An information system for developing information systems

by BRUCE I. BLUM
The Johns Hopkins University
Baltimore, Maryland

ABSTRACT

This paper considers a paradigm for the development of software through the use of a complete, integrated-design database. A prototype information system for the implementation of information management systems is then described. This tool manages the requirement definitions, the process and data flow, and all design and operational documentation. Using a relational-data model and program specifications, it generates programs in a selected target language. The results of two years’ production use are presented, and the potential for transfer of the concept is examined.
INTRODUCTION

One convenient mechanism for generalizing about progress in computer technology is to speak of generations. Thus, the Japanese are developing a fifth-generation computer\(^1\) that will be an order of magnitude of improvement over the von Neumann computer, and language developers are working on fourth-generation languages\(^2\) that will result in an order of magnitude of improvement in productivity over conventional high-order languages.

If we follow this analogy, we should be looking to a fourth-generation development environment. The first generation required hands-on manipulation of code and data, the second generation used punched cards and their electronic equivalent, and the third generation added libraries of tools and databases to support aspects of the life cycle. In this context, a fourth-generation development environment would provide uniform, integrated support to the full life cycle development process. This includes, of course, the maintenance and configuration management activities that constitute roughly half the total system cost.

Since the end result of computer system development is a machine-sensible product, one would expect practitioners in the field to have developed full-support automated tools. Unfortunately, this is not the case. Source programs are still retained as modified card images in files. Access to and control of programs are normally left to general file management programs. The contents of the program text are usually managed by the compiler with little analyzed data preserved in machine-sensible form. The link loader manages object programs without providing much system structure information. Standalone and integrated packages may manage the database or data dictionaries. Word processing is used to record requirements and specification documentation. Independent tools may support project management, test reporting, configuration management, and so on.

Figure 1 illustrates the role of automation in the current production model. Most machine-sensible products are not created until the process of coding begins. Coding is complete (for a given build) before testing begins; the coded product is used during operations. Superimposed upon the coded product is a nonintegrated, machine-sensible adjunct to the product. This adjunct includes text-processed documentation, independent reporting tools, standalone specification data, and so on. Clearly, our goals should be directed toward an integrated support environment that manages all information for each step of the process and links data among development processes. This is called the fully automated paradigm; as shown in Figure 1, it treats the system design information (requirements, specification, code, test data, etc.) as a single integrated database.

In what follows, a prototype fourth-generation development environment is presented. This system, The Environment for Developing Information and Utility Machines (TEDIUM\(^*\), a trademark of Tedious Enterprises, Inc.) has been in production use at the Johns Hopkins Medical Institutions for over two years. The tool is directed toward a specific application set: the information management system (IMS). The IMS is characterized by its use of commercially available hardware and software, its reliance upon a database, and its lack of realtime demands beyond those normally associated with interactive processing. The useful life of an IMS normally is 5 to 10 years; perhaps 75 percent of all commercial EDP activity is related to IMS projects.

The IMS provides a convenient target for a fourth-generation design environment. There is little technical risk associated with a project. That is, because of its reliance upon off-the-shelf products, there is little doubt whether the technology will be able to support the application. There is, however, great application risk—that is, risk that the finished project will not meet the users' needs. This failure to produce acceptable results is a major problem in IMS development. For example, surveys show that 52% of the requested applications are backlogged for two or more years.\(^3\) Further, one report indicated that of the software undertaken, 75% was never completed or not used if completed.\(^4\)

Figure 2 presents an overview of the state-of-the-art with respect to these two axes of risk. While a fully automated paradigm may be appropriate for all system development, it is reasonable to narrow the scope for prototype development. In the case of TEDIUM, the application area is the IMS in general and clinical information systems in particular. The latter manage clinical data in support of patient care. The

\(^1\) von Neumann
\(^2\) High-order
\(^*\) TEDIUM is a trademark of Tedious Enterprises, Inc.
primary challenge is the definition and refinement of requirements in a dynamic user environment. Once the needs are established, there is little difficulty in transforming them into a final product. (Of course, the previously cited performance statistics suggest that although implementation may not be difficult, mere implementation does not imply that there will be a satisfactory product.)

METHOD

TEDIUM is a comprehensive methodology and environment intended to support the development and maintenance of an IMS. It operates upon a database containing all the machine-sensible information about the target IMS. In this sense, it is an information system for developing information systems. As an IMS, TEDIUM is implemented using TEDIUM.

The target applications for which TEDIUM was designed are moderate to large IMSs with heavy user interaction, a limited understanding of the final requirements, and sophisticated data structures. These are custom-built systems that require from one-half to twenty man-years of effort. Though TEDIUM clearly has the ability to act as a report generator or "programmerless" application generator, its design capabilities are considerably broader. Various aspects of the system and its implementation techniques have been described in detail elsewhere. The remainder of this section contains an overview of TEDIUM and provides references for further reading.

A model of the IMS development life cycle is presented in Figure 3. Although it resembles the traditional waterfall diagram, this schematic shows the process as a series of transformations between environments. The underlying concept is that we must first understand each operational environment before we can develop the tools needed to support the transformation of knowledge from one environment to another.

The application environment is the arena in which the system will be used. The needs of this environment establish what is to be done; its procedures are generally altered by the delivered IMS. (This is a combination of the Heisenberg Uncertainty Principle and a fundamental restructuring of the environment.)

The analyst's description of what the IMS is to do is created in the development environment. The description usually consists of text and diagrams limited to the scope of the IMS. Part
of the description is used only during the development phase; other parts (e.g., user manuals) are required throughout the life cycle.

In the design environment there is a translation of the description that tells what is to be done into a specification of how it is to be done. The form may be either text or a formal specification language. This specification is translated into instructions that can be executed. The process is an expansion of the design into code (commonly called programming). This function is carried out in the implementation environment. It is supported by the system operation environment that represents the hardware and software (e.g., computer and operating system) used for testing and the operation of the finished application system. Not shown in the figure are the necessary feedback loops or overlaps due to phased implementation.

In this development model there are five major transformations. The first is from the application to the description. It is independent of programming language, operating system, and computer equipment. The description is typically documented as text that is partially maintained throughout the system's life cycle. The next transformation is from the descriptive text to the design specification. If a formal specification language is used, this is a translation from a text document to an unambiguous description of how the system shall be implemented. (Some software engineering documents limit the role of formal specifications to the requirements that simply describe what is to be done. It is now generally recognized that it is impossible to always separate structure from behavior; thus this convention is not used with TEDIUM.)

The design specifications are based only upon the description and are independent of the application environment. They also are generally independent of the implementation language and target system. The next two transformations involve the conversion of the design to an implementable product and a test of that product in a system operations environment. For the target applications, there is little technical risk in these transformations. The major concern is that the design specifications satisfy the application environment needs.

The final transformation is system maintenance. System maintenance consists of (1) correction of errors and (2) changes and expansions to the system. In either case the process involves each activity required during the initial design. For system maintenance, however, the problem may be complicated by the fact that key analysts and documentation are not available. The model shows system maintenance as a feedback loop that restarts the cycle. (This flow is described in greater detail by Blum.)

TEDIUM uses this development model to structure the design process. It begins with a statement of requirements that is developed within the application environment. These requirements are produced by the users and application designer. Though in some cases diagrams may be used, the requirements are normally text descriptions. Where the requirements are well understood, they are written in document format; for less well-understood requirements, an outline frequently suffices. The process of requirement definition is intended to document mutual understandings and to work toward the establishment of the users' true needs. The system requirements, therefore, do not act as a fixed document that initiates the design activity (i.e., a "build to" specification). Rather, the requirements represent an initial step in a continuing user-designer dialogue. (The overall process is called system sculpture and is further described by Blum and Houghton.)

Once the requirements for an application (or build) are established, the requirements are decomposed into processes and data groups. There is no algorithm for the allocation of requirements to processes and data; it is a highly subjective and personal activity. Of course, the general principles of structured analysis apply. One requirement may initiate many processes (data groups); one process (data group) may satisfy many requirements. Following the examples of Jackson and Warnier, we have made the notation for processes and data symmetric. Depending upon the application, the designer initially may elect to concern himself only with processes or data groups. Both processes and data groups are documented as text. Again, outlines may be used where uncertainty exists.

Although there is freedom to work within the design methodology, the final design of an application must reach equilibrium and completeness. This is accomplished when all requirements are satisfied by at least one process or data group, all processes and data groups satisfy at least one requirement, and a data flow exists that includes each process and data group. And finally, the requirements, processes, and data groups must have been organized in hierarchical structures (outlines).

The activity just described represents the transformation from the application environment to the description environment. TEDIUM supports this process by managing the hierarchy nodes and text of the requirements, processes, and data groups. It also provides links between the various entities and thus controls the descriptive network. Text recorded in the descriptive environment may be flagged with respect to importance (great, moderate, detailed) and potential audience (general, user, programmer, operator). TEDIUM provides tools to extract and organize text based upon class, importance, and audience in order to produce general documents and manuals. Thus, the requirements and design data are seen as an initial step in a continuing process of refinement that produces the final application documents. (The documentation process and data structures used to support it are presented in greater detail by Blum.)

Once the descriptive environment material is available, the transformation to the design environment begins. Processes are decomposed into executable entities (programs) and data groups are decomposed as relations (tables). Again, programs and tables are linked to the processes and data groups that spawned them. This many-to-many map is also maintained by TEDIUM. It facilitates maintenance by directly linking any program or table with the system elements or requirements that it affects. Of course, the validity of these links depends on the judgment and dedication of the application designer. Though the network can get out of date, the relative ease of update and the ability to produce cross-references to highlight inconsistencies represent improvements over traditional documentation methods.

Figure 4 summarizes the linkage of the various descriptive
and design components. Several points should be noted. First, the documentation process is not mandatory. One can define programs and tables without starting with requirements. For small, limited-use applications, written requirements are not prepared. Second, the definition network allows the designer to conceive of the system as a network with neither top nor bottom, even though processes and data groups are represented as a hierarchy—that is, data flow is possible between data groups at one level of depth and processes at another. Consequently, data flow diagrams that impose a system hierarchy upon the data-process interactions may be managed by TEDIUM; however, they are the exception rather than the rule when the TEDIUM design methodology is used. (A more complete discussion of the design methodology with sample descriptive outputs is given by Blum and Brunn.)

Once the programs and tables have been identified they can be specified. An underlying axiom of TEDIUM is that it should minimize duplication and provide immediate access to all established application design information. This is most easily illustrated by considering the process of table definition. (Because TEDIUM relies heavily upon user interaction, the term table was selected in place of the more threatening term relation.)

The TEDIUM data model uses tables with multiple, variable-length keys (index elements) and variable-length records containing zero or more data elements. Unlike a true relational model, it always orders the entries in a table by index element. (This knowledge of access order is essential when one is producing applications for small systems with no DBMS facilities.)

The short notation used by TEDIUM is

\[
PATIENT(PATID) = \text{NAME, AGE, RACE, SEX}
\]

where \(\text{PATIENT} \) is the table name, \(\text{PATID} \) is an index term, and \(\text{NAME}, \text{AGE}, \text{RACE}, \text{and} \ \text{SEX} \) are data terms. The underlining of \(\text{PATID} \) indicates that it is a defined term. That is, the table \(\text{PATIENT} \) will act as a dictionary for values of \(\text{PATID} \). All attempts to enter a value of \(\text{PATID} \) that is not already in the table \(\text{PATIENT} \) will be rejected (unless, of course, this feature is overridden within a specification.)

The TEDIUM data model also allows the creation of secondary tables that are subsets of a primary table. For example, the table

\[
\text{PATNAME}(\text{NAME}, \text{PATID})
\]

is a name index to \(\text{PATIENT} \). (\(\text{PATID} \) is required to provide uniqueness for patients with the same name. There are no data elements.)

The process of defining these two tables can be done in a five-minute interaction session. Once the table definition option is selected, the table name is prompted for. If it has not yet been defined, it is added to the database. Descriptive text information about the table is also requested; this may be supplied now or added at a later date. For new tables, TEDIUM prompts for the index and data elements. Again, if the element has not been defined, it is added to the data dictionary. Element definitions contain an external name (used for prompts), the data type and format, optional validation criteria, and text description. After the data element has been established, the system allows the user to add attributes to the element within this table, that is, defined, mandatory, optional, defaulted to a given value, initialized as a given value, and so on. (A more complete description of this process plus a sample dialogue and printed table listings are contained in another paper.)

The TEDIUM schema contains all table and data element definitions. The definitions include both formal parameters and free text. (User requests for information about an element first display the text; a subsequent request produces the formal parameters and validation criteria.) The data types include fixed and variable-length strings, integer and real numbers, date and time. There is also a text data type that calls in the text processor for each input request and the text formatter for all writes. The TEDIUM requirement table, for example, has the following general format:

\[
\text{REQUIRE}(\text{APPLICATION, REQUID}) = \text{REQNAME, REQTEXT}
\]

where \(\text{REQNAME} \) is defined as a variable-length character string of maximum length 120 and \(\text{REQTEXT} \) is a text data element of arbitrary length.

Programs in TEDIUM are defined by use of minimal specifications. The minimal specification is the least amount of information needed to produce the program within the context of a given system style and the design database. That is, one should be able to give a programmer a minimal specification plus the schema and a style manual and, without any further interaction, receive a complete and correct program that fully satisfies the specification and style requirements. For example, given the system style currently in use at the Johns Hopkins Medical Institutions, the specification for a program that would add, edit, delete, list, change index values, and produce table listings for the table \(\text{PATIENT} \) and also maintain the consistency between the tables \(\text{PATIENT} \) and \(\text{PATNAME} \) would be

Entry program for the table \(\text{PATIENT} \)
The "Entry" program is an illustration of a generic program. It is a template that accepts specific required and optional commands in order to fully define a program. Another example of a generic specification is that for a program to search through the name index (PATNAME), list out all patients who match a given root name, allow the user to select a patient (or indicate that no patient was found), and return the value of the selected patient's NAME and PATID (or the not-found code). This would be written

Prompt program for table PATNAME
Input is NAME
Output is NAME,PATID

In each of these cases the program specification is given as nonprocedural statements. Unfortunately, few applications can be specified exclusively through the use of nonprocedural statements. Thus TEDIUM provides the TEDIUM language in order to augment generic specifications or to create complete specifications.

Commands in the TEDIUM language tend to be housekeeping-free and application oriented. For example, in the JHMI style, the PRompt command

PR CONTINUE PROCESSING (Y/N)

will print the string CONTINUE PROCESSING (Y/N) and read an input. If the input is null, it is set to Y. If the input is the escape character, an abnormal program return is made. If the input is a question mark, then a help message is printed. (If none exists, then [NO HELP MESSAGE AVAILABLE] is printed. The user has tools to add or alter help messages after system delivery.) If the input is not Y, N, or any of the above, then the system prints [ERROR] and repeats the prompt. Note how the use of this command assures both completeness and consistency in the final product. (Alternate styles might accept Y, YE, YES, NO function keys, etc.)

All TEMON commands are written as five fields:

- An optional label
- An optional guard that contains a predicate which must be true if the statement (or block started by the statement) is to be executed
- A mandatory operand
- Parameters for the operand
- Optional control fields that indicate where control should be sent after normal or abnormal execution of operand

This command structure builds a GOTO into each statement. Although this may seem to violate current thinking, it is effective for TEMON specifications because

- Specifications are compact and short: normally one CRT screen-full, seldom longer than an 8½ × 11" printed page.
- All procedural statements are always printed in a box with labels outside the box to the left and control statements outside the box to the right. With this format, control flow and block structure are much easier to recognize than in the more commonly used nested style.

All generic programs are written as expansions of TEMON statements and nonprocedural inputs. Nonprocedural statements are used for entry and exit assertions. All generic programs allow the insertion of TEMON commands at key points in their flow. Thus one can create a generic program and modify it with custom statements that will alter the flow, function, or interface. A common practice in the TEMON environment is to generate a program with the desired functionality quickly and then, based upon designer and user experience with the interactive flow, modify the program; hence the term system sculpture. (A more complete discussion of the specifications, system style, the command language, and the method for creating generic programs may be found elsewhere.)

Programs are defined as a program specification and a frame specification. The frame is a generalization of the output environment—interactive or batch, scroll or page, or 80 or 132 characters wide, for example. The frame definition may contain head and foot lines and other specific features which facilitate the maintenance of uniformity within an application. The frame allows programs to process output according to the execution mode. Thus, reports being printed interactively will recognize the end of the screen and prompt to continue or quit. When execution is in the batch mode, the page size is adjusted and the prompt is ignored. Since this is managed by the frame, the same program specification and program generally are used for both interactive and batch reports.

Design is completed when the table schema and program and frame specifications are complete. Since TEMON uses the concept of minimal specifications, the translation from the design environment to the implementation and systems operations environment is a mechanical task which requires no knowledge beyond that already available in the design database. TEMON takes advantage of this fact by generating programs from the design database. Figure 5 illustrates the flow.

![Figure 5—Code generation tree](image-url)

The generation of programs is a function of program style. Since the TEMON specifications are implementation independent, the target language is arbitrary. For example, the flow presented for the PRompt command could be imple-
implemented in almost any language. The initial prototype of TEDIUM (called SIMPLE) was designed to produce COBOL outputs. The present version generates MUMPS using a custom coded generator. The next-generation system, currently under development, will have the generator written in TEDIUM and will be designed to produce MUMPS, COBOL, and other target languages. This will provide the tools for users to produce programs that conform to their individual target styles.

The final development process of the life cycle model is maintenance. TEDIUM supports the maintenance process by providing online and printed access to the entire design database. A query capability is available from all specification definition screens. It allows the user to identify programs, tables, frames, and elements, and to examine their contents and structure. Online access is also provided to cross references that indicate, for each program, what programs call and are called by it and what tables are read and written by it. Similar information is available by table name. Program structure listings present a call tree for a root program. All of these listings are derived from the design database and are always accurate and timely. Because the TEDIUM programs are written in TEDIUM, it is easy to add new functions to the tool base.

RESULTS

TEDIUM has been in production use since the summer of 1980. There have been 15 or 20 different people using the system. Since 1981 virtually all programming by a staff of eight has been done with TEDIUM. Four TEDIUM users are persons with no previous professional programming experience; after six months of experience, each was able to work independently with users in the design and implementation of applications. The obvious conclusion is that—even though documentation and training materials are severely deficient—the product is understandable at the level of specification and generation. Two other questions remain: (1) can TEDIUM be used to support the development of complex systems as its design objectives intend, and (2) do the features of TEDIUM improve productivity? Each is addressed below.

Scope of Systems Generated

TEDIUM is being used to implement applications that are put into production use. Five major applications have been implemented.

- TEDIUM.—All of TEDIUM except the program generator and selected utility programs are written in TEDIUM. Work on this version of TEDIUM was frozen in September 1981.
- SOCIAL WORK.—A small system used for management by objective, it was the first complete system to be implemented. It has been frozen since June 1981.
- CORE RECORD.—A prototype ambulatory care system was implemented in three clinics, accounting for 75,000 visits a year. This version of the system was frozen in December 1981.
- ONCOLOGY CLINICAL INFORMATION SYSTEM (OCIS).—A major tertiary care system with protocol directed patient management, this has been reprogrammed using TEDIUM. Work on system expansion continues.
- OPERATING ROOM SCHEDULING.—This subsystem for the Department of Anesthesiology is in production use; work continues on it.

Table I summarizes the approximate size of each application. It can be seen from these brief descriptions that TEDIUM can be used to produce complex systems.

<table>
<thead>
<tr>
<th>Application</th>
<th>Operational Date</th>
<th>Tables</th>
<th>Elements</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEDIUM</td>
<td>8/80</td>
<td>105</td>
<td>175</td>
<td>318</td>
</tr>
<tr>
<td>SOCIAL WORK</td>
<td>3/81</td>
<td>21</td>
<td>78</td>
<td>94</td>
</tr>
<tr>
<td>CORE RECORD</td>
<td>3/81</td>
<td>85</td>
<td>245</td>
<td>533</td>
</tr>
<tr>
<td>OCIS</td>
<td>6/81</td>
<td>456</td>
<td>1,251</td>
<td>2,177</td>
</tr>
<tr>
<td>ANESTHESIOLOGY</td>
<td>1/82</td>
<td>20</td>
<td>66</td>
<td>166</td>
</tr>
</tbody>
</table>

Productivity

Productivity is generally measured in terms of lines of code per unit of work. This measure is imperfect because there are no uniform definitions for a line of code. Moreover, the measure includes only an indication of the bulk of the delivered product. Lines-of-code measures do not address utility of the delivered product (Gladden states that 37% of the systems started were delivered but not used), or the impact of maintenance (generally considered to be at least half of the total life cycle cost.) Software science provides improved measures for volume and effort but does not address the other two variables.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Effort</th>
<th>Lines per man-day</th>
<th>Equivalent Custom Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start-up</td>
<td>10 man-years</td>
<td>13.8</td>
<td>101</td>
</tr>
<tr>
<td>Anesthesiology</td>
<td>10 man-days</td>
<td>35.3</td>
<td>247</td>
</tr>
<tr>
<td>Symposium</td>
<td>10 man-days</td>
<td>45.5</td>
<td>402</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Activity</th>
<th>Effort</th>
<th>Generated MUMPS</th>
<th>Generated COBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start-up</td>
<td>10 man-years</td>
<td>13.8</td>
<td>101</td>
</tr>
<tr>
<td>Anesthesiology</td>
<td>10 man-days</td>
<td>35.3</td>
<td>247</td>
</tr>
<tr>
<td>Symposium</td>
<td>10 man-days</td>
<td>45.5</td>
<td>402</td>
</tr>
</tbody>
</table>

a Estimated lines of custom code in the target language necessary to produce the same functionality. Based upon analysis by the author.

b The figures on COBOL are of questionable accuracy but do indicate rough orders of magnitude.

c Generated lines are based upon 30 characters per line of MUMPS code.

d All programmer activity from the time TEDIUM was first available to the end of 1981. Includes training of new employees, debugging of TEDIUM, etc.

e Prototype of the operating room scheduling system. First use of TEDIUM by the analyst.

f A symposium program management system developed by the author and a novice.
Table II summarizes three exercises in measuring TEDIUM productivity. More detailed analysis will be initiated when the major system we are working on is frozen. Each system cited is in operation and fully satisfies the users' needs (to the extent that this can be done within resource constraints). Finally, long-term maintenance will be facilitated because

- The size of the specifications is one-seventh the size of the generated programs; thus, there is less to maintain.
- Programs are generated from specifications and the data model definitions, so documentation is always correct and up-to-date.
- TEDIUM's automatic generation of cross-reference indices and set/used tables facilitates maintenance.
- The linkage between programs and user documentation is part of the TEDIUM program.

**DISCUSSION**

There are very few systematic attempts to address the problems of managing the software life cycle. Most tool development is directed to standalone, nonintegratable packages. Most of the work on specifications ignores the maintenance of a bidirectional link between the design and the delivered product. The computer generally is used as a passive store for design information without any provision of access to the knowledge imbedded in it. Too much of the design and coding effort is spent on the mechanical and repetitive translation of general, predictable requirements into programs—a process that is time-consuming and imperfectly executed. Clearly, there is a need to establish a conceptual framework for the development of automated tools that can manage this activity.

This paper began by outlining a fully automated paradigm for the development life cycle. It continued by illustrating how—for a specialized application area—a prototype environment is approaching this problem. It also suggested that the use of this tool offered an order of magnitude of improvement in productivity, a necessary condition for a fourth-generation development environment. While this specific product is of significant use to a small user community, the key issue is how the concept of a fully automated paradigm will affect the process of software development.

First, we must question whether products such as TEDIUM are transportable. Because TEDIUM represents such a significant change to the established life cycle model and product development flow, there is bound to be strong resistance. The barriers are the existence of program libraries and databases, investments in programmer/analyst training, emotional distrust, competition with proprietary products, variation in local styles and needs, and the presence of backlogs that preclude reassignments from production tasks to training in new methodologies.

The technical barriers are relatively simple to address. TEDIUM currently runs on a minicomputer within partitions of less than 10K. The system can be installed on a microprocessor-based system with four terminals, a printer, and 30M disk for about $20,000. This analyst station could support all development and rapid prototyping and send the generated code to a target machine for compilation and final testing. With the program generator written in TEDIUM, system programmers could adapt the system style to meet local needs or to interface with tools such as a DBMS, distributed processing software, and so on. Thus, if the social and emotional barriers can be overcome, the product is transportable. The growth in popularity of "programmerless" systems demonstrates how flexible the user community is.

Another question of equal importance relates to the marketing of such fundamentally different products. In his review of information systems in the 1980s, Weil notes that, because of a need for stability and support, the number of independent software houses will be limited to a dozen or two out of the universe of hundreds. This implies that, barring a phenomenal success story, leadership in the distribution of fully automated environments will have to come from the established vendors. And each of these already has products that claim to provide fourth-generation functionality.

In conclusion, therefore, we see that there is a user community need for continuity and support and an established vendor need to build on existing product lines. Both these needs reinforce the status quo—hence the resilience of FORTRAN and COBOL. Thus, a fully automated paradigm may find wide acceptance only with fifth-generation computer systems. In the meantime, however, TEDIUM provides an effective tool for high application risk IMS projects. And though the goals of TEDIUM development are limited, it is hoped that our experience will encourage others to consider alternative methodologies for a fourth-generation design environment.

**REFERENCES**


