Problems of application-system cost, control, and effectiveness can best be addressed by highly consistent development and execution environments. This paper examines some relevant new approaches (systems description languages, new data models, application generators, and very-high-level languages), discusses the need for additional integration, and outlines a particular integration direction. This direction is intended to illustrate both the kind of consolidation needed and some of the problems involved.

Towards an integrated development environment

by P. S. Newman

The history of general-purpose software can, in some sense, be seen as the provision of increasingly powerful remedies for the following persistent problems:

- *Development cost.* It costs so much to develop and modify application systems that many important functions cannot be implemented.
- *Effectiveness.* Application systems frequently do not serve the needs of their users particularly well.
- *Systems control.* It is rarely clear to the management whose responsibilities subsume the functions of an application system precisely what functions are to be performed.

Although general problems remain constant, the specifics change with changes in hardware/software contexts and with changes in user expectations. Thus every few years brings a new set of problem analyses and associated remedies, the latter generally consisting of new methodologies (e.g., structured programming) and/or new tools (e.g., data dictionaries).

Some of the more important contemporary developments in the area of methodologies and tools include system description languages, conceptual models of information, application generators, and the integration of data base manipulation into programming languages. Another important development is the identification of tool multiplicity itself as a significant cause of application problems. This

implies a need for efforts of "creative consolidation," resulting in definitions of comprehensive and manageable application development/execution environments.

This paper develops an outline of one such environment, with particular emphasis on the development aspect. The sections under the next three main headings are concerned with that aspect exclusively. They identify objectives, review traditional environments and new approaches with respect to those objectives, and, finally, sketch an environment obtained by adapting and integrating the new approaches. The last main section examines the implications of these results for the development/execution environment as a whole.

The environment outlined is intended as much to illustrate the type of integration required and to identify problems as it is to propose a specific direction. Other possibilities certainly exist. For example, an important contemporary concept not examined here is that of *data abstraction.*[1] Environments focusing on variants of this concept, such as that discussed in Reference 2, are structured differently from the one suggested in this paper.

Objectives

The purpose of application system development may be thought of as the production of documentation, *including code.* Although this view may appear somewhat odd at first encounter, it is easily justified on the basis that documentation is not only the output of each phase of the development process, but it is also the representation of the system after development is completed. For concreteness, and to establish some terminology, we list the following general types of documentation involved:

- *System planning documentation.* This typically includes descriptions of enterprise organization, policies, and information utilization; it is used as a basis for the derivation of data processing requirements.
- *Statements of data processing requirements.* Requirements are generally described without assuming application knowledge, thereby allowing communication between data-processing and non-data-processing personnel. This level of documentation differs from the one that follows in that little structure is associated with the system. The system is described as a "black box" with certain inputs, outputs, and abstract methods of deriving the outputs.
- *Design descriptions (multiple levels).* These descriptions subdivide the previous black box into components and indicate the functions of each.
- *Implementation description.* This is the documentation executed by the hardware/software machine.

Given that the purpose of a development environment is the production of the documentation just mentioned, the major part of that environment may be thought of as a *documentation system*, consisting of the following components:

- Specifications of the documentation scheme (documentation structure, content, and means of expression).
- Software which supports documentation manipulation (entry, modification, analysis, and storage).
- The evolving documentation itself.

The view of a development environment as, essentially, a documentation system adds concreteness and focus to the search for objectives, and highlights the importance of component interrelationships. But what are those objectives? What is a "good" documentation system? In general terms, a good documentation system addresses the major problems of application systems: cost, controllability, and effectiveness. To understand what specific qualities are needed, it is necessary to investigate the relationship between the documentation system and the application system.

documentation and application systems

The relationship between the documentation system and application system is discussed in the context of the following general stages in the system life cycle: initial development, usage, and revision.

During the *initial development phase*, the documentation system serves as a means of communication between system designers and their clients (e.g., the management whose area of responsibility subsumes the system) to establish agreement on what a proposed system is to do. The documentation system also serves as a means of communication among cooperating system designers and between system designers and implementors. It is also used to express the actual implementation, that is, to communicate between people and machines.

The ease with which designs and implementations can be expressed and understood is the major determinant of application system cost. The relationship of designer/client communication to application system effectiveness is gaining recognition. For example, it has been found[3] that extensive involvement by non-data-processing professionals in the development of decision support systems is necessary to ensure commitment to such systems. Even more to the point, it is contended that effective non-data-processing professional participation in general information system development must be based on iterative development of formal high-level documentation.[4]

During the *usage phase*, the documentation system plays two major roles. First, it is used in the administration of system control. It describes the functions of the system, including those protecting

security and integrity, and allows them to be understood, reviewed periodically for adequacy, and tested.

In the case of the manual system, the management whose administrative responsibilities subsume the functions of the system are—together with internal auditors—responsible for the correct operation and adequacy of the control functions. It is now recognized[5] that adequate control can be achieved only if these individuals accept the same responsibilities with respect to automated systems. This, however, requires the existence of documentation that is accessible to non-data-processing professionals and describes all externally relevant aspects of a system precisely and completely.

During the usage phase the documentation system also influences system effectiveness in that it is responsible for the provision of information to users. For example, data base descriptions should explain what information is maintained by the installation, and program catalogs should indicate what procedures are available.

During the *system revision phase*, the documentation system provides an understanding of the current system and thus gives a basis for deciding how modifications might be incorporated. The documentation system also repeats its initial development role and thus affects both system cost and effectiveness. Effectiveness is influenced in the sense that if the process of change is too expensive, modifications may not be made, and the application systems involved may gradually become obsolete.

documentation system objectives

To summarize the results of the last section, the documentation system is best understood as a mechanism for communication among individuals and between individuals and machines.

We suggest that a documentation system can be successful as a communication mechanism to the extent that it is sufficient and accessible and to the extent that it encourages accuracy.

A documentation system is *sufficient* for a particular audience if it serves the information needs of that audience. For example, a documentation system is sufficient for the management responsible for a particular application if levels intended for their use indicate what externally visible functions are incorporated in that application. It is sufficient for programmers if levels intended for their use provide enough design detail.

A documentation system is *accessible* to a particular audience if the amount of effort required to obtain, understand, and modify the associated documentation is cost-justifiable. The amount of effort is determined, to a large extent, by the following variables:

- *Number and consistency of concepts* influence the amount of training and practice required to understand the use of the system and the content of the documentation.
- *Naturalness of notation* is the closeness to a notation that the intended audience perceives as natural. This influences both the real readability and the more subjective approachability of the documentation.
- *Storage convenience* includes both simple retrieval/update convenience and convenience in viewing information from different perspectives. For example, although relationships between application processes and data are generally identified via processing descriptions, it is useful to view that information from the perspective of data descriptions as well, in order to determine how a particular item of data is used.
- *Volume* influences effort in that voluminous documentation forces laborious change procedures and has a forest-and-trees effect on understanding. Documentation volume decreases as the power of the descriptive concepts used increases.

Documentation is accurate if all levels are consistent. A documentation system *encourages accuracy* to the extent that it is easy to compare levels and to propagate changes among levels. These characteristics are related, in turn, to the accessibility of individual levels, to the total number of levels, and to the amount of sharing among levels. Sharing can involve the sharing of descriptive concepts, description storage *(repositories)*, and particular elements of descriptions.

To summarize, we suggest as objectives for the documentation system that it be sufficient, accessible, and accurate, and that the following characteristics are needed for the attainment of those objectives:

- Information adequacy.
- Limited numbers and consistency of concepts.
- Naturalness of notation.
- Limited documentation volumes.
- Repository convenience.
- Sharing among levels.

Current status and new approaches

In this section we examine the extent to which current documentation systems meet the objectives developed above. We identify significant problems and discuss recent approaches to solving them. This discussion serves two purposes: It introduces and begins to evaluate the raw material available for the construction of an integrated documentation system, and it also provides evidence that explicit efforts toward integration are indeed necessary.

current situation

Most documentation systems in use today fall considerably short of meeting the objectives developed in the previous section. First, those high levels of documentation intended for non-data-processing professionals, when provided at all, are usually informal or formalized only to the extent of following some standardized outline. Such levels are usually quite accessible, but the information provided is rather imprecise and thus insufficient. These inadequacies have a deleterious effect on the controllability and effectiveness of systems.

On the other hand, the documentation levels that are intended for use by data-processing professionals are generally sufficient, but they are often extremely inaccessible. Voluminous detail is required; different parts of a system must be described using different, uncoordinated concepts; and the resulting descriptions are often stored in unrelated repositories.

For example, the lowest user-generated level of system documentation generally consists of a set of high-level-language programs that are intermixed with other descriptive material, such as assembly-language programs, data-definition macros, job-control-language procedures, and linkage specifications. In addition, many high-level languages are rather fragmented internally and can be seen as collections of sublanguages (e.g., declarative, algorithmic, and I/O statements). It has been (entertainingly) estimated that to construct and test a single program the implementor must be familiar with approximately twelve different languages and sublanguages.[6] This situation is directly responsible for the enormous difficulty users face in cost-justifying the implementation of new systems or the modification of old ones.

Another difficulty with current documentation systems is that there is generally a lack of sharing among different levels. This, combined with inaccessibility at the lower levels, causes the maintenance of consistency (accuracy) to be very expensive. The lack of accuracy, in turn, affects both systems control and the costs of systems modification.

Note that the problems caused by development tool fragmentation are not limited to those directly affecting the application developer. An unintegrated set of tools is more costly to develop and maintain than an integrated set. Thus, as long as the tools produced are relatively unintegrated, fewer tools will be produced, and those that are produced tend to cost more.

new approaches

The above situation was universal for many years and still most accurately portrays the situation in a standard installation. Relatively recently, some more effective approaches have come into use and descriptions of others have been published. The approaches involved fall into the following categories:

- System description vehicles.
- Module interconnection languages.
- Application packages, customizers, and generators.
- New data models.
- High-level language extension.
- Very-High-Level Languages (VHLL).

Each of these areas is discussed in more detail in the following sections. For each, a brief description is given together with an evaluation of its relevance to the objectives established in the previous section.

system description vehicles

There have been many relatively recent efforts to provide facilities above the implementation level for describing how application systems work—for example: Functional Specification Technique (FST),[7] Problem Statement Language (PSL),[8] Systems Architects Apprentice (SARA),[9] and TELL (after William Tell).[10]

Three features of such facilities are discussed here: system description vocabularies, automated repositories, and executability. System description vocabularies are common to all methods. Each method sees a system as constructed of certain kinds of components. For example, in FST, systems are considered to be built of multilevel machines, with formal communication both between adjacent levels of the same machine and among machines. In PSL, systems are considered to be built of nested processes that represent functional partitions.

To describe a system using such a facility, one identifies the types and names of its components, how the components are nested and linked, and, in a prescribed manner, what they do. What a component does is generally couched in terms of communicating with other components and system interfaces and in terms of accessing system-maintained data bases. Those terms may be descriptive (e.g., "The inventory process updates inventory-on-hand.") and/or algorithmic (e.g., "INVENTORY-ON-HAND = BGN-INVENTORY + SHIPMENTS.").

Most of these description vocabularies have either an explicit or implicit graphic interpretation. In fact, in at least one case, TELL, the primary vocabulary is graphic.

The most important property of a system description vocabulary is that it formalizes and imposes a discipline on requirement- and design-level documentation. This contributes to both the accessibility and the sufficiency of the resulting documentation. *Accessibility* is improved in that the process of generation is simplified. That is to say, one does not have to devise a framework for each instance of documentation. Furthermore, much specification can be omitted because many of the properties of the components identified are

implied by the component type designations. The use of standard concepts and the succinctness of documentation also aids in understandability. *Sufficiency* is fostered in that the use of a standard framework encourages completeness.

The two other basic features of system description methods—the use of repositories and executability—are not features of all methods. The inclusion of an automated repository further enhances accessibility. Descriptions can be more easily reviewed, updated, and subjected to various kinds of automated analysis and reporting.[8,10] Executability implies a precise means of describing component function. If present, it can increase comprehension. It can also assist in verifying design logic, and can aid in investigating performance-related aspects of a design (given suitable assumptions about real execution times).

Note, however, that an executable system description is not normally adequate as an implementation. To be useful, a system description must be relatively succinct and readable. This generally rules out the use of traditional methods of accessing system interfaces and data. For example, in FST, communication among components and between components and system interfaces is represented by the transmission of simple parameters. This representation may be used to abstract the complex, voluminous interchanges typical of interactive applications. Such an abstraction, however, is not an implementation.

module interconnection languages

Module interconnection languages have been developed as controlling frameworks for implementation to increase intercomponent consistency. Although there are many variations, they all provide for the description of the externally visible aspects of application components separately from and prior to the development of implementation code. Thus they allow the verification of interface usage during compilation, while also serving as substitutes for traditional linkage specifications. Some formulations are designed to complement a particular implementation language,[11–13] while others are applicable to many implementation languages.[14,15]

These languages, although they increase early-stage implementation effort to some extent,[16] are powerful communication vehicles. They allow parallel and consistent development of individual components and greatly simplify analyses of the effects of change. Changes to a component not relating to its use of shared data or to its interface do not affect other components, whereas components potentially affected by other types of changes can be easily located.

An important property of module interconnection languages for our purposes is that they are a form of system description language. Seen in this way, they represent a potential bridge between design and implementation level documentation, possibly connecting and/or allowing the elimination of some levels.

application packages, customizers, and generators

Application packages, customizers, and generators are methods of obtaining running applications without the use of general-purpose programming languages. We use the term *application package* to denote precoded business applications that are modifiable only by source code alteration. An *application customizer* is similar to an application package, except that it includes facilities allowing certain anticipated kinds of tailoring without resort to source code. A customizer is capable of creating a family of applications, all conforming to the same overall business pattern.

Application generators, such as the Application Development Facility (ADF),[17] the Development Management System (DMS),[18] and the research vehicle discussed in Reference 19, are like customizers, but the families of applications involved reflect data-processing patterns instead of business patterns. For example, the basic pattern presupposed by ADF is that of interactive data entry/edit for Information Management System (IMS) data bases.

A useful perspective in this area is contributed by Reference 20, in which an application generator is explained as the result of a process of program generalization. An example given there is that of a parser generator. Such a generator allows the construction of a language-specific parser by combining code embodying a standard parsing method with a grammar for the language of interest.

The essence of these facilities is that a processing pattern is built in, and the user need supply only some details, normally in declarative form. The processing pattern is often presented as combining several types of processing, such as data entry and edit. Note that it is not important whether the actual processing is performed by generated code or by an interpreter. Note also that where the processing pattern is presented as a combination of patterns, the generator can be viewed as a combination of several partial generators, each with its own input. These two observations suggest that many familiar functions can be thought of as application generators. For example, a high-level device manager can be considered an interpretively based partial application generator.

The use of customizer and generator input for implementation-level component descriptions has great potential for simplifying both component generation and review. Input to a single generator can often replace many different kinds of specifications, including data base access and display formatting. This has two significant effects on documentation accessibility: The education required to generate component descriptions is reduced, and the descriptions are succinct and easily modified.

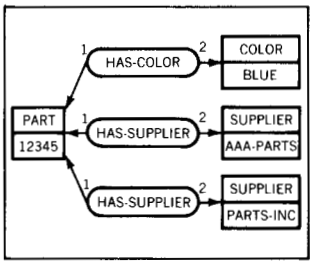
On the other hand, there is a potential problem associated with the use of customizers and generators. To the extent that the application patterns supported by generators are relatively constrained, and the

options provided are limited, ease-of-use is great. To the extent tha the patterns are relatively unconstrained, and the options are exten sive and interdependent, ease-of-use tends to be reduced. (An generators lose their advantage over general-purpose languages, especially the upgraded languages to be discussed later in this paper.) Thus, large numbers of specialized customers and generators seem to be desirable. The problem is that such numbers tend to exacerbate development tool fragmentation.

new data models

New, more abstract, physical-storage-independent data structures developed during the last decade have a considerable contribution to make in documentation systems. The most important types of structures in this class are versions of the relational[21] and Entity/Relationship (E/R) models.[22] To analyze their significance, we use as an example a version that shares characteristics of both models.

Figure 1 Data model example



This version is illustrated in Figure 1. Here there are two types of objects, entities and relationships. *Entities* are scalars and are placed in *entity sets* to indicate the type of object represented. *Relationships* are ordered associations of two or more objects (entities and/or other relationships) and are placed in *relationship sets* to indicate the type of association represented. Additional information about this particular model can be found in Reference 23.

The characteristics that make such models important for our purposes are their expressiveness, neutrality, and potential for high-level access.

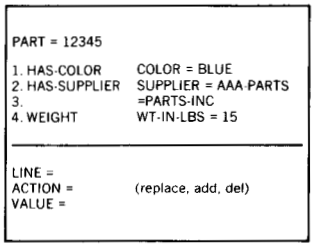
An *expressive* data model is one in which the organization of the data provides a significant amount of information about its meaning. The differences in expressiveness between models such as the one illustrated in Figure 1 and the more record-oriented ones are discussed in Reference 24. Briefly, in record-oriented models, the combination of *record type* and *field name* must convey the following: (1) what kind of object is referenced is by the field, (2) what the object is being related to, (3) by what kind of relationship, and (4) what is the role of the object in the relationship. The model illustrated provides each of these items as separate pieces of information.

Expressiveness is significant for several reasons. It enhances the probability that a user might understand the content of a data base, given only the most cursory description. Thus it is useful for data bases that are accessed only occasionally by given individuals. Repositories and casually used application data bases fall into this category. Also, given that the objects of a data model are referenced in code accessing that model, an expressive model would probably give rise to more easily understood access code.

The property of *neutrality* provides the ability to view the content of a data base from many perspectives. Thus, if one were to browse the data base shown in Figure 1, one might begin at a part, a color, a supplier, a cost, etc. In contrast, for example, accesses of hierarchic data bases generally require beginning at the top of a predefined hierarchic pattern. Where data bases must be accessed in an *ad hoc* fashion or where applications change rapidly, the requirement of access pattern predefinition is burdensome, and neutral data bases are preferable.

The *potential for high-level access* is perhaps the most important characteristic of the new data models. First, excellent graphic browse/update facilities can be devised. The interface described in Reference 25 and adapted in Figure 2 is an example of this. More important is the fact that the entities and relationships of Figure 1 are mappable onto sets, and the binary relationships are mappable onto single- and multi-valued functions. These mappings allow data to be accessed by the succinct set-oriented Very-High-Level Languages discussed later in the paper. The succinctness of these languages provides ease of coding. When the succinctness is coupled with an expressive data structure, the resulting code is very easily understood.

Figure 2 Graphic browse/update



In summary, the new data models have great potential for increasing user understanding of data base content and for simplifying both data description and data accessing.

high-level-language extension

Languages that we typically think of as High-Level Languages (HLLs), languages at the level of FORTRAN, COBOL, PASCAL, etc., are generally designed to include provisions for accessing only simple file structures. With the advent of more sophisticated data base structures (networks, hierarchies, and relations), the need arises to allow access from the HLLs. Early provisions made for this purpose were rather *ad hoc*. They generally required communication with the facilities involved via CALLs, using shared communication areas for request details and responses. These attempts were not particularly satisfactory; they made the fragmentation problem more acute and tended to obscure the meaning of the application code.

More recently, there have been attempts to extend HLLs to provide more natural, expressive interfaces to DBMS facilities, such as those described in References 26 and 27. These language extensions are generally much easier to learn than the earlier methods, and the resulting programs are easier to generate and read.

The method of language extension is not completely satisfactory, however, at least as compared with the other possibility, that is, the creation of new languages. Existing languages are already very complex. When new elements are added, not only is an increase in complexity a certainty, but that increase may also be larger than

expected. A language is made up of both individual constructs and the relationships between them. Thus the addition of one new element may require the specification of many new relationships.

In contrast, when developing a new language, one is free to reexamine the total set of desired features. Then it can be determined whether some features subsume others, whether some can be discarded, or whether generalizations of several features can be developed. Thus new language functions can be provided without an increase in complexity over existing language.

general-purpose VHLL

The term Very-High-Level Language (VHLL) is very inclusive, and there are a large number of disparate languages that can be so categorized, as can be concluded from the general discussions in Reference 28 and 29. These languages have in common an emphasis on the elimination of implementation detail—on providing the capability of specifying "what" rather than "how." Based on this characteristic, application generator inputs—being declarative—can be thought of as VHLLs. However, the term VHLL is used here (and in Reference 28) to connote general-purpose languages only, that is, languages that do not assume a particular application pattern.

One important VHLL class consists of the set-oriented languages,[28] the most well developed of which is Set Language (SETL).[30] Languages in this class obtain their power by viewing all data as being organized in sets and expressing operations in terms of set manipulation. The details eliminated are the details of accessing low-level data structures.

An interesting characteristic of set-oriented VHLLs is that one can map the constructs of some new data models to the constructs of such languages. Specifically, as previously described, one can map entities to sets of scalars and relationships to sets of vectors and to enumerated single- and multi-valued functions.

Consider the data base illustrated in Figure 1. Instead of entity and relationship sets, one might speak in terms of sets of scalars, exemplified by PARTS, SUPPLIERS, COLORS, and sets of vectors, exemplified by HAS_SUPPLIER. Then to add a part to the data base, one might specify, for example:

```
PART += "12345"
```

To assign a supplier to the part, one might specify something like:

```
HAS_SUPPLIER += <"12345", "ACME_PARTS">
```

or more attractively,

```
HAS_SUPPLIER("12345") += "ACME_PARTS"
```

Similarly, to remove the suppliers of part 12345 from the data base, one might specify:

SUPPLIER − = HAS_SUPPLIER("12345")

The right side of the last statement denotes a set of suppliers. Thus, the expressions of the language can function as a query language.

As a more powerful example, consider the following:

?p WHERE HAS_SUPPLIER(?p) = = SUPPLIER

which denotes the set of parts supplied by all suppliers.

These examples have been taken from Reference 23; related forms are suggested in References 31–33.

It is difficult to exaggerate the potential importance of set-oriented VHLLs if other necessary facilities can be handled as well as data access and if adequate execution-time performance can be obtained. The use of such languages at almost any documentation level would almost necessarily serve to increase the coherence and succinctness of descriptions at that level because much of the detail of traditional programs can be traced to the need to access data within relatively inconvenient structural contexts.

observations

We have asserted that current documentation systems are generally insufficient at upper levels and inaccessible at lower levels, the latter problem being a result of both volume and fragmentation. We have also outlined the following approaches that have great potential for increasing documentation sufficiency and accessibility:

- System description languages
- Module interconnection languages
- New data models
- Application generators
- Language extensions
- Very-High-Level Languages (VHLLs)

If these approaches are developed independently, however, the fragmentation problem may remain or even worsen. Many new tools focus on a single documentation level or category within a level. Thus if a set of such tools is incorporated into a documentation system, "as is," the fragmentation of that system may increase. This is especially true if each tool incorporates its own dedicated repository, as many do.

Also, the number of *nonredundant tools* (i.e., tools usable simultaneously) in each category may tend to increase. For example, there are many perspectives from which one may analyze a firm for system planning purposes (e.g., those of the Business Information Analysis and Integration Technique (BIAIT)[34,35] and Business Systems Planning (BSP)).[36,37] Similarly, there are complementary ways in which one might analyze a system design (e.g., performance, effect on data,

or user interface), each potentially giving rise to a different system description language. The most important source of additional diversity, however, may be in the area of customizers and generators. As previously discussed, evolution will probably be in the direction of a large number of specialized facilities, all usable to advantage within the same documentation system.

The problem of fragmentation is being actively discussed, with some treatments focusing on the implementation level alone[6,38] and others having a broader scope.[39,40] The achievement of adequate solutions, however, will require considerable exploration of alternative directions and designs by many people. Since a desired result is to minimize the number of concepts used, superficial connections between development facilities (such as allowing them to be accessed from the same menu) are of relatively little use. Instead, integrated environments must be constructed by fundamental, closely coordinated reworkings of those facilities. Such reworkings are not trivially obtained, and if only a limited set of concepts is to be retained, that set must be very carefully designed.

Documentation system construction

In this section we begin to outline a documentation system environment that may satisfy the objectives established earlier in this paper. The method used is to combine adaptations of the approaches previously introduced, in a manner analogous to that of solving a jigsaw puzzle, except that the pieces can be bent somewhat to fit. We select an initial approach and determine (1) what functions of the environment it fulfills; (2) what adaptations are required; and (3) what constraints on further selections are implied by its inclusion. We then select another, complementary approach and address the same questions, etc.

The discussion remains at a rather general level, focusing on approach selection and on major adaptations, rather than on details of individual approaches. Although the choices and adaptations are relatively well-motivated, other possibilities certainly exist. Thus the result obtained represents an illustration of the type of integration needed and a suggestion of direction, rather than a solution to the integration problem.

system description methods

Both a system description vocabulary and a repository for the descriptions are needed to document system design. The *description vocabulary* should include a means of describing system structure in terms of components and their interrelationships. It should also include methods of describing component function. Both informal (text) and formal descriptions should be used. The latter should be succinct enough to be understandable, yet precise enough to be executed, for purposes of design analysis or simulation. The *repository* should be accessible by relatively untrained users, and susceptible to *ad hoc* as well as programmed access.

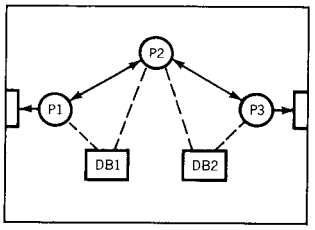
It should be possible to create several levels of design description to aid in the comprehension of large systems. A critical requirement is that all such levels use the same vocabulary and repository. Furthermore, data-processing requirement statements can be represented in the same way as design descriptions, with hypothetical structure used only to explain externally visible function.

The planning- and implementation-level documentation should also be related, wherever feasible, to the design-level descriptions and should share the same repository. The discussion here does not cover system planning documentation, but examples are given in References 34–37. We note here only that, although the planning and implementation vocabularies must differ from the design vocabulary (as they deal with different subjects and are generally more application-oriented), linkages should be created in expression of elicited or derived requirements.

Considerable potential exists for the close linkage of implementation descriptions (code) with design descriptions. First, the design-level system structure descriptions might be augmented to serve module interconnection purposes as well. Structural descriptions then cascade from the general design to a level that includes both design information and module interconnection information (e.g., precise interfaces) to lower levels described in terms of module interconnection alone. To do this, the design-level model of system structure must correspond to a feasible implementation-level model.

For example, the simplified model of Figure 3, representing a synthesis of many models, might be used. It consists of three kinds of elements: (1) *Processes*, which are asynchronous units of execution; (2) *Data bases*, which are collections of data maintained within the system and accessed by processes; and (3) *Messages*, by which processes communicate with each other and with system interface points. Processes and data bases that have single instances in the system are distinguished from those potentially having as many instances as users of a particular type.

Figure 3 System model



A design description using this model includes (1) *A structural description* identifying the processes, data bases, and their interrelationships; (2) *Formal descriptions* of process functions; and (3) *Descriptions of data base content*. The data model assumed in describing data base content has to be consistent with that assumed in describing the processing of the data.

A further means of connecting design and implementation level descriptions is to use identical methods of describing the processing and the data. This does not mean that the descriptions themselves are identical. For example, design descriptions tend to summarize user interface handling by abstract formulations of net inputs and outputs, whereas implementation descriptions must include precise

descriptions of prompts, responses, etc. The feasibility of this approach rests on performance questions and is discussed further on.

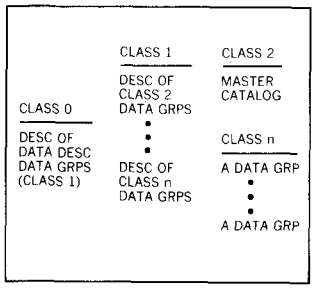
new data models

We have identified the significance of the newer data models as lying in their expressiveness, neutrality, and potential for high-level access. We now use these structures to fill gaps in the environment under construction. To begin, they are clearly appropriate for the repository, as they are suitable for data bases that are subject to infrequent access via unanticipated paths. The system description method PSL uses what may be viewed as an early approximation of an Entity/Relationship (E/R) model as its repository vehicle, and it benefits accordingly.

We now also make more detailed decisions about the repository to lend concreteness to the subsequent discussion. We assume that the repository does not consist of a single data base; rather it is made up of a collection of data bases or data groups, each organized according to the example data model discussed earlier. These groups are related by a *master catalog data group*. Each *data group* contains a particular kind of information about a particular element of the application system. For example, one data group may contain system planning information and another may describe an application data base.

There are several reasons for the use of a multiple data group descriptive repository. We assume that the descriptive repository is used as a development focus for all applications and application systems associated with a particular physical system—possibly distributed. As such, the somewhat decentralized approach has distinct logistic advantages in the areas of naming conventions, data sharing, and change control. Another reason for the choice is discussed further on. Note that there need be no loss of function as compared to a single data group repository approach, except possibly some built-in integrity checking.

Figure 4 Repository



Each descriptive data group in the repository is itself described by a built-in data group, rendering the repository self-describing. Each collection of data groups having the same descriptor constitutes a *data group class*. Figure 4 illustrates some of these ideas. Class 0 is special and circular. Its single contained data group describes the class of data description data groups.

Besides the repository, the data model is also ideal for the modeling of application data at design levels. Descriptions of system functions at those levels generally attempt to convey what information is involved and what is done to it in abstract terms, that is, in terms that are relatively independent of performance-oriented data structure selection. E/R and E/R-like models are well suited to this task. In fact, many early discussions of such models termed them "concep-

tual models,"[41] and considered their most important use to be in describing the information content of a data base.

The use of the same data model both for system description data and for the modeling of application data at the design level is an important step in reducing the number of descriptive concepts needed. It might be noted, however, that different design levels may reflect different partitionings into data groups of enterprise data. To use an extreme example, at a higher level all the data might be represented as a single data bank. At a lower level, decisions as to the allocation of the data to different data bases might be expressed (together with the processing implications).

This brings us to the other reason for the use of multiple data groups for systems description data. Coherence seems much better served by making no fundamental distinctions between data used to describe application systems and data operated upon by application systems. Inasmuch as the latter is clearly partitioned among data groups, there is no reason why the former should not be also.

We finally consider using the data model for internal data at the implementation level. If one draws a boundary around an automated system and considers all data retained within that boundary as being internal data, note that no one ever sees the internal data as stored. This contrasts with information passing over the system boundary: human interfaces, portable files, etc. The format of internal data is relevant only for performance reasons; if the use of conceptual structures simplifies the implementation-level descriptions, only optimization concerns stand in the way of their use. These concerns are discussed further in connection with languages later in this paper.

application packages, customizers, and generators

In an earlier section we concluded that customizers and generators had great potential for reducing the education and effort needed to generate and maintain implementation-level component descriptions. Thus we now include them in our environment. To connect this decision with earlier ones, we can specify that generator inputs be placed in repository data groups (of generator-specific classes). To create an application, a user creates the appropriate data groups and invokes the generator. If the application is a component of a larger system, the module interconnection specification is used to indicate that those data groups, together with the appropriate generator, constitute the implementation of that component.

These decisions do not exhaust the set of basic issues connected with the inclusion of generators in our environment. Those to be considered here are the following: (1) the problem of generator multiplicity (introduced earlier); (2) the extent to which generators solve the implementation problem; and (3) the implications of generator use for design-level documentation.

multiplicity of generators

Recall that a multiplicity of simple generators may be required because the alternative may be a few generators with large numbers of options. To allow this without adding to the complexity of the environment, carefully constructed descriptive material and usage instructions must be provided for each facility. The word "catalog" captures the desired effect. The selection of a facility appropriate to a need should be comparable in simplicity to the use of a mail order catalog. Also, diversity should be limited to that required. Concepts presented by generation facilities relating to aspects of the application system outside their domain—such as data bases and interprocess communication—should be presented similarly throughout the generator catalog.

pervasiveness of generators

Does the introduction of generators represent a complete solution to the problems of implementation-level specification? Does it obviate improvements to more general-purpose methods of application implementation? The position taken here is that it does not. No matter how many facilities are provided, they will not cover all needs. At least some applications will have to be constructed by more traditional means. Also, many applications will be constructed by applying source-level modifications to application packages. It is mentioned in Reference 42 that one reason for the wide use of application packages is the extreme difficulty of estimating development costs for new applications. In other words, application packages are often used—even when extensive source modifications are needed—as a way of lowering project risk.

Thus improvements to general-purpose languages are still relevant. In fact, one of the more important kinds of improvements may be in the area of simplifying the construction and modification of application generators to permit user installations to create generators for their own use.

relation of generator input to intermediate-level documentation

One purpose of elaborate design documentation is to organize the work of the substantial number of programmers normally involved in an application system development project. Given the availability of customizers and the associated reduction in required programmers, there may well be a temptation to avoid the generation of design-level documentation, especially processing descriptions, for small-to-medium-size systems. If this is done, however, the other benefit of design documentation is lost, namely the understanding of system operation as a whole. This understanding is needed for reasons discussed earlier in this paper, including management control, and is not normally provided by generator input for the following reasons:

- The different generators used in a single system generally have different input vocabularies.
- Generator inputs often center on aspects of the application that are unrelated to system flow (such as screen formats), and thus tend to obscure that flow.

This suggests that customizers and generators be asked to produce not only executable code, but also design documentation. Another—possibly less practical—approach is to use a common language for generator output, one sufficiently expressive to be susceptible to automated abstraction to a design level.

These comments on the use of application packages, customizers, and generators in the environment under construction may be summarized as follows:

- These facilities should constitute an important part of the environment.
- Their use, however, does not make improvements to general-purpose languages unnecessary.
- One important extension to general-purpose languages might be the addition of aids for the construction of generation facilities.
- Generators might emit design-level processing descriptions as well as code.

Very-High-Level Languages

The environment as constructed up to this point appears to require general-purpose language capability in the following contexts:

- Accessing the repository.
- Representing processing at the design and implementation levels.
- Implementing application customizers, generators, and packages.

Two candidate approaches to general-purpose language have been outlined: extensions to existing high-level languages and new languages at a still higher level (VHLLs). Both approaches have drawbacks. The complexity problems of the extension approach have already been mentioned. There is also the probability that an extended language is less powerful (and thus less succinct) than a set-oriented VHLL. Some extensions proposed for data base access, such as those discussed in Reference 43, provide more power in manipulating the data involved than is provided by the base languages for manipulating local data. Thus, it is reasonable to suppose that, at the very least, a VHLL can extend that power to all data accesses. On the other hand, the VHLL approach presents performance problems, in that current optimization technology cannot guarantee performance equivalent to HLL-coded applications in many cases.

One possibility is to try to use the VHLL approach for all the above purposes except where performance is an issue, that is, in implementation-level component description. To minimize the total number of concepts used in the environment, however, it is preferable to use the same general-purpose language throughout. In the remainder of this section we investigate the feasibility of this more consistent approach, examining the implications and problems of VHLL usage in each of the above contexts, except that application generator implementation is not addressed further.

VHLL at design level

The use of a VHLL for design-level processing descriptions presents no intrinsic problems. To provide a better picture of a VHLL design-level specification, we now flesh out our assumed language to some extent.

We assume that the VHLL to be used is similar to that previously discussed, and accesses data organized according to the model in Figure 1. This implies the use of that model for internal data at the design level. The data may be divided among many data groups associated with different areas of the application and having different lifespans. Data groups local to a process, i.e., existing only while the process is active, should be declarable within that process, using declarative text closely related to the form of global data descriptions.

The second subject to be addressed with respect to VHLL at the design level is that of language style, and more specifically, that of procedurality. Some VHLLs use adapted forms of classic procedural syntax—permitting loops, explicit processing sequences, and so forth. It might be objected that this is improper because such languages must indicate "what," not "how." A procedural style should not be ruled out, however, for the following reasons.

Sometimes a specific sequence must be followed to obtain a desired result. For example, in an order processing program, the orders must be processed one-by-one even if multiple processors are available.[44] Also, it is very often easier to express a desired result as the outcome of a series of steps than by other means. In fact, relatively pure specification languages are often accessible only to the mathematically sophisticated. In any case, much of the procedurality that hinders accessibility in traditional programs is due to low-level data manipulation.

Thus, procedurality should be acceptable to some extent in a VHLL. For purposes of optimization, if the language is properly constructed a compiler should be capable of distinguishing between necessary and optional sequencing, via data dependency analysis, for example.

The third issue discussed here in connection with the use of VHLL at the design level is that of interprocess communication. Referring back to the system structure assumed (processes, data bases, and messages), it is clear that some provision must be made for messages. By adapting elements of many approaches, such as, for example, those of References 45–47, to the needs of our VHLL, we can specify that a message consists of a list of values, with each value being either a set or an entire data group. We may then add statements to the language to SEND and RECEIVE messages into and out of specified local and global data groups.

Having made these assumptions about the VHLL, we might now visualize a design-level system description as consisting of a struc-

ture-description data group that identifies system components and their interrelationships, and a set of process and data description data groups detailing those components. Process-description data groups contain VHLL code represented as relationships between statements and statement numbers.

VHLL for repository processing

The VHLL elements needed for the simple accessing of repository data groups are the same as those used for describing processing at the design level. Therefore, in this section we focus on the use of a VHLL for the production of design analyses and reports in display or hard-copy form. For this purpose, methods are needed for communicating with system interfaces succinctly and expressively. This is a critical area because the suitability of the methods developed will have great influence on the overall usability of the language. It is speculated[29] that the relative lack of success of extant VHLLs can be attributed to the fact that although they "make it simpler to code small parts of a program," they do not "significantly ease the problem of overall program formulation and organization." It is quite possible that the lack of appropriate facilities for system interface handling is an important aspect of this problem.

Examples of the types of facilities needed are found in References 33 and 48. However, these proposals, in which the interface facilities are included in the language directly, seem to unnecessarily constrain the types of devices that can be accommodated and the levels at which they can be addressed. We therefore look to the use of partial application generators in one or more of the following forms:

- *Interpretive interface processes.* These generator-equivalents would be used as intermediaries in communicating with a device at a particular level. For example, to create a display, one might send a message of a prespecified class to a display interface process. The message is to contain format, content, and processing (e.g., editing) information. The amount of detail sent would depend on the level of the interface process.
- *Module generators.* These would resemble application generators but have significantly narrower scope.
- *Subroutine generators.* These would be similar to module generators, but they would operate on generator-input blocks containing declarations imbedded in VHLL code rather than on separate generator-input data groups. This approach can be thought of as a generalization of the method in Reference 48. It conveniently localizes generator inputs and allows generators to share local data descriptions with the VHLL language processor. However, it requires provisions both in the definition of the VHLL and in the construction of its processor.

The use of partial generators, especially if extended to the implementation level, has the additional advantage of increasing the range of generator applicability while reducing the need for generator proliferation.

VHLL at the implementation level

The major impediment to the use of a VHLL at the implementation level is that of performance. Simply stated, because VHLLs specify "what" rather than "how," the compiler is left to decide "how." This matter is not resolved here. Rather we attempt to demonstrate that the problem is less serious than might be supposed.

Among the factors involved is the significant amount of work that has been done in such relevant areas as classic optimization of high-level languages,[49,50] related optimization of very-high-level languages,[51–55] automatic selection of data storage structures,[56,57] data base access optimization,[26,58] and artificial-intelligence-oriented program synthesis.[59,60] (The preceding references are a representative rather than an exhaustive set.) This work has not advanced sufficiently to solve the problem of optimizing VHLLs, but it does represent a significant body of research.

Another important factor is that current application trends seem to decrease the need for the more difficult global optimizations. To be more specific, interactive transactions on increasingly integrated data bases account for an ever-growing proportion of application code. The relevance of the trend toward interactive transactions is that the optimization methods most relevant to this type of processing, namely data access optimization and relatively local VHLL optimization, are among the more tractable ones. Data access optimization has probably reached the point where an automatically determined strategy is as good as a manually specified one. Local VHLL optimization, although less advanced, shows promise.

The increase in data base integration is important because it implies the need for relatively neutral storage methods, thus lessening the importance of storage structure selection optimization.

A final factor affecting VHLL feasibility is the continuing decline in hardware costs relative to programming costs. Although automatic implementations will probably not match good HLL implementations for some time, the real question is whether the cost of a manual implementation is less than the cost of buying faster/larger hardware to make up for any inefficiencies.

The really difficult cases are large batch applications. Here, the arbitrary use of a VHLL without adequate optimization can lead to greatly reduced performance. In such cases, the user might be advised that the compiler preserves at least the overall logic of the code, so that major subdivisions of the program must be carefully decided upon. Alternatively, the HLL extension approach might be used.

Implications for the total environment

Up to this point, we have focused on the documentation-system aspect of the development/execution environment. Important omissions from the discussions are the following:

- Command languages that activate development and production processing.
- Repositories for the execution environment.
- Facilities for installing new function in the execution environment.

These subjects are not addressed here in the same detail as those of earlier sections. Instead, the discussion is limited to some general suggestions that are consistent with the documentation system structure and with the overall objective of environment integration.

The most obvious suggestion is that the development and execution environments be the same and thus that there be a single command language and a single repository. Furthermore, since command languages have evolved over the years from simple facilities for data binding and application invocation to programming language equivalents, it is reasonable to propose that a form of the VHLL be used as the command language for the environment. This would contribute to overall consistency and provide a built-in capability for *ad hoc* data access to both development and application data.

If the repository is to be extended to include application data groups, it must acquire additional organization, such as subdivisions for purposes of data group naming, ownership, and so forth. Also, to make the repository as inclusive as possible, special data groups should be allocated for specifications governing system authorization and accounting.

The subdivisions of the repository, if allowed to contain running processes as well as data groups, might also be used to represent work contexts, i.e., subenvironments used as a basis for resource utilization accounting, symbol resolution, application of defaults, etc. A given work context might be associated with a particular user or with a particular centralized application system function. Processes in one context must be able to access data in and send messages to other contexts.

Given this structure, the installation of an application system basically involves associating its components with the appropriate work contexts. To allow the procedure to be automated, the design-level structure descriptions might be augmented to group processes and data by target contexts or context types.

Concluding remarks

In this paper we have attempted to demonstrate two points: (1) that there is a need for increased attention to environment integration, and (2) that such integration requires a fundamental reworking of various approaches rather than the establishment of superficial connections.

We have also suggested a particular direction for such integration, with the following as its most important characteristics:

- The use of a coherent system repository for all descriptive and application data.
- The use of a highly expressive data model that shares characteristics of the Entity/Relationship (E/R) and relational models for that repository and thus for all data maintained within a system.
- The use of closely related design- and implementation-level description methods, thereby allowing the documentation of these levels using similar concepts and providing inter-level comparability.
- The use of a multitude of partial and full application generation capabilities.
- The use of a single general-purpose language for all non-generator-based processing description, as well as for system description analysis.

Many questions and problems have been raised and left unresolved in the development of the basic direction. Two of the most important were (1) how to use VHLLs in the building of application generators, and (2) how to provide significant optimization for VHLLs. These questions should be considered important subjects for future research.

ACKNOWLEDGMENTS

I thank D. W. Low, J. G. Sakamoto, R. C. Summers, and B. P. Whipple for their careful reviews and many helpful suggestions.

CITED REFERENCES

1. B. Liskov, *An Introduction to CLU*, Memo No. 136, Computation Structures Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA (1976).
2. Xerox Learning Research Group, "The Smalltalk-80 System," *BYTE* **6,** No. 8, 36–48 (August 1981).
3. M. J. Ginzberg, "Redesign of management tasks: A requirement for successful decision support systems," *MIS Quarterly* **2,** No. 1, 39–52 (March 1978).
4. H. F. Juergens, "Attributes of information system development," *MIS Quarterly* **1,** No. 2, 31–41 (June 1977).
5. *Systems Auditability and Control Study*, Institute of Internal Auditors, Altamonte Springs, FL (1977).
6. J. R. Ehrman, "The new tower of babel," *Datamation* **26,** No. 3, 156–160 (March 1980).

7. M. Berthaud, "Towards a formal language for functional specifications," *Proceedings of the IFIP Working Conference on Constructing Quality Software*, North-Holland Publishing Co., New York (1977), pp. 379–396.
8. D. Teichroew and E. A. Hershey, "PSL/PSA: A computer-aided technique for structured documentation and analysis of computer-based information systems," *IEEE Transactions on Software Engineering* **SE-3,** No. 1, 41–48 (January 1977).
9. G. Estrin, "A methodology for design on digital systems—Supported by SARA at the age of one," *AFIPS Conference Proceedings, National Computer Conference* **47,** 313–324 (1978).
10. S. N. Zilles and P. G. Hebalkar, "Graphical representation and analysis of information systems design," *Data Base* **11,** No. 3, 93–98 (Winter–Spring 1980).
11. J. L. Archibald, *The External Structure: Experience with an Automated Module Interconnection Language*, Research Report RC8652, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (January 1981).
12. J. G. Mitchell, W. Maybury, and R. Sweet, *Mesa Language Manual, Version 5.0*, Xerox Palo Alto Research Center, Palo Alto, CA 94304 (April 1979).
13. N. Wirth, "Lilith: A personal computer for the software engineer," *Proceedings of the 5th International Conference on Software Engineering*, March 1981, pp. 2–15.
14. L. W. Cooprider, *The Representation of Families of Software Systems*, Technical Report AFOSR-TR-79-0732, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA (April 1979).
15. W. F. Tichy, "Software development control based on module interconnection," *Proceedings of the 4th International Conference on Software Engineering*, Institution of Electrical Engineering, London (September 1979), pp. 29–41.
16. D. L. Parnas, "On the design and development of program families," *IEEE Transactions on Software Engineering* **SE-2,** No. 1, 1–9 (March 1976).
17. *IMS Application Development Facility, General Information*, IBM Reference Manual, Order No. GB21-9869-1 (November 1978); available through IBM branch offices.
18. *Development Management System, General Information*, IBM Reference Manual, Order No. GH20-2195 (January 1979); available through IBM branch offices.
19. E. D. Carlson and W. Metz, *A Design for Table Driven Display Generation and Management Systems*, Research Report RJ2770, IBM Research Laboratory, 5600 Cottle Road, San Jose, CA 95193 (March 1980).
20. P. Lucas, "On the structure of application programs," *Lecture Notes in Computer Science 86: Abstract Software Specifications*, Springer-Verlag, New York (1980).
21. E. F. Codd, "A relational model for large shared data banks," *Communications of the ACM* **13,** No. 6, 377–387 (June 1970).
22. P. P.-S. Chen, "The entity-relationship model—Toward a unified view of data," *ACM Transactions on Database Systems* **1,** No. 1, 9–36 (March 1976).
23. P. S. Newman, *An Atomic Network Programming Language*, Report G320-2704, IBM Scientific Center, 9045 Lincoln Boulevard, Los Angeles, CA 90045 (June 1980).
24. W. Kent, "Limitations of record-based information models," *ACM Transactions on Database Systems* **4,** No. 1, 107–131 (March 1979).
25. R. G. Cattell, "An entity-based database user interface," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM, New York (May 1980), pp. 144–150.
26. D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, "SEQUEL 2: A unified approach to data definition, manipulation, and control," *IBM Journal of Research and Development* **20,** No. 6, 560–575 (November 1976).
27. C. J. Date, "An introduction to the Unified Data Language (UDL)," *Proceedings of the 6th International Conference on Very Large Data Bases*, October 1980, pp. 15–27.

28. W. A. Wulf, "Trends in the design and implementation of programming languages," *IEEE Computer* **11,** No. 9, 14–25 (January 1980).
29. M. Hammer and G. Ruth, "Automating the software development process," *Research Directions in Software Technology*, P. Wegner (Editor), MIT Press, Cambridge, MA (1979), pp. 767–790.
30. J. T. Schwartz, *On Programming, An Interim Report on the SETL Project,* Computer Science Department, Courant Institute for Mathematical Sciences, New York University, New York (1973).
31. D. W. Shipman, "The functional data model and the data language Daplex," *ACM Transactions on Database Systems* **6,** No. 1, 140–173 (March 1981).
32. N. Goldman and D. Wile, "A database foundation for process specifications," *Proceedings of the International Conference on Entity-Relationship Approach to Systems Analysis and Design*, Los Angeles, CA (December 1979), pp. 426–445.
33. M. Hammer and B. Berkowitz, "DIAL: A programming language for data intensive applications," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, ACM, New York (May 1980), pp. 75–92.
34. W. M. Carlson, "Business Information Analysis and Integration Technique (BIAIT)—The new horizon," *Data Base* **10,** No. 4, 3–9 (Spring 1979).
35. D. V. Kerner, "Business information characterization study," *Data Base* **10,** No. 3, 10–17 (Spring 1979).
36. *Business Systems Planning—Information Systems Planning Guide*, Order No. GE20-0527-2 (October 1978); available through IBM branch offices.
37. J. G. Sakamoto and F. W. Ball, "Supporting Business Systems Planning studies with the DB/DC Data Dictionary," *IBM Systems Journal* **21,** No. 1, 54–80 (1982, this issue).
38. P. A. Demers, "System design for usability," *Communications of the ACM* **24,** No. 8, 494–501 (August 1981).
39. A. I. Wasserman and C. J. Prenner, "Toward a unified view of data base management, programming languages and operating systems—A tutorial," *Information Systems* **4,** 119–126 (1979).
40. T. Winograd, "Beyond programming languages," *Communications of the ACM* **22,** No. 7, 391–401 (July 1979).
41. H. Biller and E. J. Neuhold, "Concepts for the conceptual schema," *Architecture and Models in Data Base Management Systems*, G. M. Nijssen (Editor), North-Holland Publishing Co., New York (1977), pp. 1–30.
42. J. F. Collins, G. J. Feeney, and J. Gosden, "Calling a spade a spade. . . . A chat with MIS executives," *Datamation* **25,** No. 11, 13 (November 25, 1979).
43. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," *Proceedings of the 1979 SIGMOD Conference* (1979), pp. 23–34.
44. S. Jones and P. Mason, "Proceduralism and parallelism in specification languages," *Information Systems* **5,** No. 2, 97–106 (November 1980).
45. R. P. Cook, "*MOD—A language for distributed programming," *IEEE Transactions on Software Engineering* **SE-6,** No. 6, 563–571 (November 1980).
46. J. A. Feldman, "High level programming for distributed computing," *Communications of the ACM* **22,** No. 6, 353–368 (June 1979).
47. B. Liskov, "Primitives for distributed computing," *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 33–42.
48. J. M. Lafuente and D. Gries, "Language facilities for programming user-computer dialogues," *IBM Journal of Research and Development* **22,** No. 2, 145–158 (March 1978).
49. F. E. Allen and J. A. Cocke, "A catalog of optimizing transformations," *Design and Optimization of Compilers*, R. Rustin (Editor), Prentice-Hall, Inc., Englewood Cliffs, NJ (1972), pp. 1–30.
50. R. G. G. Cattell, "Automatic derivation of code generators from machine descriptions," *ACM Transactions on Programming Languages and Systems* **2,** No. 2, 173–190 (April 1980).
51. J. T. Schwartz, "Optimization of very high level languages—I," *Computer Languages* **1,** No. 2, 161–194 (June 1975).
52. J. T. Schwartz, "Optimization of very high level languages—II," *Computer Languages* **1,** No. 3, 197–218 (September 1975).

53. F. Arbab, *Notes on the Semantics and Optimization of a VHLL*, Report G320-2706, IBM Scientific Center, 9045 Lincoln Boulevard, Los Angeles, CA 90045 (October 1980).
54. W. H. Burge, *An Optimizing Technique for High Level Programming Languages*, Research Report RC5834, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (February 1976).
55. E. J. Neuhold, *The Design of a Business Definition Language Compiler*, Research Report RC6475, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (April 1977).
56. J. R. Low, "Automatic data structure selection: An example and overview," *Communications of the ACM* **21**, No. 5, 376–385 (May 1978).
57. P. D. Rovner, "Automatic representation selection for associative data structures," *Proceedings of the AFIPS National Computer Conference* **47**, 691–701 (June 1978).
58. M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of INGRES," *ACM Transactions on Database Systems* **1**, No. 3, 189–22 (September 1976).
59. S. Fickas, "Automatic goal-directed program transformation," *Proceedings of the National Conference on Artificial Intelligence*, August 1980, pp. 68–70.
60. E. Kant, "The selection of efficient implementations for a high-level language," *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, August 1977.

The author is located at the IBM Scientific Center, 9045 Lincoln Boulevard, Los Angeles, CA 90045.

Reprint Order No. G321-5162.