Challenges and requirements for new application generators

by ALFONSO F. CARDENAS
University of California, Los Angeles
Los Angeles, California
and
WILLIAM P. GRAFTON
Continental Airlines
Los Angeles, California

ABSTRACT

The need for application generation as an alternative to the aging programming languages (COBOL, FORTRAN, PL/1, Pascal, etc.) is put forth. Major technologies that participate in the movement toward application generation are reviewed. A variety of technical problems that plague these individual technologies are identified. The problem of lack of standards in the participating technologies is discussed. A satisfactory application generator (AG) should synthesize the overwhelming variety of services and individual technologies into a cohesive whole. The desired characteristics of an AG are indicated. Last, the resistance of programmers, analysts, and other specialists to embracing a new level of application development beyond handcoding in programming languages is indicated.

INTRODUCTION

The last fifteen years have seen a gradual introduction of new techniques in application development. High-level procedural languages have largely replaced assembler languages for application code. Complex operating systems have assumed most of the computer system management tasks, freeing application programmers to concentrate on solving business problems. Standard sort programs, subroutine packages, utility programs, access methods, and such have solved common, recurring problems once and for all.

Generalized file-management systems have simplified report programming and data manipulation for simple to moderately complex applications. Generalized database/data-communication systems have made it feasible to put a whole enterprise on line. Data-dictionary/directory systems have offered the potential for storing the definitions, relationships, and usage of all data in an organization. Database modeling technology permits an organization to be described in terms of its need for and use of information.

Software development practices have emerged under the banner of software engineering. Structured programming, modular programming, top-down design, and so on are some of the development methodologies that have been advocated as enhancing software development productivity. However, the actual productivity enhancement realized has rarely been in terms of orders of magnitude.

The Basic Approach Remains Unchanged

These innovations, while encouraging, are merely refinements of a basic approach to application development that has remained unchanged since the early days of data processing.

We still analyze requirements, conduct a feasibility study, identify and evaluate alternatives, select a solution, prepare cost and schedule estimates, obtain budgetary and staffing approval, design the external and internal system specifications, write and check out detail procedural code, publish program and system documentation, conduct exhaustive testing, train the user and operators to run the system, and finally implement. By this time the environment has often changed, the schedule has slipped, the budget has been exceeded, the user is unhappy, the DP staff has turned over, and we are ready to begin again.[1,2]

The Basic Approach Is Unsatisfactory

A strong case can be made for the proposition that this approach is unsatisfactory in several important ways:

1. *It is not responsive.* The process just described generally requires months or years to produce a usable application system. The time required from the recognition of a user requirement until a computer-based solution is available is perceived as excessive and unacceptable by many users, and the data-processing organization is universally regarded as being unresponsive to user requirements.
2. *It handles changes poorly.* The basic application-development process is inherently weak in its ability to handle changes in design. The further along in the development cycle an application project progresses, the greater the impact of a change. Environmental changes, misunderstanding of problems, new or improved ideas, and errors in communication will inevitably lead to the need for changes; the result is rework, schedule slippages, cost overruns, and frustration. The response of data-processing organizations has been to lengthen and formalize the front end phases of the process in an attempt to obtain specifications from the user that amount to a signed contract.[3] While this may help in avoiding unnecessary changes and may assist in fixing responsibility for changes that do occur, it does not address the underlying problem.
3. *It results in a backlog of undeveloped systems.* The process is so labor intensive, and the skills involved are so scarce and so expensive, that most organizations simply cannot put together an application-development staff of the quality and quantity needed to meet user requirements. This results in a backlog of undeveloped applications, failure to exploit the potential of the computing system, and a general dissatisfaction with data processing.
4. *It produces systems that are difficult to maintain.* The difficulty of change noted above does not only impact development; it follows a conventionally developed system throughout its life. Maintaining production applications occupies a major part of a data-processing organization's time. It is often estimated as requiring 70 to 80 percent of programming resources. The culprit seems to be the procedural coding technique, which requires human definition, in extreme detail, of exactly how the application is to be processed. Designing ease of maintenance into a procedurally coded system requires time, talent, and foresight, and these resources are usually in short supply in a development project.

A New Approach Is Necessary

What is needed is a new capability that will overcome the problems and break the logjam in application development and maintenance. Application systems should be produced directly from their specifications without the need for pro-

cedural coding and lengthy testing. This capability is of course the subject under discussion—the application generator (AG). The call for automatic production of applications directly from their specification is not new. Various authors and efforts in the past have worked toward this.[4, 5, 6, 7, 8, 9, 10]

CURRENT APPLICATION-GENERATION TECHNOLOGY

Under the broad category of application-generation technology we can identify the following major technologies and emerging product lines.

Turnkey Applications Software

A variety of proprietary software packages are available, usually from software houses (for example, most of the bread and butter business applications such as payroll, order entry, inventory control, and invoicing).

Turnkey applications usually offer only a limited range of customizing options in the final application. The range of features is very fixed and is mostly limited to cosmetic changes (e.g., names of fields in files but no changes in the structure of the files). Significant changes required in the application necessitate hand coding via "user exits" or recoding of the code "generated." Thus, turnkey software cannot be classified as a full-fledged application-generation technology.

As a consequence, the prime clients for such software are organizations that, since they do not wish to develop their own special systems, are willing to adjust or even change significantly their usual systems or procedures to fit the turnkey software. A small company today can obtain a set of related turnkey applications for as little as $15,000 to $25,000.

Self-customizing Applications

Self-customizing applications or application customizers have the characteristic that the *structure* of the programs that make up the applications can be changed significantly to fit a customer. It is not a matter of just changing a few parameters or making rather cosmetic changes that may be required by a client. It is the capability of allowing significant structural changes that is required. This is a fundamental capability of self-customizing applications.

The changes are to be done automatically by the customizer without programmer intervention at the programming-language level. A few customizers have been reported in the literature: for example, IBM's Application Customizer Service (ACS) for System /3 environments,[11] Distribution System Simulator,[12] and "programming-by-questionnaire" efforts.[6]

Most special-purpose languages fall in the category of self-customizing applications. For example, there are GPSS and SIMSCRIPT for discrete system simulation and CSSL and CSMP for solving ordinary differential equations.

Generalized File-Management Systems (GFMS's) and Sophisticated Report Writers

This technology excels in report writing, sorting/merging, and related tasks, usually against existing files and databases. In contrast to turnkey and application-customizing technologies, this technology is not application specific, that is, it can apply to completely different application areas, such as sales/marketing analysis and real-time spacecraft data analysis. The GFMS adapts itself to the application by means of detailed data definitions and high-level, nonprocedural instructions provided by the user at execution time (e.g., "select all invoices greater than 6 weeks past due"). Prime examples of such technology are Informatics' Mark IV, ASI's ASI-ST, and Cullinane's Culprit. Experience has shown that GFMS's provide productivity gains over using conventional programming languages (COBOL, PL/1, FORTRAN, etc.) of on the average 8 to 1 in a significant percentage of the bread and butter applications of an organization. By productivity is meant the elapsed time and person/days required to do the job. GFMS's are mostly used by medium to large organizations with a major investment in EDP. There are probably from 3000 to 4000 installed GFMS's.

Query Language Processors

This technology has emerged with generalized database management systems (GDBMS's) to provide high-level and intelligent data retrieval and update (write, modify, delete) capabilities over a database. Query languages are transaction/record oriented as opposed to GFMS's, which are batch report/full file oriented. They are particularly effective in online, interactive applications not requiring the data volumes or complex report writing, sorting/merging, and so on where GFMS's excel. Without a query langauge, a nonsimple request for data from a database would require a number of statements in the programming language and in the procedural data manipulation language of the GDBMS (e.g., DL/1 in IMS, DML in CODASYL GDBMS). The more intelligent query processors, and in particular those furnished with the newer relational GDBMS's such as IBM's QBE[13] and SQL/DS,[14] automatically generate the commands that would otherwise have to be hand-coded by a programmer-user to obtain the answer. An example of a nonsimple data request that illustrates this point is the often-quoted "list the names of all employees whose salary is more than that of their managers, and list also the corresponding manager names."

Application Development Systems (ADS's)

This is a nebulous area of more recent activity than the previous technologies. ADS's are attempting to automate some of the tasks involved in developing more advanced applications involving online, screen-dialogue, and database environments. Specific examples are the American Management Systems Generation Five (possibly),[13] IBM's Application Development Facility,[16] Cullinane's Application Development System,[17] and Informatics' Mark V.[18] These systems provide a variety of tools that have shown significant gains in programmer productivity in developing applications for a database-communications environment.[19,20] Libraries of precanned code for frequently used database I/O, screen formatting, auditing, and so on can be invoked by the application developer, thus reducing total programming time and effort.

Thus far, these new systems have been developed for use with specific GDBMS and data-communications facilities.

Other Generators

Data-dictionary/directory systems (DDS's) have emerged recently as central tools in a database environment. Descriptions of data residing in databases, of application programs, of users involved, of the terminal network, and so on are concentrated in the DDS. The DDS can then generate the record descriptions and file descriptions (FD's) for application programs, as well as schema and subschema descriptions in the language of the particular GDBMS to which the DDS interfaces.

A number of other less encompassing code-generation aids that do not fit well in the previous categories have been developed and are in some use.[8] Two of these are

1. Decision table packages that can generate procedural code based on the logic defined in a decision table
2. COBOL aids and COBOL precompilers such as ADR's METACOBOL

However, their program-generation capability is usually limited to only narrow portions of the complete application required.

At the R&D and futuristic end of the spectrum lie efforts on automatic programming conducted by the artificial intelligence community. However, such exciting possibilities continue in the research stages, are highly speculative and still long range, and remain to be proven in practical commercial situations.

PROBLEMS OF CURRENT APPLICATION GENERATION TECHNOLOGY

A number of problems are observed in much of the current application development software.

1. The large majority of turnkey applications software and self-customizing applications do not use database management systems. Neither the generators themselves nor the applications that they generate enjoy the benefits of GDBMS, such as data independence, relatability between files, access flexibility through query languages, and performance and efficiency, etc. This is rather surprising, since a number of the flexibilities provided by a GDBMS would tend to enhance the range of services and customizing provided by the turnkey or self-customizing applications software. A major reason for the current situation is that conversion of software to a database environment usually requires reprogramming of the software. In addition, if the vendor wishes to penetrate the full database market, it must either provide a version for all of the popular GDBMS's (e.g., IMS, TOTAL, IDMS, ADABAS) or utilize a generalized GDBMS interface with a translation bridge for each GDBMS supported, accepting some loss of function and performance as a result. This is a very expensive process that most vendors of such software have not been willing to undertake or able to justify. Nevertheless, it is expected that the bulk of such software will eventually evolve toward databases, just as the bulk of hand-coded applications software is evolving (actually, being largely reprogrammed) toward databases.

   User companies that have entered the world of databases will very likely find it undesirable to make use of application generators if the code generated is not database oriented. Only in the case that the applications generated have no connectivity to other database applications and do not use or interact with data from existing databases will non-database application generators be attractive. However, there may be various applications that can have such isolation, so non-database application generators can not be automatically ruled out.
2. The few turnkey applications software systems and self-customizing applications that do use or interface with GDBMS's usually select the one or two most widely used GDBMS's to work with, namely, IBM's IMS and Cincom's TOTAL. The CODASYL standard is followed by a significant number of GDBMS's but not by IMS or TOTAL. Applications software targeted at non-CODASYL systems is thus tightly cemented to a specific GDBMS and its vendor. Vendors of turnkey software and application generators face the same question that everybody else faces: which GDBMS should be used? For the vendor it makes sense to interface with the GDBMS in widest use.

   The same can be said of GFMS's regarding their use of database technology. They usually provide interfaces to work only with databases under IMS and TOTAL. There are only a few exceptions at this stage of evolution, significant ones being Cullinane's Culprit and IDMS (a CODASYL GDBMS).
3. Applications are evolving toward not only database but also data-communication environments. Users are increasingly demanding online terminal access and remote access through communication lines. This leads toward the need for generalized software for data communication, for example, IBM's CICS and Cincom's ENVIRON. Such generalized data-communication systems (GDCS's) are unfortunately incompatible with one another because no standard exists. Thus, any choice by a vendor of the turnkey software or of a customized application generator essentially cements the software to the particular data-communication system chosen. There is a trend to choose IBM's CICS because it is the most commonly used of the data-communication systems.
4. Turnkey software and application generators may claim to have built in their own database and data-communications (DB/DC) facilities. This means that part of the software generated for a client includes the DB/DC facilities. It is very doubtful that these generated DB/DC facilities match up to the power of the acknowledged generalized DB/DC systems. However, this approach may (a) enable the generated software to be independent of the variety of full GDBMS' and GDCS's in the market place, and (b) enjoy custom DB/DC horsepower. But what if, as most likely is the case with medium to large

organizations, the application software generated is to use a shared database under a GDBMS and not be isolated from other hand-coded applications using the database? Now we have an interface problem to solve, and it is not a minor one.

5. There is no standard for report writers and the more sophisticated GFMS's. Consequently, all GFMS's differ in syntax, semantics, and features (although functionally they pursue the same goals). The architect of an application generator faces the same problem that the architect of an application faces: which GFMS to use. Once one is chosen, the generator or application is cemented to it. To no longer use the specific GFMS would mean reprogramming for another GFMS, or reprogramming by hand-coding in a conventional programming language, which might cost much more than reprogramming for another GFMS.

   It is observed that many millions of dollars invested in hand-coded applications and also in turnkey software and application customizers could have been saved by the use of report writers and, particularly, sophisticated GFMS's. What applications do not involve report writing, sorting/merging, and so on? We seem to be reinventing the wheel all the time by hand-coding these tasks. The lack of a standard for generalized report writers and GFMS's has contributed much to the problem.
6. There is no standard for query languages. Even the CODASYL standard followed by a significant number of GDBMS's does not specify a query language. Practically all GDBMS vendors now market a query language with their GDBMS's. Unfortunately, all of the query languages are incompatible with one another, even among CODASYL GDBMS's. It would make much sense to be able to use the same high-level, nonprocedural query language, or maybe two or three at the very most, to access data from databases with little or no concern for the particular GDBMS managing the database. Unfortunately, in the 1970's we fell into the practice of developing essentially n query languages for n GDBMS's, rather than 1, 2, or 3 query languages for most of the GDBMS's.

   The trend toward relational query languages and GDBMS's may be a way out of the dilemma. Unfortunately, no standard for the relational architecture has been developed yet. Consequently, it is to be expected that incompatibility will predominate even among the new relational GDBMS's or relational query languages that may be developed on top of enhanced existing GDBMS's.

DESIRED CHARACTERISTICS OF AN APPLICATION GENERATOR

1. *It should be a system.* As indicated in previous sections, we have a significant and rather overwhelming variety of tools that go beyond the aging programming languages. Worse still, the standards for such tools are few. As a result, questions of overlap, incompatibility, what complements what, what interfaces are needed from whom, and so on make the task of the individual application architect most difficult.

   A satisfactory AG should synthesize all of the available services and productivity tools into a cohesive whole. It should be the driving force that ties the development process together. It is thus a system, not merely an application package.
2. *It should be databased.* A reasonable requirement for generated applications is that they be databased systems. If an organization's data and relationships are predefined and the data may be accessed by a GDBMS, then the problem of generating systems to manipulate the data is greatly simplified. If an enterprise intends to use an AG, it should also do some sort of information modeling, design major subject databases, install a GDBMS, and develop detailed "families" of operational databases in parallel with the implementation and use of the AG. A data-dictionary/directory is also indicated, either as an integral part of the GDBMS, as a part of the AG, or as a separate package.

   Subject databases are beginning to attract attention, and products are beginning to emerge. An example is Cullinane's Integrated Manufacturing System (CIMS), which embodies the structures of a manufacturing database.[21] Subject databases should be a good start and a more attractive alternative than starting from scratch for a significant portion of the market.
3. *It should interface with GDBMS.* The AG should use the GDBMS but should not be dependent on any specific GDBMS product. This means that the AG developer will have to build the system to be "GDBMS independent." An interface module will have to be supplied to bridge between the AG and the more popular GDBMS's such as IMS, IDMS, and TOTAL. Interface module specifications should be made available so that software vendors, user groups, and so on may develop interfaces with other GDBMS products. An interesting possibility in this direction is the Informatics' TAPS interface package,[22] offered to software vendors as a means of making their products DB/DC independent.

   Unfortunately, current application-generation software packages that embrace databases seem to be tied either to a specific GDBMS or to their own database facilities.
4. *It should emphasize terminal-based applications.* The AG should recognize that more and more applications will be on line and terminal based, and its architecture should emphasize this type of development. Fortunately, report-based batch type applications can be viewed as essentially a subset of the more complex terminal-based transaction type applications, and a system that has been designed to produce terminal-based applications can be extended to produce report-based applications with minor extra effort on the part of the system vendor. Specifically, the following facilities can be envisioned.
   a. Screen-format designer aid. Terminal-based application generation should begin with a user-oriented definition of the input and output terminal dialogs (i.e., screen formats) involved. The AG should include a screen-format design aid in its architecture.

This facility would permit the application designer to enter on line to the AG the exact formats desired, including data names, screen placement, and attribute characteristics. The AG would in turn obtain data characteristics from the data dictionary and/or database schema, perform editing, error checking, and standards verification, and return a "picture" of the desired formats to the designer, repeating the process interactively until the design is satisfactory. The AG would then create correct input data for whatever screen-format generation process is used by the DC system involved (e.g., Message Format Service in IMS/DC, Basic Mapping Support in CICS).

b. Batch application I/O descriptions using GFMS. The input/output specifications for batch applications should be developed by the AG using analogous techniques. A GFMS facility should be part of the AG, and it should use input transaction record descriptions and output report formats in the same manner as input and output screen descriptions.

5. *It should include data-flow fallout from user I/O specifications.* The information developed in the input/output specifications should be used directly by the AG as the foundation of the input-process-output data-flow specifications of the application. Data requirements that cannot be inferred from the I/O specifications should be added by the application designer in terms of what data are needed. Given the data-element names the AG should be able to determine the data format, where it resides, and how to access it. It should be noted that all information supplied by the application designer is machine readable and nonprocedural.
6. *It should include nonprocedural data manipulation.* Data-manipulation requirements should be defined to the AG through a menu or decision table. The associated data-manipulation code should be generated by the AG from a library of "canned" modules. Perhaps the AG should include a standard global set of such routines, with extra-cost options being available for various industry groups. The AG standards and interfaces should encourage easy library expansion of data manipulation capability by in-house development, user groups, software houses, and the product vendor. User exits should also be provided for custom-coded procedural language modules as required.

PROTOTYPE APPLICATION DEVELOPMENT SUPPORT

One of the greatest potential benefits of AG's is in their ability to reduce the impact of change on the application-development process.

Heuristic Development

It is unarguably true that users often do not know precisely what they need, that system developers often design a less than optimal solution, and that the application environment often changes during or soon after development. It is also true that great benefits would often accrue if an application could be implemented quickly, albeit inefficiently, and optimized later. It would thus be desirable to develop applications on an iterative heuristic basis if feasible. The AG offers the potential to do this.

"Prototype" Development

AG architecture should include heavy support for "prototype" application development for "quick and dirty" implementation, followed by migration later to more efficient implementation, where indicated. Features of such support might include:

1. DB space "for rent" in standard formats
2. Interpretive processing
3. Global screen formats
4. Application invocation of query facility
5. Effective monitoring and reporting of resource utilization, so as to highlight hogs and assist cleanup

Prototyping Scenario

In a typical scenario, the user and the application designer would discuss requirements and specifications. They would then use a terminal to define data elements, relationships, and space requirements in a temporary generalized database that was already in place. Next they would customize a few global screen formats to meet application needs. This would be followed by definition of data-manipulation instructions to an interpretive processor and the query facility. Preparation and input of data would follow. The application would now be ready to check out and use. Elapsed time would vary from a few hours to a few days, depending upon complexity. Probably response time would be slow for high-volume production, and resource consumption would be relatively high. But the user would be up and running and happy. He/she could easily make changes or do the whole system over if necessary. When the user was satisfied, the system could be regenerated on a permanent basis, perhaps as part of an integrated system and/or database plan.

IMPROVED APPLICATION MAINTENANCE

Another major benefit of the AG is in the area of application maintenance. Since there is little or no procedural code involved, application changes as a result of specification modifications should be greatly simplified. The AG vendor should design maintenance aids into the system, including automated documentation support, change-control and audit-trail facilities, and data-dictionary interface.

HUMAN BARRIERS AGAINST NEW SOFTWARE PRODUCTION TECHNOLOGIES

The introduction of new software-production technologies faces various challenges. One major challenge is the resistance of programmers, analysts, and other specialists whose

activities are directly affected by such technologies. Years ago one frequently heard the words "assembly programmers die hard" as we were leaving behind assembly programming and evolving toward higher-level languages such as COBOL and FORTRAN. Now we can perhaps observe that "COBOL programmers die even harder" as we are trying to leave behind programming in the conventional COBOL, FORTRAN, and such and evolve toward another level of software development.

One significant technology that is part of the wave toward replacing conventional programming languages is GFMS technology. GFMS's excel in report writing, sorting, and so on. As noted earlier, productivity gains of GFMS's over conventional programming languages have been publicized as averaging 8 to 1 in a significant percentage of the bread and butter applications of an organization. In spite of these productivity advantages, GFMS's are used in only a fraction of such applications. All too frequently one of the reasons why this is so, and sometimes the major reason, is the resistance of EDP staffs to abandoning conventional programming languages, or the inertia of continuing to do business as usual.

We should realize that many EDP staffs have been in existence for almost two decades by now. Bureaucracy has solidified in many EDP shops. Worse still, the "Peter Principle" is already present in a growing number of cases. EDP staffs used to be among the most dynamic and innovative groups in organizations, but are now at times exhibiting reactionary behavior like many established professional groups. There is the reality that in many shops EDP staff have invested 10, 15, or 20 years in becoming proficient and capable with conventional-language programming. To suddenly ask them to put aside the tools that they have learned so well over so many years is bound to raise problems in many cases. Resistance to change is a strong human tendency with which we must cope. It takes time to educate, change attitudes, and gain proficiency in significantly new ways of doing business. If it took several years for most assembly programmers to abandon their old tools in favor of conventional programming languages, it will take more years to abandon the latter in favor of a new generation of software production tools.

Such human resistance to change will have to be added to the technological difficulties of developing new application-generation technologies to replace the aging conventional programming languages.

Recent history is filled with examples of rapid public acceptance of profound changes in lifestyle, religious, social, and political attitudes, foods and consumer products, the arts, medical care, and other fundamental components of our culture. We should therefore be able to effect technological change in software-production techniques with equal success.

CONCLUSIONS

A number of important technologies participate in the wave toward application generation, away from the aging programming languages. The technologies range from

1. the inflexible turnkey packages, to
2. intelligent self-customizing packages, to
3. generalized file-management systems that can replace programming languages for batch I/O, report writing, and sorting/merging tasks, to
4. query processors and associated database-management systems (GDBMS's) that can replace programming languages for all I/O from databases, to
5. the newer application-development systems that integrate a number of tools to develop more quickly online, terminal-oriented applications in a database/data-communications environment.

Unfortunately, none of them individually fulfills the role of a complete AG. Besides constituting an overwhelming variety, these technologies exhibit various limitations and suffer from the lack of standards. Complementing one technology with another becomes a difficult interfacing problem.

There is the need for AG's that synthesize the variety of services and individual technologies into a cohesive whole. The AG should exhibit the following characteristics:

1. It should support a database environment.
2. It should interface with popular GDBMS's.
3. It should emphasize terminal-based applications.
4. It should include data-flow fallout from user I/O specifications.
5. It should include nonprocedural data manipulation.

The AG should include heavy support for "prototype" application development for "quick and dirty" implementation followed by later migration to more efficient implementation where indicated.

A major benefit of the AG should be in enhancing application maintenance by greatly simplifying the introduction of inevitable application changes.

Not all the problems and challenges of evolving toward future generators are of a technical nature. A major problem is the resistance to change of programmers, analysts, and other specialists toward embracing a new way of doing business, away from hand-coding in the traditional programming languages.

REFERENCES

1. Fisher, D. A., "DOD's Common Programming Language Effort," *Computer*, March 1978, pp. 25–33.
2. Brooks, F. P., *The Mythical Man-Month*. Reading: Addison-Wesley, 1975.
3. "Introduction to SDM/70 Systems Development Methodology." Atlantic Software Inc.
4. Tiechroew, D., and H. Sayani, "Automation of System Building," *DATAMATION*, August 1971, pp. 25–30.
5. "Problem Statement Language, User's Manual." IS-DOS Project, University of Michigan, Ann Arbor, Michigan.
6. Low, D. W., "Programming by Questionnaire: An Effective Way to Use Decision Tables," *Communications of the ACM* 10 (1973), 5, pp. 282–286.
7. Hammer, M. M., W. G. Howe, and I. Wladawski, "An Overview of a Business Definition System," in *Proceedings, ACM SIGPLAN, Symposium on Higher Level Languages*, Santa Monica, Calif., March 28–29, 1974.
8. Cardenas, A. F., "Technology for Automatic Generation of Application Programs—A Pragmatic View," *Management Information Systems Quarterly* 1 (1977), September, pp. 49–72.
9. Winograd, T., "Beyond Programming Languages." *Communications of the ACM* 22 (1979), 7, pp. 391–401.
10. Zollicker, M. L. (ed.), "Proceedings of a Conference on Application Development Systems," *Data Base* 11 (1980), pp. 1–20.

11. "Application Customizer Service, System /3 Application Description Manual." IBM Reference Manual GH 20–0628.
12. "Distribution System Simulator (DSS) for System 360/370." IBM Systems Guide LB-21-0980.
13. "QBE, Query-by-Example." IBM Reference Manual G320-6062.
14. "SQL Data System." IBM Reference Manual GH24-5013.
15. "Generation Five." American Management Systems Inc., Arlington, Va.
16. "IMS Application Development Facility, General Information Manual." IBM Manual GB 21–9869.
17. "Application Development System/OnLine." Cullinane Data Base Systems, Westwood, Mass.
18. "Mark V Concepts and Facilities Manual." Informatics, Inc., Canoga Park, California.
19. Holtz, D. H., "A Non-Procedural Language for On-Line Applications." *DATAMATION,* April 1979, pp. 167-176.
20. "Programming Aid Reduces Cost 1.5 Million Dollars." Information Processing, 1 (1982). Information Systems Group, National Accounts Division, IBM Corporation.
21. "Cullinane Integrated Manufacturing System, Summary Description." Cullinane Data Base Systems Manual TXCM-110-10,Westwood, Mass.
22. "Terminal Application Processing System, Concepts and Facilities." Informatics Inc., New York, New York.