Program generators and their effect on programmer productivity

by RICHARD L. ROTH

Information and Systems Research, Inc.
Coraopolis, Pennsylvania

ABSTRACT

This paper investigates the concepts and utilization of source-code program-generating systems within the business programming sector; and, using a “standard” system development framework as a guide, discusses the advantages and disadvantages of program generators to software organization and their personnel.
INTRODUCTION

The topic of software tools is becoming an increasingly popular one within the software community. Programmers' salaries are constantly increasing, in line with increasing demand for these technicians; and with the steady decline of hardware prices, the software industry is constantly seeking new ways to cut costs, increase productivity, and maintain the quality of the software produced.

Many different kinds of software tools have come into existence since the introduction of the digital computer. Of course, depending on the definition of the term software tool, the nature of these various tools may be less than obvious, or may even be taken for granted, in today's world of high-technology software.

For example, computer hardware provides access to magnetic storage such as memory, tapes, and disks. The operating systems software provides a simplified method of storing and retrieving data from these devices.

Computer hardware provides a set of machine-level instructions to guide its internal actions; higher-level languages and language compilers exist to provide a method of communicating algorithms with a vernacular consistent with the problem to be solved.

The use of single-user batched systems has been to a great degree superseded by interactive, multiple-user systems that provide the programmer with tools to ease the burden of creating and debugging programs. For example, these systems allow the use of interpretive languages, which bypass lengthy compilation procedures and allow immediate communication of problems and errors to the user during the entry of the program; on-line diagnostic aids have replaced unreadable memory dumps and allow the user to examine, step by step, the actions of the program during execution; on-line text editors have replaced the key-punch, allowing easy access to the source program for immediate correction and retesting of incorrect source code; and other utilities have been designed to simplify the task of communicating a problem-solving procedure to the machine that will eventually perform that procedure.

Another tool in use today is the concept of relational databases and database management systems.

Aside from the technical considerations, these types of tools provide an almost unrestricted access to data stored on an information processing system. Coupled with sophisticated query languages and report generators, a database management system can provide an unsophisticated user with easy access to information while simultaneously providing a powerful tool to the software designer.

However, the design of most query languages and report generators requires the inclusion of routines for all possible requests that can be made. Obviously, the more routines that are included in the query language, the more powerful it becomes.

Unfortunately, users who would most benefit from the power of this type of system are, for the most part, excluded from using this tool. The user of a small- to medium-sized system is unable to support, by virtue of machine size and software cost, the overhead of a database management system. It is a fact that the needs of the users of the small systems are no different from those of the users of the larger computers; regrettably, the "scaled-down" versions of these tools cannot provide the type of access that most users need.

However, none of the tools mentioned really accommodate what is generally the largest, most time-consuming task. Problem-solving procedures must still be designed and hand-coded prior to the use of any of these other programming tools. As a result the programmer invariably spends less time thinking about the problem and more time thinking about communicating the problem to the computer by a method the computer will ultimately understand.

One of the most recent additions to the class of programmer tools has been the program generation system.

PROGRAM GENERATORS AND THE SYSTEM PRODUCTION PROCESS

A program generator is, very simply, a program that writes programs. Directed by series of parameters entered into the system by means of a user-oriented front end, the program generator actually creates a source code program, which can then be compiled and executed in the same fashion as an equivalent hand-written program. This is very different from table-driven application generators, or even program generators that produce machine level code directly, in that neither of these methods allows for the manual modification of generated code.

The introduction of a program generation system into the system development environment has a number of effects on the individual aspects of the process of system design and programming.

Using the typical system implementation process as a benchmark, one can examine how the introduction of a program-generating tool would affect each of the six segments of the procedure and simultaneously determine the effect of such a tool on programmer productivity.

The first step in the creation of a software system is the system design stage. Normally, during this stage, the analyst would discuss the needs of the systems end user, and attempt to translate these needs into terms to which the programmer can relate. During this stage the programmer or systems ana-
lyst must organize a vast amount of information concerning the structure of files, the interactions of fields within these files, and the operation of the actual programs.

The introduction of a program-generating system has a profound effect on this task of designing a software system. In order to use the program-generating tool fully, the analyst must be fully aware of the philosophy of the program-generating system.

The typical program-generating system creates programs within a very small class of all programs. It is necessary for the system designer to keep in mind while performing this analysis: that is, if this tool is to be used efficiently, the programs (or at least the functional procedures) designed during the specification stage must be as close as possible in internal operation and in form to the class of programs that can be generated. If the analyst strays too far from these prototype programs, the increase of productivity with the use of the program generator would be negated in proportion to the routines that would be "nongeneratable" and would therefore have to be created by conventional means.

It is the analyst's responsibility to use the features of the program-generating system within the application system design; in fact, the degree to which the program-generating system aids in the production of the programs is totally dependent upon the initial design of the system.

The existence of a program generator may also have ramifications in terms of the types of individuals who would actually perform the analysis. Because of the fact that the program generator would be addressing most of the computer programming issues that normally arise during an analysis, the individual actually performing this analysis should not necessarily have to be as sophisticated in terms of computer programming. This indicates that the individual performing the analysis could be more oriented to the application; in fact it may be that the end user could perform this analysis without the aid of a technician. As such, many of the communications problems that arise between nontechnical users and technical analysts would disappear, providing an analysis that would be more oriented toward the application area rather than the technical side of systems development. Finally, many of the difficulties that arise between the user and the designer would necessarily disappear, since they could very possibly be the same person.

The impact of a program-generating system can also affect the actual end result of the software systems design process. Most analyses yield a systems design communicated in the form of file layouts, including descriptions of fields, data types, field lengths, and allowed value ranges; data entry screen layouts, handwritten to show the position of fields and computer responses; report layouts showing the positions of fields on the printed page and totals and subtotals given on the report; flow charts and narrative flow descriptions, indicating the actions of individual programs and routines; and other information communicated by the printed page.

However, when considering the existence of a program-generating system, many of these written documents could be eliminated, thereby decreasing the amount of time the analyst would have to spend in preparing them, not to mention the clerical time and cost necessary to produce them. In fact, since the program-generating system would require the input of its parameters in a predetermined manner, the actual form of the end product of the system design could very well be the data input sheets for the program-generating system.

The second step in the system development process would be the actual programming, according to the specifications developed and accepted during the previous step. The role of the programmer during this phase of the project would also be dramatically altered by the introduction of a program-generating system. Assuming that during the previous step the systems analyst took pains to adhere to the philosophy of the program-generating system, and further assuming that the end result of the analysis was recorded on the data input sheets for the program-generating system, for the most part the programmer's task would become clerical. In fact, depending upon the user-friendly nature of the front end to the program-generating system, entering the results of the systems analysis into the program-generating system might not require the services of a programmer but rather those of a clerical worker. The effects on productivity here are obvious. Routines that are inherently simple and are repeated many times within the series of programs making up the application system would most probably be those created by the program-generating system. Programs such as formatted screen data entry programs, sorting programs, formatted report programs, and less complicated file update routines would most probably be targets for the program-generating system. In fact, depending upon the application system, it can be shown that programs that fall within these categories represent anywhere from 50% to 90% of most business-oriented systems in existence today. In this regard one can further extrapolate to say that programmer productivity would then increase by similar percentages on the typical project.

In any system there are programs that because of their application area or complexity of design fall outside the class of programs that could be generated. The programmer would necessarily have to produce these programs in a conventional manner. However, because the program-generating system has been used to write many of the more mundane programs, these generated programs actually set up guidelines for the internal structure of any other program that would have to be handwritten. For example, portions of source code that defines file structures, performs initialization functions, opens files, performs input/output functions, and performs other system-wide utility routines could be extracted from the generated programs. Even in the cases where programs would have to be written by hand, a major part of these programs could be borrowed from the source code generated in other, automatically created programs.

In terms of program text editing time alone, the savings would be quite significant. In terms of more important factors such as program correctness, debugging, and organizational source code programming conventions, the savings would be substantial.

The third of these six steps in the system development process would include the actual testing and debugging of the software system. Assuming again that the 50% to 90% figures are accurate for the average software system, it follows that the same percentages would hold for the number of programs which, by virtue of their being generated rather than hand-coded, could be assumed to be error-free (at least at the
programming level). One could then assume that the amount of time necessary for a programmer to test individual routines for desired functionality would be reduced by a comparable amount. The major portion of the testing and debugging process would only involve checking the interaction between program modules, checking any routines that had been hand-coded, and verifying the mechanically generated routines that for one reason or another might have been modified by the programmer after their generation.

The testing and debugging process is all too often skipped over in the interest of getting the system up and running. With the use of the program-generating system this technique may in fact become more acceptable as more and more of the programs can be considered bug-free at the point at which they are generated.

The fourth step in the system implementation process involves the creation of the software documentation, both from a technical and from an end user standpoint. The program-generating system can increase productivity in this critical area as well. From the fact that the programs that would be generated exist in a class of similar programs, it follows that the documentation describing the actions of these routines would also be fairly regular in their form. This fact holds true for technical-level as well as end-user-level documentation. The only variables within this documentation would be based on the actual parameters input during the initial program generation process, which could be preserved during their entry. The program-generating system would generate technical documentation in the form of file layouts, screen formats, report formats, and flow charts of the program’s activities, complete with machine-generated English-language narratives describing this flow. The clerical time required to produce this documentation would be reduced to the level needed to generate technical documentation for the hand-coded portions of the system, overall system philosophy, and other nongeneratable documentation.

In addition, because the activities of these routines are also defined within a narrow class, the user level documentation would be similarly generated. For example, English-language narratives describing the activity of a screen data entry program, complete with field-by-field descriptions of data to be entered, range checks and edit checks performed, and legal versus illegal values, could all be produced during the process of program generation. In this fashion pages for the users’ guide could be generated, and the clerical time needed to develop the complete users’ guide would be reduced to the task of developing users’ instructions for the hand-coded portions of the system.

The use of program generators also adds a new dimension to the fifth step, which is training the users of the software system. This area is vital to the success of any software installation. Many excellent techniques using the computer’s power for CAI (computer-assisted instruction) have been developed, but for the most part they have been ignored because of the time and expense necessary for incorporating these techniques into the design and programming of an application system. In addition, the added size and disk capacity needed to store this added information could become prohibitive on a small computer system. However, program-generating systems could be used to solve these problems. During program creation, dual sets of programs could be generated, of which one would be a normal application program for production use and the second would be a temporary program containing the additional code and textual information required for computer-assisted instruction. This secondary set of programs would be maintained in lieu of the first set during the initial training and installation and then removed from the system and replaced with the production programs once the operators were trained sufficiently. Again, the amount of personnel time required to do training at the user’s site would be significantly reduced by the existence of computer-assisted instructional programs, without the added burden of writing these programs and the permanent burden of maintaining these routines in a production environment. The programs that interface directly to the user (for example, screen data entry programs and query-level report programs) would be excellent candidates for computer-assisted instruction code created by a program-generating system.

The last of these six steps generally occurs sometime after the final installation of the software system. This step of course, is the modification and enhancement of the software system once it has been placed in production use. In many instances, and especially where a database management system has not been used, the addition of fields to data files and the addition of new features to application programs represents a major problem for the software systems organization. Individual programs must be individually modified to acknowledge the existence of new data and functions, documentation must be updated, and in some cases users must be retrained. Even in instances where COBOL-like libraries have been used throughout a system of programs, the procedural sections of each individual program must still be modified in order to use this new data. In this situation the program generator becomes an invaluable tool. Unlike the COBOL-like libraries and other source code management techniques, the parameters entered into a program generator carry with them information concerning the actions of the programs, relationships between fields, and specific descriptions of individual fields within data files. In the event of post-installation modifications these parameters can be updated to include new fields and concepts and the program generator can be used to create updated programs, training materials, technical documentation, and user level documentation for the newly modified system. It is not difficult to foresee instances where it will be more cost effective to automatically recreate entire subsystems that have been previously handwritten, even when the updates to that system are not dramatic. In cases where the original software was automatically generated this type of update would be only an operational (rather than a technical) matter.

In summary, then, the existence of a program-generating system dramatically increases productivity in all the major areas of the system design and development process. Assuming that the system design is consistent with the philosophy of the program-generating system, the systems installation process—including the system design; the actual specifications; and the programming, testing, debugging, installation, and training—would all enjoy significant increases in productivity.

This new mode of systems development is based to a great
degree on the existence of a program-generating system containing features in all the areas of system development, including systems analysis, actual programming, technical documentation, user level documentation, computer-assisted instruction, and end user training. In addition, it assumes that the capability of this program-generating system extends into many subapplication areas, including screen data entry program generation, sort program generation, file update program generation, reports program generation, and the existence of a user-friendly front end to access the program-generating system. But even with a program generation system that does not contain all these features, it is evident that the role of computer professionals in light of these program-generating systems, and the activities that will be performed by these professionals, have been dramatically altered. In essence the program-generating system would cause a type of professional migration. Systems analysts would strive to become experts in one or more application areas, and programmers and program analysts would strive to become algorithmic engineers, spending less time thinking about explaining problems to the computer system and more time thinking about sophisticated solutions to the actual problems at hand.

ACCEPTANCE WITHIN THE PROFESSIONAL COMMUNITY

It may be, however, that some programmers are fearful of this attempt at automating their jobs; in fact, many become resentful. This is unfortunate for two reasons: First, the concept of program generation is truly in its infancy and requires the input of a majority of the data processing community to become a truly effective tool. Program generators must be created for a much wider range of application areas than they exist for today. And, of course, these program-generating systems are not today themselves generated; in that they are also programs, they are today handwritten. Second, it is usually these types of people who resist the professional advancement that program generators offer—who are either fearful of or unable to make these changes within their professional community.

It is conceivable that the introduction of a program-generating system into a software organization where these types of professionals exist would not cause a dramatic increase in productivity. The impact of the system would be lessened by individuals who resist the use of the product.

ADVANTAGES AND DISADVANTAGES

A program-generating system is not a panacea; and, depending on the method of its use within a particular environment, the program-generating system has several advantages and disadvantages to the organization using it. In terms of the organization, the existence of a program generator, properly used, would necessarily decrease the need for programmers and programming support individuals involved in the creation of systems software. For the most part, the low-level programming activities would be handled cost effectively by the program-generating system. In addition to the actual programming, the organizational overhead would also be reduced in terms of documentation, training, and technical support personnel. The need for systems design and analysis staff within the organization would probably not be reduced, but the skills that these personnel would have or acquire would be dramatically different from those of their counterparts in organizations not incorporating a program-generating system. Systems personnel would necessarily begin to acquire skills in application areas rather than systems programming areas to supplement the effect that a program-generating system would have on the organization. Although this is not a direct productivity gain, it does have positive ramifications for the software organization. It is apparent that the most successful software houses concentrate on a small number of application areas, and the quality of their products is in direct proportion to the amount of knowledge concerning the application (not systems) that the software house and its personnel command.

In examining the disadvantages of a program-generating system it is possible to draw several analogies between program-generating systems and the concept of structured programming. For either of these techniques to be used effectively within an organization, the organization must be organized to use the tool properly and effectively in the day-to-day business of creating programs. One of the most obvious disadvantages of a program-generating system is its inherent inability to generate all the programs required in a particular software system. Because no program-generating system can accomplish 100% of the required tasks, there are gaps in the program-generating process that must be filled by conventional programming techniques. If the organization is not structured in such a way as to monitor this activity, a long-term trend of increasing numbers of conventional programs and decreasing numbers of generated programs may occur. Obviously, then, this one disadvantage of program-generating systems would effectively negate all the aforementioned advantages.

Another disadvantage of program generation systems is that the source code is generated with a single, immutable style. Obviously a given program-generating system will generate programs with a single, consistent internal structure; and as such, this internal design will probably not be consistent with that which the organization has been producing previously. Again, the degree of acceptance of a program-generating system would be based on how well the organization could change its techniques and internal standards to match those of the program-generating system. Unlike a human programmer whose techniques may be modified, the program-generating system cannot be so easily retrained. For the same reasons as mentioned above, any roadblock in the path of acceptance of a program-generating system decreases its effectiveness within the organization. However, this disadvantage might also be considered an advantage in organizations having no internal standards for the generation of conventionally written programs.

A third disadvantage, along these same lines, has to do with the actual operation of the programs that the program-generating system would produce. In terms of highly visible, user-interactive programs such as screen data entry programs, and in terms of printed output such as that produced by report programs, the output of the automatically generated pro-
grams might not conform to the standards already accepted within an organization or might not match those formats created by canned packages used by the organization. This could lead to extensive modification of generated programs, abandoning the program-generating system as an in-house tool or massive updates of these canned routines. Obviously, any of these actions would have a severe effect on productivity gains.

It is interesting to note that all the disadvantages mentioned could be (for the most part) eliminated by either modifying standards to include the program-generating system or designing the program-generating system to match widely accepted standards. Similar to the structured programming techniques mentioned above, the organization’s ability to accept these techniques is directly proportional to the benefits gained by using them. Many programmers within an organization may resist the installation of new techniques or programming tools, stating reasons such as “We’ve always done it this way.” As Edward Yourdon states in his book, *How To Manage Structured Programming*, “It’s literally all they can do to write programs in the disorganized fashion to which they’ve been accustomed; to suggest they should introduce some organization, some structure into their work is literally beyond their ken.” (P. 172)¹

Unlike structured programming techniques, however, the program-generating system can, in effect, replace this type of individual within the organization instead of attempting to modify the individual’s behavior to fit some predetermined pattern.

OTHER USES OF PROGRAM GENERATORS

As previously discussed, program-generating systems must continue to grow, both in their capabilities and in the scope of generated programs, in order to be an effective tool for the future.

For example, program-generating systems should be expanded to cover more of the tasks involved before and after the actual generation of a program. Tools designed to simplify the task of systems analysis, including natural-language input of system parameters, would be a very desirable addition to the front end of a program-generating system. In fact, within a certain restricted set of application programs, it may be possible to design a table-driven decision tree system that would actually guide the unsophisticated user through a system analysis of the desired application. This system would simultaneously gather the parameters to be used by the program-generating system. Coupled with computer-assisted instruction and even a natural-language interface, this technique would provide for direct interface with the end user and eliminate the need for computer expertise during the design stage.

The program-generating system could be used as an instructional tool for teaching proper programming techniques within a classroom situation and as an instructional tool for new members of the programming staff of a software organization. Obviously, if the internal standards of the organization were closely related to the structure of the generated programs, an instructional program could be based on the program generation system, teaching new programmers the desired structure of conventionally written programs. In a classroom situation dealing with first-time programmers, the program-generating tool could be invaluable in instructing students in the use of proper programming techniques. The student would simply describe a desired program to the program generator and would be able to examine the final results. By imitating these programming techniques, the student programmer would learn these techniques by example.

Another natural migration path for a program-generating system would be the generation of programs in other languages, as well as the generation of programs for other manufacturers’ computers and operating systems, as an option during the generation process. For the sake of ease of migration from one manufacturer’s computer to another, and especially in the case of languages such as COBOL and BASIC, this migration would be a fairly trivial task when compared to the task of creating a new program generator for the new computer system. However, the task of translating from one language to another is not as simple.

For example, consider the case where a COBOL program is written to generate COBOL programs. Simply translating this COBOL program into another language, say BASIC, would result in a BASIC program, which would still generate COBOL code. It is therefore necessary to translate not only the program itself, but the class of programs that program would generate as well.

There are many ramifications to consider regarding the possibility of an easily transportable program-generating system. Unlike any other transportable feature of an operating system or language, the idea of a transportable program-generating system implies the existence of transportable programming techniques across manufacturer lines and the software organizations using the manufacturer’s computers. It is reasonable to assume that if a single program-generating system were to become popular in the industry, and further, that the program-generating system created source code in a transportable language, for a variety of popular computers, these conditions would define a standard of programming technique unparalleled in the industry today.

The existence of such a portable program-generating system also has some implications for the microcomputer side of the industry. By simply reading today’s microcomputer journals it is easy to see that the average cost of a software package, be it operating-system or application-level, is generally a small percentage of the hardware price. As hardware prices continue to drop, the prices asked for the software may drop to almost ridiculous levels. If one adheres to the concept of getting what one pays for, it becomes obvious that the quality of conventionally written customized programs existing on microcomputer systems would be less than desirable. However, the existence of program-generating systems on these microcomputer systems would be a natural way to increase the quality of the custom software products without increasing their price (beyond that which the typical microcomputer user would be willing to pay).

SUMMARY

It is obvious that, unlike any other tool, the program-generating system is today and will continue to be a major
factor in changing the face of software organizations. As these tools become more sophisticated in terms of their internal structure for the generation of programs, expand their scope in terms of the types of programs they will generate, and improve their interface to the end user via user-friendly natural-language front ends, the program generator will literally reorganize the whole structure of the design, installation, and implementation of computerized software systems. Many of the problems besieging our industry today, including lack of good-quality personnel, lack of widely accepted standards, problems with reputation and acceptance by the general public, and the rapidly decreasing price of hardware with respect to the labor-intensive costs of software production, are problems that can be solved today and in the future by the program-generating tool.

REFERENCES