The Cross System Product application generator: An evolution

by W. K. Haynes
M. E. Dewell
P. J. Herman

An application generator is a generalized application development tool with which professional programmers develop applications using a fourth-generation language. This paper describes the requirements that led to the Cross System Product application generator, and how the product progressed from a single-environment product to the current multienvironment product. Also described are how the Cross System Product fits within Systems Application Architecture and how that may affect the future of the Cross System Product.

In the 1970s, the growth in demand for interactive applications was at a rate never before seen in the industry. Technology had made interactive system hardware available at a price that justified the cost of interactive applications. Interactive systems presented major new challenges within a data processing industry that had already matured significantly.

Interactive system control programs, such as the Customer Information Control System (CICS) and the Time Sharing Option (TSO), that are used to support interactive applications were offered to end users for the first time. However, these system control programs still mapped the functions of the input/output of applications to specific devices, as had always been done in batch programming. It had not yet been recognized that providing an interface to a person through an interactive terminal was more complex than reading from or writing to a tape or disk. The system control program interface was not simplified to the point where a programmer could produce interactive applications on a productive basis. The demand for interactive applications continued to outstrip system control program productivity improvements.

The introduction of interactive programming tools increased the development productivity of interactive system control programs. But even with these introductions, productivity fell short of the interactive applications demand.

Experience with interactive system control programs showed that the majority of interactive application functions could be categorized into a set of generalized functions (e.g., read or write to a file or database, or display information on a screen). If each generalized function could be specified with only minor differences, the majority of interactive applications could be produced with very high productivity. This approach became known as an "application generator." Early generators had a built-in model of the general types of interactive application functions. Customization of the function and flow could be

accomplished to a small degree with little time and education. The length of time devoted to design activities for generated applications was shortened because the basic design architecture was fixed by the built-in generalized functions of the generator. The time for coding and debugging activities was also shortened because generated function was reused by the generator, rather than rewritten for each application by the application developer.

The early application generators were aimed at the challenge of the still-growing interactive application backlog. Some generators were introduced with great expectations, but their success was limited for the following two reasons:

- The generators were too rigid in their underlying architecture. The generators had a built-in concept, or model, of the application logic that could not be changed except for a few variations. Therefore, the applications developed using the generator had to be designed around the architecture of the generator.
- The architecture of the generator had to be understood completely by the designer in order to be well exploited. Although little or no education about the generator was required by the coder, the designer required significant education to understand the generator before design activities could begin. Because of overwhelming problems being faced by the data processing community, few were willing to take the time to re-educate their designers in the use of an unproven concept.

In 1978, Data Management System/Distributed Processing Programming Executive (DMS/DPPX), the forerunner of Cross System Product, was developed to fit a niche that had not yet been addressed by any other product. That niche was dealt with by designing the product to retain the advantages of an application generator and to remove the two major weaknesses cited above in the following ways:

- The rigid application structure of a generator was relaxed enough to allow changes, if desired. The framework provided could be utilized for maximum productivity, or changed by the designer. Less education was required because conventional designs could be used while the generator method was learned.
- The functions of the generator were available to the developer through a menu-driven specification interface which assisted the developer in implementing the design. The generator language contained both nonprocedural functions integrated with the application structure and procedural functions similar to the high-level programming languages with which all developers were familiar.

The generator gave the interactive application developer the flexibility of combining fixed application structures with minimal procedural logic to implement general end-user requirements, or combining fixed application structures with more complex procedural logic to implement more complex end-user requirements. In both cases, application development productivity was maximized.

Cross System Product

In order to significantly increase the productivity of the developer, application generators had to improve the programmer environment. Cross System Product did so by automating major time-consuming activities of interactive application programmers. Following are key programmer objectives for the product:

- Perform all development tasks on an interactive display terminal
- Check the validity of syntax and semantics at entry time
- Exploit reusable application modules
- Encourage documentation for all program definitions
- Test interactively, with trace
- Provide on-line help
- Use generic terminology
- Provide a dictionary-like source library
- Provide a modular application structure

Development methodology. The initial design of the Cross System Product encouraged modular programming, module reuse, function point design, single-process entry and exit, and structured programming. The basic premise was to have the developer think of his application in terms of data and the processes to be performed on those data. The definitions were to be high-level nonprocedural specifications, called *process options*, for data access and data editing. Procedural logic definition was provided to specify the conditions for and sequence of execution of the processes. The high-level process and data specification resulted in increased productivity and reduced skill-level requirements.

Comprehensive development environment. A prime goal was to provide a comprehensive development environment that addressed the application phases

of definition, test, generation, and maintenance as defined below.

- *Definition*—Definition included data, screen formats, report formats, and logic. A specific methodology for defining programs was built into the interactive program development facility. This methodology supported the design of applications that used many current state-of-the-art programming design techniques. Checking of interactive syntax and semantics at the time of source entry was provided, which improved application developer productivity by providing immediate feedback on inconsistencies within the application under development. An interactive screen design and definition facility (painter) with a WYSIWYG (what you see is what you get) approach was also provided.
- *Test*—Test included interactive testing and debugging of source code that did not require compilation or generation, and also included tracing of process, logic execution, and data modification. Trace output routing supported both terminal and file destinations.
- *Generation*—Application generation was performed after an application had been developed and tested. Application generation bound the generated form of an application to a specific system environment.
- *Maintenance*—The comprehensive development environment needed a common active dictionary or definition repository for the data, screen formats, report formats, and logic definitions. The initial release of the Cross System Product on the IBM 8100 System Distributed Processing Programming Executive (DPPX) used the DPPX Display Presentation Services for dialog management and library services. The library services provided a means for sharing Cross System Product application data and function definitions among developers.

Portability. An evolving goal is to make applications portable between environments. Portability had not been achieved by most high-level languages. COBOL, for example, was available in multiple environments, but COBOL source code was not considered sufficiently portable at the time that Cross System Product was initially released.

A significant number of IBM installations had a distributed data processing environment with hardware consisting of large, medium, and small systems. Within this environment there was a need to provide an application development tool that presented a common development methodology across the different hardware and operating system architectures. Portability was required to eliminate the need for redesigning or recoding the same application for different systems. Cross System Product provided that tool.

An evolving goal is to make applications portable between environments.

The initial release of the Cross System Product, developed for the IBM 8100 system, which was a new small distributed system at the time, was an ideal environment in which to introduce this new application development tool.

Because of the original design of the Cross System Product, its success on the 8100 system, and the lack of a similar product in the System/370 environment, Cross System Product was developed for Customer Information Control System/Virtual Storage (CICS/VS), then for Multiple Virtual Storage/Time Sharing Option (MVS/TSO) and Virtual Machine/System Product Conversational Monitor System (VM/SP CMS) in 1982. The Cross System Product was unique in its capability to move an application definition from system to system without change. Only functions that could be made compatible in all supported environments would be added to the generator. Exceptions would be made only for major strategic items. An example of an exception was the addition of support for IBM's hierarchical databases, Information Management System/Database (IMS/DB) and Disk Operating System/Data Language/One (DOS/DL/I).

Portability was accomplished by providing a generalized application specification language, interpretive execution, and a common interface to environment-unique system services. The two-part structure of the Cross System Product, Application Develop-

ment and Application Execution, allowed an application to be created on one system (where the application development functions were installed) and easily run on another system (where the application execution functions were installed). Application Development generated an application from an application source that was relatively independent of operating system and hardware considerations. When a generated application was used, Application Execution provided the system implementation and automatically adapted the execution to the specific production environment.

Common User Access. The user interface was designed to be system-independent. Therefore, as new environments were added, the user interface remained the same. Because the application specification language was the same regardless of environment, it was also completely portable. Common terminology was used for functions that existed on multiple systems but were provided by different facilities.

Cross System Product provided three ways to edit user-entered data.

The implementation of the common user interface required a screen display and dialog manager. Cross System Product implemented both of these functions by using the Display Presentation Services taken from the IBM 8100 system. Because of the use of a common interface to environment-unique system services, it required minimal effort to move, or "port" the Cross System Product screen display and dialog manager to new environments. It also provided consistency in implementation that could not have been achieved using existing facilities in target environments.

The application development facilities provided a menu-prompt interface for application specification, an on-line tutorial, and a help facility. Defaults were provided for most application specifications, such as database access that allowed the developer to use databases without having to be an expert.

Novice/expert interface. A customer review of the preliminary design for DL/I support indicated a need to support both the novice and the expert developer. The novice developer needed to be able to define DL/I database applications without having to understand DL/I. The experienced DL/I database application developer needed both the productivity advantages of the Cross System Product and the ability to utilize the full power of the DL/I database. The resulting design set the direction for the Cross System Product database support that followed. In that methodology,

1. The developer (or database administrator) defines a data structure.
2. The developer specifies a function (in Cross System Product terminology, a *process option*) to access the data structure.
3. A default database call statement model for the function is provided by the generator. (The defaults typically satisfy 80 percent of the database input/output requirements of the application.)
4. The developer views and (optionally) modifies the default database call. The inexperienced developer uses the defaults to learn how calls are defined. The experienced developer modifies the statement when specialized functions are required.

This methodology allows the structure of the database application to be under the control of the application programmer. The novice is productive more quickly, and the experienced developer manipulates the database in a way that is directly related to his previous experience with other high-level languages.

As stated earlier, most early application generator implementations were based on fixed application models. Limited procedural logic and editing, developed in a language external to the application generator, could be performed only at generator-identified user exits. The exit routines were mainly used for editing user-entered data and reporting errors back to the user.

It was important that the Cross System Product implementation be more flexible than its predecessor. In order to give the user greater flexibility, Cross System Product provided three ways to edit user-entered data:

- Nonprocedural specification of standard edits
- An exit capability that could be implemented using the Cross System Product procedural logic statements for each input variable
- A call-level interface, invokable anywhere

Cross System Product provided an application program call and transfer function to high-level language programs. The call was provided with a common syntax that was independent of the language of the called program. This interface allowed applications to exploit system functions not supported by the generator. In providing this interface, the Cross System Product application retained its portability and environment-independent characteristics.

The present version of the Cross System Product, Version 3, includes a number of enhancements that complement the strong base of Cross System Product functions. As can be seen in the following paragraphs, many of those enhancements tie the Cross System Product to direct exploitation of other Systems Application Architecture (SAA) components.

Relational database support. Relational database support allows Cross System Product applications to access relational databases via Structured Query Language (SQL). The support provides two powerful levels of interface:

- A novice interface for the untrained SQL user
- An expert interface for the trained SQL user

Cross System Product/Application Development generates default SQL statements, permitting the novice user to access the relational database with minimal knowledge of Structured Query Language/Data System (SQL/DS) or Database 2 (DB2). Another new process option allows the trained relational database developer to code multirow insert, delete, and update statements or data definition statements not directly supported by the common set of process options.

Source interface utility. The source interface utility support allows a user to import file definitions from alternative file sources. This capability relieves the customer of the error-prone redefinition of data. The alternative file sources supported include

- COBOL data structures, COBOL being one of the SAA languages
- DL/I Program Specification Block definitions, DL/I being an environment-unique product supported as a tower of function upon SAA

MVS/XA support. Support of Multiple Virtual Storage/Extended Architecture (MVS/XA) permits Cross System Product/Application Development and Cross System Product/Application Execution to exploit the 31-bit addressing mode (above the 16M-byte boundary) in MVS/XA systems. Virtual storage utilization in the large MVS/XA environment is enhanced.

Development library concatenation. This support permits each developer to access up to six Cross System Product/Application Development libraries. Developers can share information that is common for multiple applications, yet have their own individual library for development and testing of their own applications. Each developer may set the order in which the libraries should be searched for a development object.

The fourth-generation language environment has been driven by one main theme—*productivity*. Behind this theme there are numerous issues: How can software solutions be delivered to the end user more quickly, more efficiently, and with fewer development and maintenance costs? How can the ease of use be enhanced, thereby increasing end-user productivity? How can the latest technology be integrated into the application development process? In order to understand how these and other issues relate to the Cross System Product future, we will identify a number of technology areas that influence Cross System Product direction, followed by a description of how these technology areas may fit with the future of the Cross System Product.[1]

Computer-Aided Software Engineering (CASE) technology. The collection of integrated development tools that shorten the software development life cycle constitutes CASE technology. CASE technology parallels SAA concepts very closely, because both define end-user common programming interfaces that work together in an integrated programming environment. The Cross System Product already contains many of the SAA attributes, in that its development tools are built as separate components that are integrated into a unified development environment. Cross System Product should continue to build upon this architecture, so that when enhanced tools and enhanced environment-unique system functions are made available, they can easily be integrated into the Cross System Product environment.

Intelligent workstation. End-user productivity has increased with the introduction of the personal com-

puter. The reason for this increase is that host computing capabilities, often available only in the information systems organization, are now available to the end user. As the functional capabilities of the personal computer increase, particularly in new technology areas such as networking, distributed processing, and graphics, opportunities for greater end-user productivity may be realized with exploitation of these technologies. For example, Cross System Product could support implementation methodologies that allow end-user function and data to be distributed among host and intelligent workstation computers. The user should be able to determine the best location of his or her function and data, and what data should be transmitted between the host and intelligent workstation.

Application models. The development and use of application models is not new. For many years, developers have used models, often called skeletons, to develop new application functions. Cross System Product could automate the use of models by prompting the user for the function to be performed, and present the user with a potential list of application models that could be modified to the end user's new needs. When the new function is completed, it would then become an additional application model for future end-user application requirements.

Abstracted application specification. One of the characteristics of a procedural language is that the user must specify *how* an application performs a function. The end user's real objectives are *what* an application does. Greater development productivity is gained when the developer specifies application function without concern about the details of the implementation. This abstracted application specification also provides greater implementation independence.

Implementation independence has been a cornerstone of Cross System Product in areas such as screen definition, screen data verification, and database manipulation. Implementation independence could be enhanced with Cross System Product natural-language constructs that abstract procedural function, thereby reducing the amount and complexity of logic required to develop applications. For example, a statement such as "FOR EACH CUSTOMER WHOSE ORDER IS SHIPPED ..." could result in files containing CUSTOMER records being implicitly opened, records read and selected based on ORDER having a value of SHIPPED, and files closed without explicitly specifying these procedural operations in the user program.

Artificial intelligence (AI). The emergence of nonprocedural, rules-driven technologies has made possible a new set of user applications that were never before cost-effective when using traditional procedural programming functions. Cross System Product could integrate AI functions with its current set of procedural functions in order to provide the end user with an even greater set of potential implementation functions. The developer could then address an even greater spectrum of end-user applications by matching his implementation needs with the more logical implementation methodology, either a procedural one or a nonprocedural rules-driven one.

Concluding remarks

As described in this paper, Cross System Product has been a product of evolution. Because of the pioneering role of Cross System Product in developing Common User Access, Common Programming Interface, and consistent applications across architectures, all driven by the requirement for higher end-user productivity, Cross System Product now plays a key role as a leader and supporting product of Systems Application Architecture.

Acknowledgments

The authors sincerely appreciate the efforts of their knowledgeable peers in developing an understandable, consistent Cross System Product SAA paper. In particular, we would like to recognize the efforts of Hazel C. Bodner, Larry E. Clark, Pamela J. Clifton, Paul R. Hoffman, Louis J. Mamo, Jerry E. McCall, and Richard K. Runyan.

Note

1. Identification of technology areas, and how they might be incorporated in future releases of the Cross System Product, should not be considered to be a commitment to include such technologies in the Cross System Product.

William K. Haynes *IBM Programming Systems Development Laboratory, 11000 Regency Parkway, Cary, North Carolina 27511.* Mr. Haynes joined the IBM Field Engineering Division in 1968 as a customer engineer in Dallas, Texas. From 1971 to 1976, he served as a program support representative. In 1976, he transferred to the General Systems Division, Rochester, Minnesota, where he held various development, management, and staff positions for the development of the initial System/38 operating system. In 1981 he transferred to Information Systems/Software Development in Dallas, Texas, where he worked on a number of products, including the Patient Care System/Application Development Sys-

tem application generator. In 1985, he transferred to his present location to work on development solutions for intelligent workstations. Mr. Haynes has been working in the Application Generator Products group since 1986.

M. Eugene Dewell *IBM Programming Systems Development Laboratory, 11000 Regency Parkway, Cary, North Carolina 27511.* Mr. Dewell joined IBM in 1967 in Indianapolis, Indiana, as a systems engineer. After seven years in that capacity, he was transferred to software development at Research Triangle Park, North Carolina, where he was a programmer for the EXTM product. He was on the initial team that designed and developed the DMS/DPPX application generator product, which was the forerunner of today's Cross System Product application generator.

Paul J. Herman *IBM Programming Systems Development Laboratory, 11000 Regency Parkway, Cary, North Carolina 27511.* Mr. Herman joined IBM in 1965 and worked first in the Field Engineering Division for eleven years, and then as a systems engineer for four years. Starting his Cross System Product experience in 1980 with the forerunner product, DMS/DPCX, and continuing to today, he has held development, testing, customer support, and (most recently) performance-related roles. Cross System Product was chosen to be the primary application generator for the development of software for the 1988 Winter Olympics, during which Mr. Herman provided on-site support.

Reprint Order No. G321-5333.