* **9**, 38-53 (1984).
[4] B. I. Blum, "On How We Get Invalid Systems," *Workshop on Software Specification and Design,* IEEE Comp. Soc. Press, pp. 20-21 (1985).
[5] A. Newell and H. A. Simon, "GPS, A Program that Simulates Human

Thought," in *Computers and Thought,* Feigenbaum and Feldman, eds. (1983).

[6]F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, eds., *Building Expert Systems,* Addison-Wesley (1983).

[7]G. B. Buchanan and E. H. Shortliffe, eds., *Rule-Based Expert Systems,* Addison-Wesley (1984).

[8]B. Curtis, "Fifteen Years of Psychology in Software Engineering: Individual Differences and Cognitive Science," in *Proc. 7th Int. Conf. on Software Engineering,* IEEE Comp. Soc. Press, p. 100 (1984).

[9]E. Horowitz, A. Kemper, and B. Narasimhan, "A Survey of Application Generators," *Software* **2,** 40-54 (1985).

[10]G. Wiederhold, *Database Design,* 2nd ed., McGraw-Hill (1983).

[11]M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, eds., *On Conceptual Modeling,* Springer-Verlag (1984).

[12]B. I. Blum, "A Tool for Developing Information Systems," in *Automated Tools for Information System Design,* H. J. Schneider and A. I. Wasserman, eds., North-Holland, pp. 215-235 (1982).

[13]B. I. Blum, "Experience with an Automated Generator of Information Systems," in *Int. Symp. on New Directions in Computing,* IEEE Comp. Soc. Press, pp. 138-147 (1985).

[14]R. S. Pressman, *Software Engineering: A Practitioner's Approach,* McGraw-Hill (1982).

[15]B. I. Blum, "The Life Cycle—A Debate Over Alternate Models," *ACM SIGSOFT SEN* **7,** 18-20 (1982).

[16]Brodie et al., op cit., pp. 11-12.

[17]P. Chen, "The Entity-Relationship Model: Toward a Unifying View of Data," *Trans. Data Sys.* **1,** 9-33 (1976).

[18]B. I. Blum and V. G. Sigillito, "Knowledge-Directed System Development," in *Proc. 24th Annual Technical Symp.,* Washington Chapter ACM, pp. 97-102 (1985).

[19]B. I. Blum and V. G. Sigillito, "Some Philosophic Foundations for an Environment for System Building," in *ACM Annual Conference,* pp. 516-524 (1985).

[20]J. S. Aikins, J. C. Kunz, E. H. Shortliffe, and R. J. Fallat, "PUFF: An Expert System for Interpretation of Pulmonary Function Data," *Comp. Biomed. Res.* **16,** 199-208 (1983).

[21]R. E. Lenhard, Jr., B. I. Blum, J. M. Sunderland, H. G. Braine, and R. Saral, "The Johns Hopkins Oncology Clinical Information System," *J. Med. Sys.* **7,** 147-174 (1983).

[22]R. E. Lenhard, Jr., and B. I. Blum, "Practical Applications of OCIS, A Clinical Information System for Oncology," *Comp. Bio. Med.* **14,** 15-23 (1984).

[23]B. I. Blum, "Information Systems at the Johns Hopkins Hospital," *Johns Hopkins APL Tech. Dig.* **4,** 104-117 (1983).

[24]B. I. Blum, "MUMPS, TEDIUM, and Productivity," in *Proc. MEDCOMP,* IEEE Comp. Soc. Press, pp. 200-209 (1982).

[25]B. I. Blum, "Iterative Development of Information Systems: A Case Study," in *Software Practice and Experience* (in press).

THE AUTHORS

BRUCE I. BLUM (right) is a mathematician in the Milton S. Eisenhower Research Center. Born in Brooklyn in 1931, he obtained a B.S. degree from Rutgers University (1951), an M.A. degree in history from Columbia University (1955), and an M.A. degree in mathematics from the University of Maryland (1964). He joined APL in 1962, where he was assigned to the Computing Center. During 1967-74, he worked in private industry, and in 1974 he joined APL's Fleet Systems Department. From 1977-83, he worked full time at the Johns Hopkins School of Medicine as director of the Clinical Information Systems Division. In 1986, he tranferred to the Research Center, where he is working on projects in software engineering and artificial intelligence.

VINCENT G. SIGILLITO's biography can be found on p. 18.