Architecture prototyping in the software engineering environment

by W. E. Beregi

This technical essay presents a perspective on the evolution and problems of the software development craft and how software engineering techniques show promise to solve these problems. It introduces architecture prototyping as a program development technique for improving software quality. Experience with large software systems shows that over half of the defects found after product release are traceable to errors in early product design. Furthermore, more than half the software life-cycle costs involve detecting and correcting design flaws. In this paper, we explore a disciplined approach to software development based on the use of formal specification techniques to express software requirements and system design. As a consequence, we can use techniques like rapid prototyping, static design analysis, design simulation, and dynamic behavior analysis to validate system design concepts prior to element design and implementation. We explore how these techniques might be organized in a software architecture prototyping facility that would be similar to the Computer-Aided Design and Manufacturing (CADAM) tools used in other engineering disciplines. We also examine the process by which software engineers might use these facilities to create more reliable systems.

Personal computers, office systems, and professional workstations are on the verge of recruiting a large new class of computer users. This user explosion will accelerate the demand for reliable software applications and systems. Many applications in medicine, aerospace, and real time process control require high system reliability and defect-free software. Yet software developers are beginning to reach a plateau in terms of the quality and quantity of systems they can produce. Maturing software development techniques of the 1970s like the use of structured programming constructs and design and code inspections[1-3] squeeze fewer new errors out of software and yield fewer productivity gains from programmers.

The reason for this leveling is that the development of software is a complex intellectual task. That task involves understanding the user's world (analysis), defining needs for improvement in that world (requirements), creating a conceptual solution to satisfy those needs (design), and translating that solution into a form executable on a machine (implementation). Because few of us can manage this as a single task, we attack large software design problems in teams, using a divide-and-conquer strategy like top-down, stepwise refinement.[4-8] At any stage of design, given a specification that describes what a component of the system should do, we can derive how it should do the "what" by decomposing the component into smaller, more manageable functions that collaborate to perform the component's task. We can then specify what each of the functions should do and pass these more detailed specifications on to other designers to elaborate how each function is to perform its specified task. Most of software design can be produced using this iterative, hierarchical decomposition of complex, abstract specifications into simpler, atomic specifications, until those specifications can be compiled and run on a machine.

We begin to realize why programmers struggle to produce better software when we compare the craft

they use with the complexity of the task. The software development techniques widely used today provide aid when the problem is relatively small, local, and concrete. An example might be the implementing of a SAVE function in a file editor. These techniques provide little help, however, when the problem is as large, global, and abstract as, for example, that of describing the architecture and protocols in a telecommunications system. Widely used natural-language-based software methodologies weaken when we tackle large, multiperson design projects. Without tools to define design precisely, to partition design into mutually exclusive units of manageable complexity, to keep track of the relationships of the parts to the whole, to validate system design ideas prior to refinement and implementation, and to facilitate the reuse of previous design solutions, we will have difficulty producing zero-defect software. We examine some of these deficiencies and their consequences for software quality in later sections of this paper.

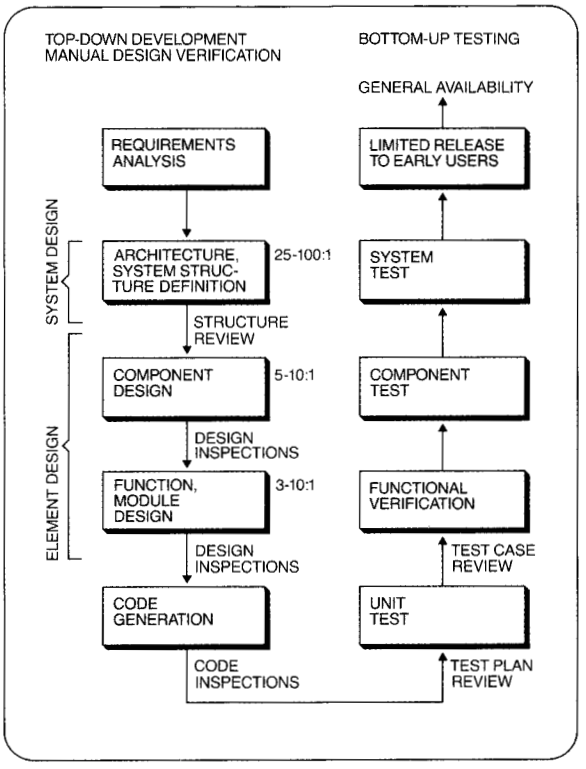
Developments in software engineering and software specification techniques promise to infuse our craft with some of the engineering discipline we need to improve software quality and productivity. These methods offer to software developers some of the specification and verification techniques available to hardware logic designers, building architects, and aircraft and automotive engineers. We examine how these methods can be used to improve software quality. We explore how these techniques might be integrated into an architecture prototyping facility that software engineers could use to test the feasibility of their early software design ideas.

The software development craft

Software developers practice their craft using some common strategies to manage complexity. In this section, we trace through a simplistic overview of the techniques most widely used in the software industry to develop programs, ignoring for the moment software engineering approaches that have begun to penetrate into practice. We point out major deficiencies of these techniques in this examination. Engineering approaches used in other disciplines also suggest ways to improve the software craft.

Most large software systems developed today are produced using a staged approach, which includes discrete phases for requirements definition, system design (which we call architecture), internal, detailed element design, and implementation and verification. This staged approach is illustrated in Figure 1. We describe the essence of these operations in the following sections.

Figure 1 Stages in the software development process



Requirements phase. Product planners, analysts, consultants, and users collaborate to define *product requirements*. Planners use enterprise analysis techniques[9,10] to understand the existing user system and environment, to collect user requirements for improvements, and to organize the requirements into related problems and opportunities. Planners then identify potential solutions to satisfy each problem, rank solutions by user demand, cost, and projected revenue, and define a product description in terms of the functions the product must provide and the constraints (e.g., performance and availability) the product must satisfy.

An English-language document is typically written to communicate the requirements to the development team. Planners and developers validate this document.

Architecture phase. Software architects produce from the requirements statement a *system architecture specification.* This specification has two parts: (1) what the system must do or provide (i.e., the external specifications), and (2) how the system must operate (i.e., internal specifications). The internal specifications define both a structural framework of subsystems that collaborate to perform system functions and system-level dynamic behavior. These specifications identify system components, system control and data flow, and how components are bound together in the structure. They postpone the definition of internal design elements and their operations until later stages of design.

Architecture is commonly expressed in natural language, sometimes augmented with diagrams.

To illustrate by analogy, software architecture specifications serve the same function as the architectural plans and drawings used by building architects and civil engineers. A blueprint portrays the building structure and the static relationships of parts to the whole. Blueprint annotations may describe attributes of facilities (e.g., size and material composition) and criteria these facilities must satisfy (e.g., performance under stress conditions). The architectural plan describes the workings of the moving parts and the dynamic operation of the facilities throughout the structure (e.g., electrical, plumbing, and elevator systems). The plan may reference design documents that describe the internal operations of the building's component facilities.

To summarize, architecture specifications describe, at minimum, three aspects of the system:

- *External functions* and interfaces presented by the system to the user.
- *Structural framework* of subsystems and interfaces that mesh to provide system functions.
- *Dynamic behavior* of the system (data and control flow through the structural framework) that results when a user requests a system function. Asynchronous aspects and timing relationships in the system are defined at the architecture level.

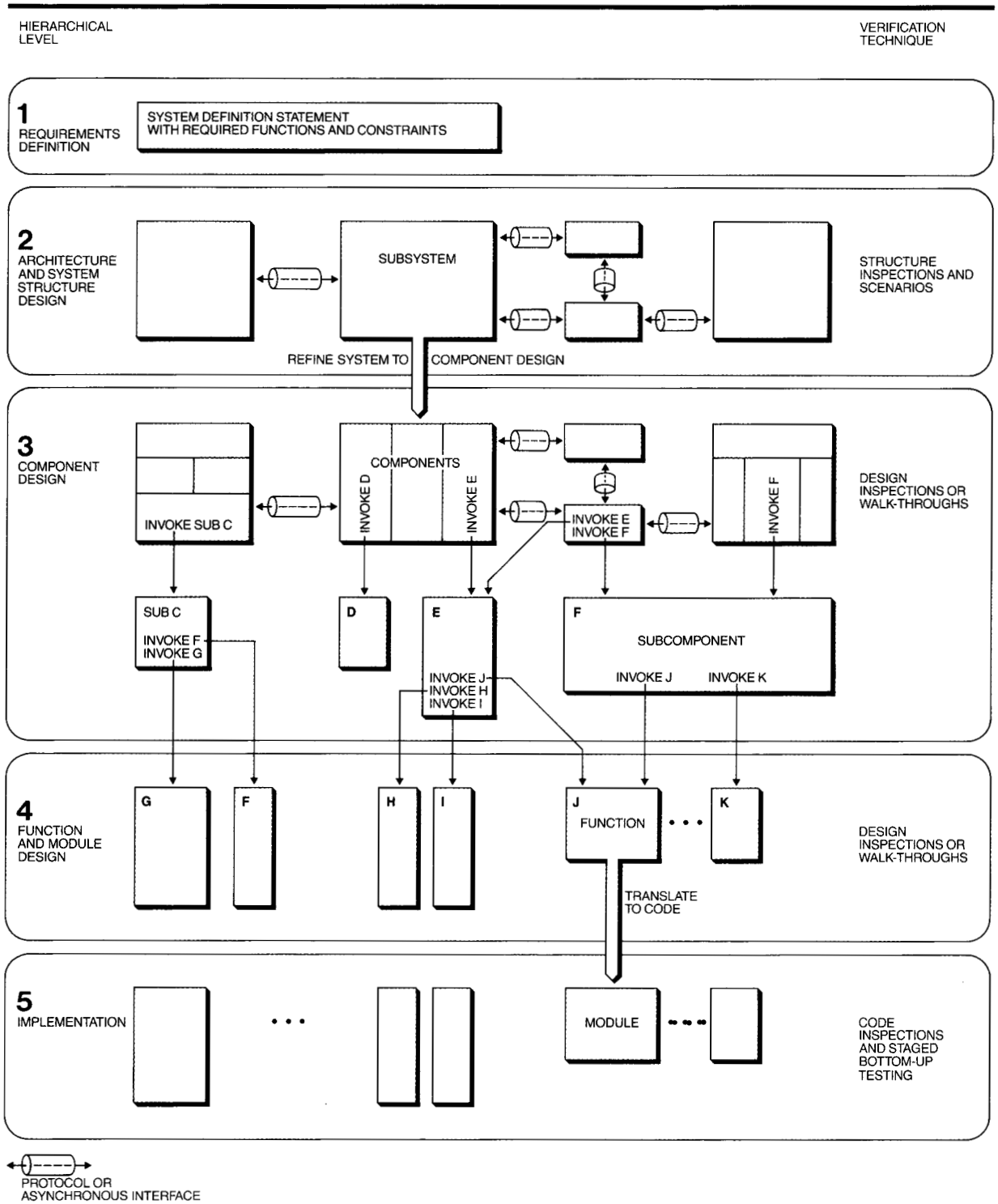
Architecture is commonly expressed in natural language, sometimes augmented with diagrams (e.g., Hierarchical Input-Process-Output (HIPO) diagrams[11]). Architecture specifications are described at a very high level of abstraction. One statement may represent between twenty-five and one hundred statements in the product implementation. They are sometimes verified using *structure inspection* or *scenario flow* techniques.[12]

Design phases. Once an architecture framework is defined, designers can fill in the details of internal elements using iterative refinement techniques. As shown in Figure 2, they use Structured Design, Composite Design, and Data Flow Design principles[13–15] to elaborate operational details for components referred to but not explicitly defined in the architecture framework. They apply the same principles to decompose and elaborate details of functions referenced in the components and modules (segments) referenced in the functions, until all system segments are designed.

One statement at the element design level of abstraction may be expanded into three to fifteen statements in the product. Design at this level is expressed using a wide variety of notations (e.g., natural language, pseudo-code, structured flowcharts, and decision tables). Multiple notations often coexist on a project, thereby forcing designers to transfer from one notation to another as they refine the design. Defects introduced during design are often manually removed from the product by means of inspections[1–3] before the next level of design begins. Defects that escape are often not detected until late in the product test phase.

Implementation and verification phases. Programmers implement the design, using primarily high-level programming languages (e.g., PL/I and Pascal). They then collect and test the coded product modules. One verification strategy is a bottom-up testing approach, by which simple system elements (modules) are tested first, then successively recomposed into more complex entities, such as functions, components, and systems, to be tested in more complex environments. Once all elements are tested and in-

Figure 2 Top-down stepwise refinement in the software development process and hierarchical relationships among levels of design abstraction



tegrated, the system is exercised to verify that it meets function, performance, reliability, and ease-of-use specifications.

Thus, the mainstream software production environment today relies on top-down development techniques, specification in natural language, successive refinement of design, manual design verification (using inspections), and late life-cycle verification that the product meets requirements and specifications. Although this process contributes to good software quality, the lack of tools and precision at several stages hinders us from reaching zero defects.

Defects in the software craft. To improve software quality, we need software engineering approaches that cope with such problems inherent in the development process as the following:

- *Requirements.* It is difficult to validate product requirements. Developers and customers may achieve some consensus (e.g., via user reviews of requirements), but without high confidence because their reviews are based on prose specifications that cannot reveal system aspects like performance, function, usability, and reliability. We cannot demonstrate for certain that a proposed system meets all customer requirements until we integrate all implemented functions late in the test phase. At that time, problems are costly to detect and correct. The cycle time required to develop products exceeds the span over which we can accurately forecast customer needs. Requirements, even if we can validate them at product conception, may become obsolete before we ship the product. A major reason for the lengthy product cycle time is that, as an industry, we reinvent most new software and seldom make use of design solutions that may already exist.

- *Architecture.* We have commonly defined architecture using ambiguous natural language, diagrams, and other free-form notations. Such expression hinders our ability to communicate accurately the system's structure and prevents us from formally analyzing the structure and dynamic behavior of the system. Thus we design and implement functions based on structures and protocols that are weakly specified, poorly communicated, and not formally validated during design. We rely on exhaustive testing to correct problems and produce high-quality products.

  We are unable to test the feasibility of our initial architecture ideas or compare alternative proposals. We are unable to examine the architecture specification and determine the effect that architecture tradeoffs and function placement decisions have on system performance, usability, and reliability. To explore these aspects, we must either create expensive, throw-away models of the system or wait until we integrate the implemented functions late in the test cycle. Costs usually dictate that few, if any, alternative designs are considered. Poor architecture decisions can propagate through all stages of a project and cause costly rework to undo design and implementation based on those decisions. We have no mechanism to validate that a proposed product architecture satisfies all known product requirements.

We are unable to test the feasibility of our initial architecture ideas or compare alternative proposals.

- *Design.* We usually custom-design a new system without taking maximum advantage of existing design solutions. We maintain no library of proved, common software subassemblies from which to build systems, as do designers in other disciplines. Subassemblies such as macros and subroutines developed on one project are generally not designed, preserved, or ported for reuse on another project. Although structured design languages have improved software quality, the use of multiple design notations on the same project causes errors when we refine a design from one stage and notation to the next. We have no mechanism to detect all of these errors early in the design phase, where they are much less costly to correct. We have no mechanism to prove that a design is correct. We base our confidence that a design works on a visual inspection of a sample subset of possible design paths and input-output permutations.

  Good design principles[13–16] are hard to enforce across large projects. Attempts to partition design

into mutually exclusive, self-contained modules are thwarted if we allow the modules to access global data, such as control blocks, and thus hap-

Aircraft and automotive designers use CAD/CAM to graphically define the architecture of their products.

hazardly affect system state variables. This causes unpredictable system behavior that we are unable to test.

- *Implementation and test.* As in design, we have no mechanism to prove that a program is correct, complete, or consistent with the design. We have no reliable metrics with which to examine code properties and predict system quality, rejecting poor code on the basis of excess complexity or poor readability, structure, or maintainability. As in design, we base our confidence that an implementation works on tests of a sample subset of possible design paths and input-output permutations. It is difficult to prove that the system works in all specified situations and system states, so we rely on exhaustive testing.

Although these are not a complete classification of software development problems, they provide ample search space for software engineering solutions. The most glaring deficiency in the software craft, from our perspective, is a lack of mechanisms for the precise specification and machine analysis of the statement of the problem (requirements) and early design solution (architecture), since all further development activity and communication proceed from these statements. It might be instructive to consider how other engineering disciplines have approached these problems.

Snapshots from the tasks of designers in other engineering disciplines provide some strategies that software developers could use to correct defects in their craft. Aircraft and automotive designers use computer-aided design and manufacturing (CAD/CAM) facilities to graphically define the architecture of their products. Data bases hooked to these facilities capture the static, hierarchical structure of the product. The designers can graphically query these data bases to provide visual, static analysis of the relationship of design elements to the total structure. These designers build quick, inexpensive prototypes that perform the same external functions as will the final product. They subject these prototypes to logical "wind-tunnel" testing (i.e., simulation) to analyze a product's dyamic behavior in stress conditions and real-life environments. They create multiple designs and choose the best alternative, based upon prototype results.

Hardware logic designers use design logic simulation techniques to test their designs. The designer can examine the functional design model for correct logic and timing relationships and physical function placement prior to committing the design to implementation. Hardware designers can capture the design specifications (Boolean logic rules) for logic elements in a data base for later reuse. These reusable elements can be included in future designs where appropriate, thereby saving specification time.

These disciplines provide their designers with mechanisms to specify design precisely, to prototype the external functions of a product, and to statically and dynamically analyze the system design of the product's behavior. We call these capabilities *architecture prototyping functions* and explore how they can be realized in a software engineering environment.

Facilities for a software engineering environment

In the discussion of the software development craft, we have purposely concentrated on the problems that hinder us from producing zero-defect software. Software research over the past decade has developed engineering techniques that have begun to invade software practice. Some of these techniques provide kernels around which we can construct an improved software development scheme.

- Machine-analyzable Very-High-Level software specification Languages (VHLLS)[17–20] and Entity/Relationship data models[21,22] offer mechanisms for the organization of enterprise analysis information, for the precise specification and analysis of software requirements and architecture,[23–26] and for rapid prototyping of product function.[27,28]
- Graphic representations of software system design structure[29–32] provide user-oriented interfaces for the definition and manipulation of system design.

- Rigorous software development approaches,[4,16,33–35] based on abstract data types, on algebraic specification of design processes in terms of their operations on data abstractions, and on proof of design and programs, provide a scheme in which reusable software subassemblies may be developed and reused.

Product descriptions expressed in procedural VHLLs can be symbolically executed to verify product behavior.

In the next section, we provide an overview of these concepts and explore how they can support architecture prototyping capabilities.

Very-High-Level software specification Languages (VHLLs). High-Level programming Languages (HLLs) like ALGOL, PL/I, and Pascal give programmers the freedom to express algorithms and data structures without the bookkeeping of machine details. Pseudo-code notations abstracted from these languages allow designers to express low-level design of procedures without regard for the implementation characteristics of data structures.

A class of notations called Very-High-Level software specification Languages (VHLLs) permits software planners and architects to formally define abstract problem statements and early design solutions. This class includes abstract procedural languages for expressing design control and data flow,[17,19,28] nonprocedural languages for representing system objects and relationships,[18,31] and algebraic design notations.[34,35]

Many of these languages share the capability of expressing operations on a particular abstract representation of data—the Entity/Relationship (E/R) or relational model.

Product descriptions expressed in procedural VHLLs can be symbolically executed to verify product behavior. System definitions captured in a machine-processable E/R data base can be statically analyzed for consistency and completeness.

We examine two members of the VHLL class, the Problem Statement Language/Problem Statement Analyzer (PSL/PSA) and the Functional Specification Tools (FST) to explore how they provide these capabilities.

Problem Statement Language/Problem Statement Analyzer. PSL/PSA, originally developed at the University of Michigan,[18] is a system that aids in the precise definition of system specifications. These specifications may include problem statements (requirements) and the framework of design solutions (system structure). PSL/PSA is composed of two components, the Problem Statement Language (PSL) and the Problem Statement Analyzer (PSA). PSL is a nonprocedural language that includes constructs for describing software systems and facilities for capturing the resultant system model in an E/R data base. PSA operates on this model, and provides report generation and data base query capabilities to allow the designer to inspect the model and statically analyze model definitions for consistency and completeness.

Defining a system in PSL involves mapping that system's objects and relationships (ordered associations of two or more objects) into the entities (e.g., PROCESS, PROCESSOR, EVENT, SET, INPUT, and OUTPUT) and relationships (e.g., GENERATES, PERFORMS, INTERRUPTS, and COLLECTION OF) of a conceptual PSL E/R model. These objects and relationships are captured and stored in the PSL data base in a network model that describes the dependency and interaction among objects in the target system.

Since this network exists in a machine-analyzable data base, the PSA report-generation facilities can be used to statically analyze it for consistency and completeness. We examine later how E/R-based specification techniques like PSL/PSA can be used for static analysis of architecture descriptions.

Functional Specification Tools. The Functional Specification Tools (FST),[17] developed at IBM, comprise a system that aids the designer in expressing system architecture and dynamic system design behavior. FST contains machine-analyzable languages for the following three purposes: (1) for expressing a system's control structure as a network of asynchronously communicating state-machine processes; (2) for defining sequential procedures invoked from

processes in this control structure; and (3) for organizing system data in a relational model on which the procedures operate. FST also allows the designer to describe in the relational model the target environment in which the architecture processes will execute. The designer can bind the architecture specification and its execution environment together into a functional model that can be symbolically executed to exhibit the same behavior as the proposed system. The designer can then exercise this model using the discrete event driven simulation facilities provided by FST to dynamically analyze architecture behavior in various test configurations.

We examine later how VHLLs like FST can be used for dynamic behavior analysis of architecture and to accomplish rapid external function prototyping.

Graphic representations of system design. Just as building architects study blueprints, software designers need to examine graphics representations of system design to understand the hierarchical structure of their creation and the relationship of parts of the system to the whole structure. Much of the work done on graphics representations of programming, such as GREENPRINT,[32] has dealt with diagramming the internal text of program- and detailed-design modules. More appropriate to the task of system design are the graphics documentation facilities of the Structured Analysis and Design Technique (SADT),[31] which give a view of the hierarchical structure and component interfaces of a system produced by means of SADT decomposition techniques.

Whereas graphic documentation facilities enhance the design process, the design task that would benefit most from graphics techniques is the architecture creation and manipulation task. Analysis of designers' tasks in various disciplines shows that much of the early definition of system structure, entities, and relationships is done graphically on a blackboard, scratchpad, or CAD/CAM facility when available.

A very useful facility for software design would be one that allows hierarchical decomposition and graphics definition of system structure and control flow at the terminal and captures these definitions in an E/R data base and computational network of processes for later analysis. A partial example of one such facility used in IBM connects the TELL system[31] to PSL/PSA.

A general-purpose graphics facility for software design and development would merge problem statement, system design, element design, and program representations with their associated text and data into an integrated hierarchical view of the product.

Data abstractions and formal specification approaches. The discussion about defects in the software design craft identified three major deterrents to design productivity and quality. When there is no library of reliable software subassemblies available, we develop a new system from the ground up. Also, we have no mechanism to prove that a design is correct. The control-block global data design approach produces design functions that are highly interdependent and behave unpredictably when states in remote corners of the system change.

Algebraic techniques provide a mechanism for specifying the external behavior of a module or data object.

These three effects are related in that the design approach creates defects, produces a haphazardly coupled system whose primitives are difficult to isolate and reuse, and creates a complex network of logical relationships that cannot be easily reduced to provable conditions.

New design techniques[4,16,33-35] have emerged over the past decade that are based upon data abstractions, on formal specification of modules that operate on these abstractions, and on formal mathematical proof of the correctness of these modules. In brief, these methods provide a mechanism for rigorously specifying the external behavior of a software module that encapsulates an abstract data object. The object may be any abstract entity—such as a queue—in the design universe that we wish to describe. This data object may be viewed as a state machine in that operations on the object (e.g., ENQUEUE and DEQUEUE) change the condition (state) of the object (e.g., QUEUE_EMPTY, QUEUE_FULL, QUEUE_HAS_ELEMENTS). The module manages this

object and its states and provides primitives that can be invoked by external users to change or observe the states of the object (e.g., IS_QUEUE_EMPTY). The implementation details of the object are hidden from users in the module, and the external behavior of the module is specified to users in terms of the allowed primitives, so that the module may be isolated and transported to other designs in need of its specified behavior.

The behavior of these abstract objects and their primitives may be formally specified. Using first-order predicate calculus,[34,35] *assertions* (or *invariants*) can be made about the values of variables and the truth of conditions at various execution points. The operation of the module may be proved mathematically correct with respect to these invariants.[33]

We can envision a design environment in which these data-abstraction-based design modules are preserved and made available for inclusion in higher-level designs, which are in turn defined as modules encapsulating higher-level data abstractions. Such an environment would be conducive to a bottom-up creation of reusable software subassemblies.

Other researchers[9] have proposed merging some of the facilities just described into an integrated software development environment. In the remainder of this paper, we examine how to exploit this technology to provide architecture prototyping facilities in this environment.

Architecture prototyping in the software engineering environment

Architecture prototyping can be viewed as the phase of software development at which, having stated the user's problem and the idea for a product solution (requirements), we seek to define that system's global structure and behavior and quickly test whether that idea is feasible.

In the large, complex software systems typically produced in industrial software environments, the architecture "idea" must be described at a level of abstraction far removed from any detailed knowledge of design elements. Architects must describe the behavior and relationships among things like network nodes, or operating system components, rather than among modules. Approaches that rely on semantic description and interconnection of design elements[28,36,37] will not provide the economy and expression set required for system design. Architecture prototyping requires a system-level description of a product's behavior.

The testing of architecture feasibility is crucial to developing quality software. Without testing, ambiguous requirements, faulty design decisions, and difficult-to-use interfaces may propagate through the rest of a design until expensive design iterations, implementation, and verification finally uncover the problems they create.

We have seen that other engineering disciplines provide designers with facilities for architecture prototyping. Until recently, software architects had no facilities for testing architecture ideas. Very-High-Level Languages and Entity-Relationship-based formal specification techniques exist today and provide software designers with the tools and environments necessary to support software architecture prototyping. Software can be designed from new perspectives using such techniques.

Structural aspects of an architecture can be verified using static analysis techniques.

Specification. Instead of expressing system design in natural language, the architect can begin design by describing the static structure of a system in terms of the entities and relationships in that structure. Graphics terminal facilities, as in Reference 30, can be useful for decomposing the initial system structure into substructures and for disconnecting, moving, and reconnecting components until the structure matches the designer's mental image. Entity and relationship data can be parsed from this graphics input and can be captured on an E/R data base like PSL. This information network can be replayed graphically to the architect at later design sessions during which the architect can decompose structures further, can annotate elements of the framework with design attributes, and can build successively more detailed descriptions of the static structure and process relationships.

When satisfied with the system structure, the architect can then define the dynamic execution behavior of this structure. Using the graphics replay of the design network model, the designer could pick regions in the network and define the interprocess control logic, state transitions, and behavior rules using a VHLL notation. If the notation used is well-defined and executable (e.g., the state-machine process model of FST), the cumulative static structure information, process relationships, and embedded process descriptions can be compiled into the VHLL environment to create a symbolically executable functional model. Two related representations (E/R and VHLL) of the architecture are then available for static and dynamic analysis, respectively.

Static analysis of system structure. Once defined and visually inspected, structural aspects of an architecture can be verified using static analysis techniques. Static analysis can be used for the following purposes: (1) to verify the *completeness* of a description, by testing whether all entities, relationships, and attributes in an architecture have been described, and (2) to verify the *consistency* of entities in a description to the total structure of an architecture, for example, by testing whether related components describe their interface as complementary, that is, as mirror images.

PSL/PSA provides for both completeness and consistency checking. PSA provides report generation facilities such as structure reports (process hierarchy), process chain reports (procedure invocation sequences through the system), and data/activity reports (cross reference of data used by processes). Each of these reports alerts the designer to undefined or inconsistently specified entities or relationships.

To maximize the usability of architecture prototyping functions, the architect should be capable of probing regions of the architecture graphically and receiving pictorial notification of PSA-generated inconsistencies and incompleteness.

Dynamic analysis of system behavior. The architect can dynamically analyze the run-time behavior of an architecture by binding together into a functional model the static structure data, process relationships, and imbedded process design descriptions with information on the execution environment, and then by symbolically executing this model of the design.

There are several approaches to the symbolic execution of functional models. In our scheme, the FST state-machine process descriptions that make up the design specification are compiled to define the executable model of the system. General-purpose simulation languages (e.g., SIMULA and GPSS) provide a similar scheme for modeling systems. References 26 and 38 explore other variations of this approach. Other approaches to dynamic analysis, which we do not explore here, include symbolically executing algebraic specifications[25,39] and analyzing graph-like models of design behavior.[23]

The architect can exercise the functional model using discrete event driven simulation techniques.

Once the functional model is established in its test configuration and environment, the architect can exercise it using discrete event driven simulation techniques. The architecture can send an *event* to activate a process in the architecture and then monitor a simulation trace to follow design rules that are traversed, interprocess communication, and interleaving that occurs when a process is interrupted in favor of a higher-priority process. Functional simulation allows the architect to examine architecture control and data flow, and to detect deadlock, resource contention, timing, and interface problems. The architect can examine the performance of the system under stress conditions. Aspects of the environment can be modified around the functional model to examine system behavior under adverse conditions.

Advanced architecture prototyping facilities should allow the architect to select regions of the architecture graphically and watch their run-time behavior by way of animation techniques.

Rapid prototyping. Rapid prototyping[25-28,39,40] is the process of building a quick, inexpensive model that exhibits the same behavior as will the final system. The purpose of rapid prototyping is to provide feed-

back to planners and designers on the suitability of system functions. Rapid prototyping can be used to analyze a variety of system attributes, including usability and performance. Prototypes can be implemented in a wide variety of techniques, ranging from symbolic execution of algebraic design specifications[25,39] to direct execution of disposable interim versions of a product implementation.

We envision a future software engineering environment in which design and code reuse is a regular procedure.

A rapid prototype in the architecture prototyping environment we are discussing deals only with the usability of the externals of the product. Other attributes like system performance and control flow are examined using functional modeling and dynamic analysis techniques. In fact, an externals prototype in this environment would be implemented as an extension of the functional model. With the facilities already described, the designer would bind to each functional process in the architecture a description of the external interface of that component. Such a description can be expressed as a sequence of screens or menus defined in a screen-programming language.[41,42] When exercised within the VHLL symbolic execution environment, the prototype behaves externally like the final product. The prototype also permits potential users of the product to invoke and interact with facsimiles of product functions, allowing them to provide feedback on the usability of product interfaces. It could also be used to clarify ambiguities about customer requirements.

When implemented as an extension of the functional model, the protoype need not be discarded. The external interfaces can be preserved with the architecture and used iteratively as the design is expanded within the architecture framework to validate that the element design does not corrupt user interfaces or introduce delays in response times. The screen definitions from stabilized interfaces can be preserved and later coupled with code translated from the functional design to produce the final product implementation.

Fitting software subassemblies into the framework. A simplified view of software development is that it involves the two problem-solving phases of requirements and architecture to create a solution framework. That is followed by tedious iterative refinement and verification to fill in the details and produce a translation that runs on a machine.

During design refinement, programmers sometimes find existing implementation routines that fit in their design structure and solve a particular design task. They take advantage of these routines and modify and reuse them rather than writing new ones. Thus they avoid some of the refinement and detailed specification process. In general, reuse occurs haphazardly; we do not facilitate the reuse of primitives in the design of systems, nor do we broadcast their availability in design environments. The Unix[43] environment is a notable exception.

We can envision a future software engineering environment in which design and code reuse is a regular procedure. Such an environment would include teams of designers building bottom-up layers of reliable, reusable primitives. These primitives could be algebraically specified and their reliability formally proved. Packages of these primitives could conceivably be reused by designers seeking a subassembly to fit in an architecture framework previously decomposed in a top-down manner. An architecture prototyping facility might become the workbench at which these elements would be bound together for feasibility testing.

Research is necessary to identify frequently used primitives and data abstractions and to provide mechanisms to select and fit appropriate subassembly solutions into the validated architecture framework at the point where the need for a solution has been identified but not elaborated. Such solutions could include a reusable unit of code, a complex hierarchy of design primitives, or an incomplete design stub that itself invokes design functions whose description will be elaborated in later design stages. An architecture prototyping facility should support the binding of framework and solution and the evaluation of the cumulative result using static and dynamic analysis techniques.

Maintenance and enhancement. Developers of software have frequently ignored what may be the largest group of software users: those who debug, maintain, and enhance existing systems. Current software development documentation hampers maintenance and enhancement in that it provides neither comprehensive views of the system structure nor mechanisms to descend hierarchically through the framework to view a function where a problem may exist or where an enhancement must fit. Architecture prototyping facilities would ease this problem by providing an integrated development repository from which the specification captured graphically and lexically during design could be replayed for those who would maintain and enhance the product. The ability to view the relationships of parts of the product to the whole and to design and test changes to the product quickly, using the abstract notations and dynamic analysis facilities, could greatly improve maintenance and enhancement productivity and quality.

Architecture prototyping facilities would provide an integrated development repository.

Toward a process for reliable software development

The methods discussed in this paper can be integrated into a *top-down verification* approach to developing software. A top-down verification methodology is illustrated in Figure 3 and features the following new tasks:

- Formal specification of user requirements, system structure, and dynamic system behavior using VHLLS.
- Rapid prototyping of external functions and user interfaces.
- Static analysis of user-system relationships and system structure for consistency and completeness.
- Dynamic analysis of system control and data flow to verify dynamic system design behavior and to model system performance and reliability.
- Selection and insertion of existing software subassemblies into the system design framework.
- Algebraic specification, proof, and insertion of new design as needed.
- Static and dynamic analysis of the cumulative product description at each stage.
- Potential automated code generation from the verified product description.[36]
- Optimization of generated code for the target environment.

A top-down verification approach would improve software development in several ways. First, it would allow the analysis of early product descriptions for feasibility as soon as they are produced, rather than waiting until the product has been designed, implemented, integrated, and has become available for testing.

Major interfaces, which are created first, would be tested first. Potential interface errors would be detected, resolved, and reworked before any further development that depends on these interfaces begins. Error rework cost would be reduced.

Design feasibility would be demonstrated early, before costly implementation, through static and dynamic analysis of the system structure. The consequences of design decisions for system performance, reliability, and usability could be explored and verified.

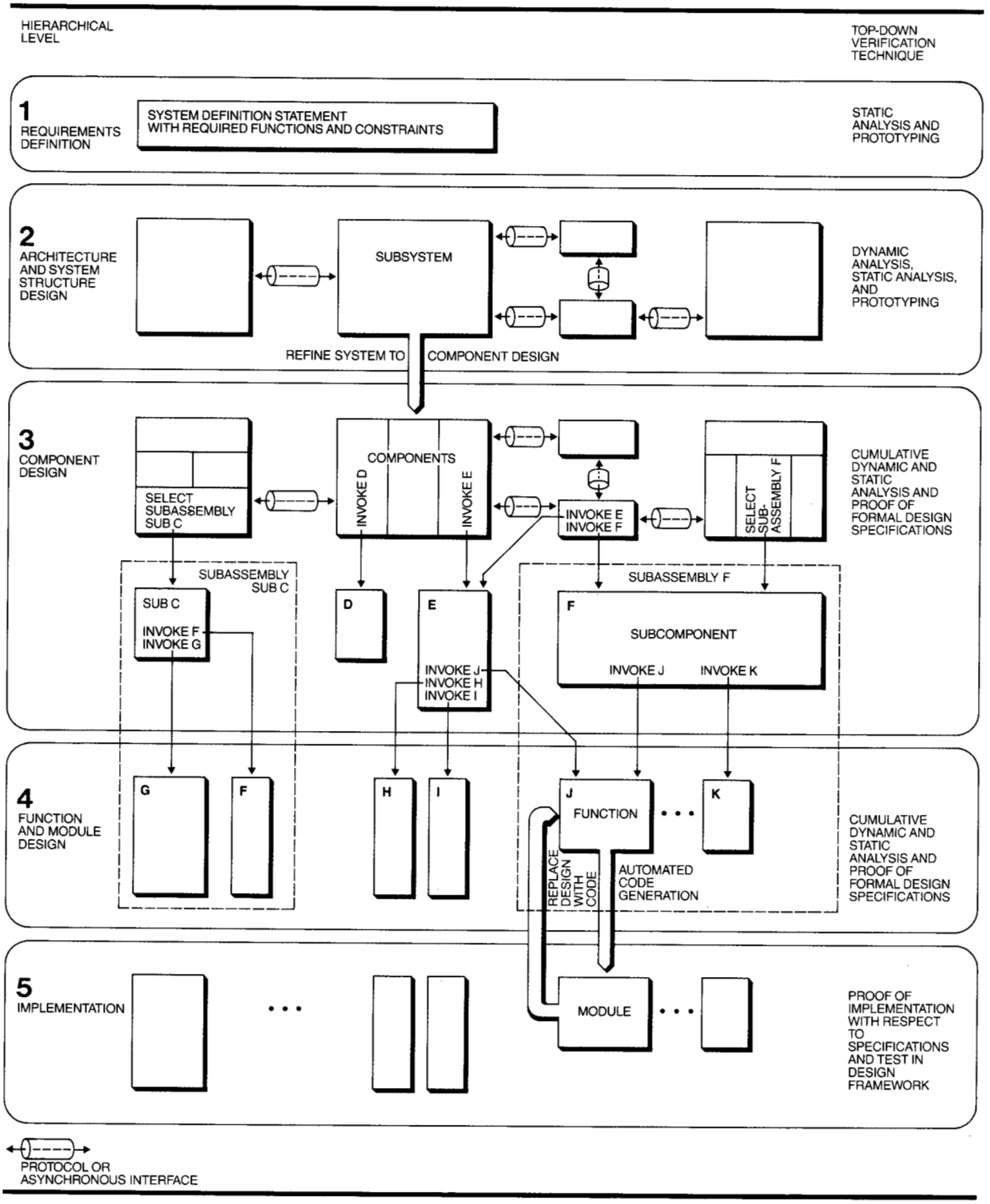
Design analysis would be automated and would be reproducible. It would not be subject to human variability, as are design and code inspections. Formal design verification could be accomplished by correctness proof.

Errors could most often be prevented, or at least detected when produced, if the formal specification methods used constantly enforced coherence checking and allowed symbolic execution.

Productivity would be improved through the reuse of existing reliable design and implementation solutions.

A top-down verification software engineering approach would create new tasks and roles. Planners would reveal prototypes to customers to validate requirements. Architects would statically and dy-

Figure 3 Top-down verification in the software development process



namically analyze their system designs prior to committing the system to production. Implementers would optimize subassemblies. Such an approach shows promise for solving some of the problems inherent in the software craft and in the engineering of reliable software.

Concluding remarks

We have explored the software craft and found that, although it is capable of producing high-quality products, more precision and automation are needed to produce zero-defect products with reasonable cost.

We have also considered more rigorous software development approaches based on the use of emerging software engineering methods like formal and algebraic specification techniques, Very-High-Level Languages, data abstractions, and static and dynamic design analysis. These tools give us more precision and control in describing and communicating design, and they permit early machine analysis of specifications.

We have discussed the idea that the development of quality software requires a phase of architecture prototyping in which initial design ideas are "wind-tunnel tested" for feasibility. We propose that the task requires an automated facility that has the following features:

- A graphics facility for the creation and analysis of system design.
- A common machine-analyzable abstract notation for defining structure attributes, for expressing dynamic architecture behavior, and for defining design refinement within the architecture framework.
- An Entity-Relationship data base on which to capture structure information for static analysis and design communication.
- A VHLL design computational model and symbolic execution environment in which to capture interprocess control and data flow specifications for dynamic analysis.
- Static analysis tools to test structural completeness and consistency.
- Dynamic analysis tools that allow symbolic execution of architecture to facilitate the prototyping of external interfaces and behavioral analysis of internal system control and data flow.
- The capability to fit further design refinement or common design and code subassemblies into the architecture framework and to evaluate the cumulative result, using static and dynamic analysis techniques.
- The use of the product data base and symbolically executable specification to aid in maintenance and enhancement.

Many of these facilities exist and have begun to be used independently today. We propose that these facilities belong in any future integrated software development environment. Further work should explore the issues of making these tools usable as partners in the design process by way of design knowledge bases and expert system interfaces. Once ready for software production, these facilities will support a top-down verification approach to software development, which shows promise for producing highly reliable software.

Acknowledgments

I thank Dr. Lip Lim, Ron Radice, Kent McCaulley, Harvey Hallman, Bill Brown, Michel Berthaud, Michel Frenkiel, Jim Miller, Gary Deen, Carol Greenstreet, Judi Powers, Jean-Pierre Augias, Philippe Potin, Dick Phillips, Bill Ciarfella, Robert Mays, Gene Hoffnagle, Mike Fagan, Linda Mason, Al O'Hara, Tim McMurray, Don Daria, and Dr. Louise Neilsen, all co-workers in IBM who have shared their ideas on architecture and software engineering practice.

Cited references

1. M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal* **15,** No. 3, 182–211 (1976).
2. O. R. Kohli, *High-Level Design Inspection Specification,* Technical Report 21.601, IBM Corporation, Kingston, NY 12401 (1975).
3. O. R. Kohli and R. A. Radice, *Low-Level Design Inspection Specification,* Technical Report 21.629, IBM Corporation, Kingston, NY 12401 (1976).
4. H. D. Mills, D. O'Neill, R. C. Linger, M. Dyer, and R. E. Quinnan, "The management of software engineering," *IBM Systems Journal* **19,** No. 4, 414–477 (1980).
5. N. Wirth, "Program development by stepwise refinement," *Communications of the ACM* **14,** No. 4, 221–227 (1971).
6. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming,* Academic Press, Inc., London (1972).
7. H. D. Mills, "Top down programming in large systems," *Debugging Techniques in Large Systems,* Prentice-Hall, Inc., Englewood Cliffs, NJ (1971).
8. R. C. Linger, H. D. Mills, and B. L. Witt, *Structured Programming: Theory and Practice,* Addison-Wesley Publishing Co., Reading, MA (1979).
9. P. S. Newman, "Towards an integrated development environment," *IBM Systems Journal* **21,** No. 1, 81–107 (1982).
10. G. B. Davis, "Strategies for information requirements definition," *IBM Systems Journal* **21,** No. 1, 4–30 (1982).

11. J. F. Stay, "HIPO and integrated program design," *IBM Systems Journal* **15,** No. 2, 143–154 (1976).
12. R. J. Pearsall, "Technique for assessing external design of software," *IBM Systems Journal* **21,** No. 2, 211–219 (1982).
13. W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal* **13,** No. 2, 115–139 (1974).
14. G. J. Myers, *Reliable Software Through Composite Design,* Petrocelli/Charter, New York (1975).
15. M. Jackson, *Principles of Program Design,* Academic Press, Inc., New York (1975).
16. D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM* **15,** No. 12, 1053–1058 (1972).
17. 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.
18. D. Teicherow 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* **3,** No. 1, 41–48 (1977).
19. R. B. K. Dewar, "The NYU ADA translator and interpretor," *Proceedings of the IEEE's Fourth International Computer Software and Applications Conference,* CH1607-1 (1980), pp. 59–65.
20. P. Wegner, "The Vienna definition language," *ACM Computing Surveys* **4,** No. 1, 5–63 (1972).
21. P. P.-S. Chen, "The Entity-Relationship model—Toward a unified view of data," *ACM Transactions on Database Systems* **1,** No. 1, 9–36 (1976).
22. E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM* **13,** No. 6, 377–397 (1970).
23. G. Estrin, "A methodology for design on digital systems supported by SARA at the age of one," *AFIPS Conference Proceedings, National Computer Conference* **47,** 313–321 (1978).
24. E. McCoy, *An ADA Language Model of the AN/SPY-1A Component of the AEGIS Weapon System,* Report NPS52-80-011, Naval Postgraduate School, Monterey, CA (1981).
25. D. Cohen, W. Swartout, and R. Balzer, "Using symbolic execution to characterize behavior," *ACM SIGSOFT Software Engineering Notes* **7,** No. 5, 25–32 (1982).
26. A. M. Stavely, "Models as executable designs," *ACM SIGSOFT Software Engineering Notes* **7,** No. 5, 167–168 (1982).
27. A. G. Duncan, "Prototyping in ADA: A case study," *ACM SIGSOFT Software Engineering Notes* **7,** No. 5, 54–60 (1982).
28. R. T. Mittermeir, "HIBOL: A language for fast prototyping in data processing environments," *ACM SIGSOFT Software Engineering Notes* **7,** No. 5,133–140 (1982).
29. S. N. Zilles and P. G. Hebalkar, "Graphic representation and analysis of information systems design," *Data Base* **11,** No. 3, 93–98 (1980).
30. P. Hebalkar and S. N. Zilles, *TELL: A System for Graphically Representing Software Designs,* Research Report RJ-2351, IBM Research Laboratory, San Jose, CA 95193 (1978).
31. D. T. Ross and K. E. Schoman, "Structured analysis for requirements definition," *IEEE Transactions on Software Engineering* **3,** No. 1, 6–15 (1977).
32. L. A. Belady, C. J. Evangelisti, and L. R. Power, "GREENPRINT: A graphic representation of structured programs," *IBM Systems Journal* **9,** No. 4, 542–553 (1980).
33. C. B. Jones, *Software Development: A Rigorous Approach,* Prentice-Hall International, Inc., London (1980).
34. B. Liskov and S. N. Zilles, "An introduction to formal specifications of data abstractions," *Current Trends in Programming Methodology* **1,** Prentice-Hall, Inc., Englewood Cliffs, NJ (1977).
35. D. L. Parnas, "A technique for software module specification with examples," *Communications of the ACM* **15,** No. 5, 330–336 (1972).
36. J. L. Archibald, B. M. Leavenworth, and L. R. Power, "Abstract design and program translator: New tools for software design," *IBM Systems Journal* **22,** No. 3, 170–187 (1983).
37. J. Archibald, *The External Structure: Experience with an Automated Module Interconnection Language,* Research Report RC-8652, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1981).
38. R. M. Bryant, "Discrete system simulation in ADA," *Simulation* **39,** No. 4, 111–122 (1982).
39. J. K. Dixon, J. McLean, and D. L. Parnas, "Rapid prototyping by means of abstract module specifications written as trace axioms," *ACM SIGSOFT Software Engineering Notes* **7,** No. 5, 45–49 (1982).
40. M. Zelkowitz and M. Branstad, editors, "Working papers from the ACM SIGSOFT Rapid Prototyping Workshop," *ACM SIGSOFT Software Engineering Notes* **7,** No. 3, 14–15 (1982).
41. M. E. Maurer, "Full screen testing of interactive applications," *IBM Systems Journal* **22,** No. 3, 246–261 (1983).
42. *Display Input/Output Facility (IOS3270) User's Guide,* IFP 5785-HAA, IBM Corporation; available through IBM branch offices.
43. J. Mashey, "Unix and the programmer's workbench," *Proceedings of the IBM Worldwide Machine Usable Programming Productivity Tools Symposium,* IBM Corporation, San Jose, CA 95193 (1982).

William E. Beregi *IBM Data Systems Division, Kingston, New York 12401.* After graduating from Carnegie-Mellon University with a B.S. in mathematics in 1974, Mr. Beregi joined IBM at Kingston, New York. He now works there as a development programmer in data systems assurance, managing the Software Engineering Process Technology group. The group's mission is to study and recommend changes to IBM software products and development processes to improve system quality. This group is investigating design languages, tools, and software engineering methodologies that may eventually be used to automate software development. Mr. Beregi also holds an M.S. in computer and information science from Syracuse University, and he is pursuing a Ph.D. there.

Reprint Order No. G321-5206.