F. D. Smith
C. H. West

Technologies for Network Architecture and Implementation

Systems Network Architecture (SNA) provides a framework for constructing networks of distributed processors and terminals. This paper discusses some of the fundamental properties of network architectures such as SNA, and the evolution of formal descriptive methods that provide precise, complete definitions of the architecture. This has culminated in the development of a programming language, Format and Protocol Language (FAPL), tailored for programming a reference model or meta-implementation of an SNA node. In this form, the architecture specification is itself machine-executable. This property has led to new software technologies that improve quality and productivity in the processes for developing a network architecture and the product implementations derived from it. Automated protocol validation provides the tool necessary to ensure a correct and internally consistent definition of the architecture. This definition can then be used as a standard for testing products to determine compliance with the architecture. Direct implementation of network software by compiling the meta-implementation program is another emerging technology. This paper reviews the current state of work in these areas.

Introduction

The continuing evolution of computer networks has made it necessary to precisely specify the message formats and protocols that define the services provided by the network. In IBM, Systems Network Architecture (SNA) has evolved since its announcement in 1974 to encompass functions for controlling large-scale networks of distributed processors and terminals [1, 2]. Similarly, the definition of the architecture has evolved from a prose description to a formal, state-oriented, machine-executable version written in a high-level programming language. This has proven to have many advantages. An architecture defined in a programming language has an unambiguous interpretation when it is executed. It also provides a basis for a number of tools that help improve the quality of the architecture itself and of the products that are derived from it.

In this paper we discuss the nature of network architecture definitions and trace the development of a machine-executable definition of SNA. We then discuss some interesting uses of the executable definition. The first use is for the automated validation of the architecture. For SNA a correct definition is ensured in two ways: 1) frequent phase reviews and intensive inspection steps in the architecture development process, and 2) automated protocol validation. The reviews and inspections cover issues of functional correctness and completeness; the automated validation methods detect errors related to the internal consistency of the detailed design. Both methods are required to identify errors in the architecture before their more costly discovery during the implementation of products that conform to the architecture. Further discussion of the techniques used to control development of the evolving SNA can be found in Ref. [2].

Finally we discuss two uses of the executable architecture definition that are shaping advances in software technology for implementing network software. These are the implementation of products directly from the architecture definition (by a compilation process) and the development of tools for testing products to determine if they comply with the architecture.

Definition of SNA

One of the most fruitful ideas in the design of computer systems has been Blaauw's [3] distinction among architecture, implementation, and realization. The first computer-related use of the term "architecture" to describe system design came out of Project Stretch [4]. In the context of processor hardware, *architecture* is the specification of the

system's functions and the ways in which a user invokes those functions. It encompasses data types, addressing, instructions, I/O, etc., as elements of a language used by the programmer to invoke functions and combine them in useful sequences.

A computer *implementation* defines the logical structure of a machine which interprets this language and performs the architecture. Some elements of a computer implementation are internal registers, memory, adder, and controls. Electrical and mechanical design of the physical components from which the implementation is built is the realm of *realization.* The goal of the architect is to provide for the programming needs of the user, whereas cost, performance, reliability, and manufacturability are major issues for those producing the implementations and realizations.

Many implementations are possible for an architecture, each having different cost/performance objectives. For example, the several models of the IBM 308X, 303X, and 43XX systems are all implementations of System/370 architecture [5] with a wide range of cost and performance. Just as there may be many implementations of the same architecture, there may be multiple realizations of the same implementation (e.g., the same design in a newer technology).

In the context of software and, more specifically, computer network software, the term "architecture" has a richer meaning. Here the architect's goal is to provide product interconnections through a network that supports general end-user to end-user communication. An end-user is typically a person working at a terminal or a program that provides some service. Not only must network architectures specify what functions are performed and how to invoke them, they must also provide for harmonious communication among thousands of largely independent elements in a computer network. The essential role of network architects, then, is to specify and enforce a set of rules for communication to which all product designs conform. Conformance is necessary to avoid a proliferation of product-specific decisions that would cause the costs of providing interconnections to increase dramatically.

The key problems are *representation* and *synchronization.* Messages flowing in a computer network contain addressing and control information that must be interpreted at each node in the paths to intended destinations. Architectural specification of formats for messages is necessary for consistent interpretation of this information by each product. This part of the design is fundamental but relatively straightforward; its problems (allocation and encoding) are much the same as those related to computer architecture for data-unit representation and instruction formats.

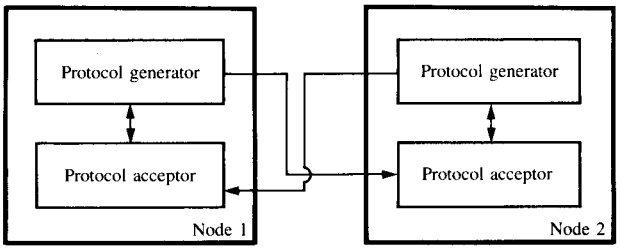
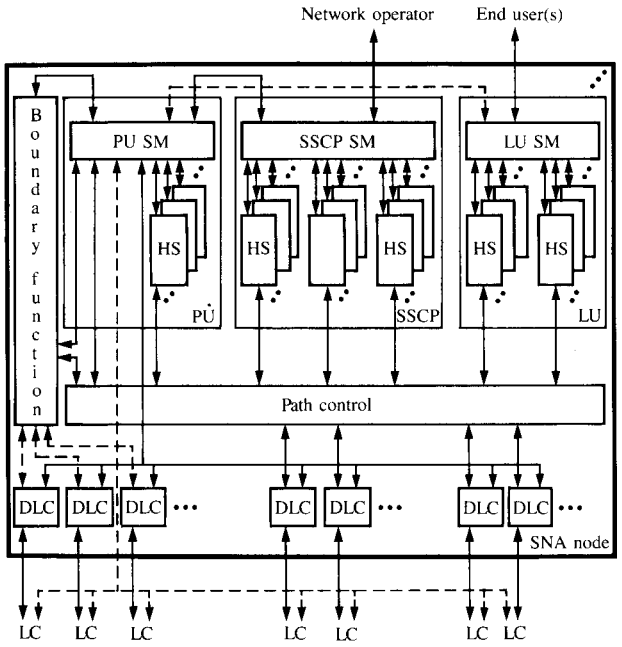


Figure 1 Interconnected protocol machines.

The hard problem is synchronization. In almost every computer system there exists some form of concurrent operation that exploits the natural concurrency among tasks. In many cases (e.g., concurrent process execution in operating systems) the processes have access to shared storage, and thus a single copy of state information (e.g., a semaphore) that can be updated quickly and reliably is sufficient for synchronization. In computer networks, the processes that must be synchronized do not, in general, share any form of durable storage because they are located in independent network nodes. They must rely on exchanging state information through a transmission medium that is subject to errors and unpredictable delays. The rules devised to govern the content and sequencing of exchanged state information are the *protocols.*

Systems Network Architecture is divided into layers and processes following well-known principles of decomposition and modularization in the design of large-scale systems. Gray provides an overview of these components and their synchronization requirements in Ref. [6]. In principle, each protocol that establishes and maintains synchronization among a collection of processes can be specified by an enumeration of all possible sequences of message exchanges. If the protocols operate in an error-free environment, the set of possible sequences is finite but very large, especially in a functionally rich architecture such as SNA. Since the transmission medium is subject to random errors and unpredictable delays that alter the protocol sequences, the set is at best countably infinite; enumeration of sequences in any manner useful to product builders is impossible.

Each protocol can, however, be modeled by abstract machines called *protocol machines* that generate valid output sequences in response to received inputs. To model a two-party protocol, a pair of machines, one in each node, is used. Further, each machine may be decomposed into generator (sender) and acceptor (receiver) machines that are logically connected so that the output of one composite machine is input to the other (see Fig. 1). These abstract



Key (based on SNA terminology):
PU = physical unit
SSCP = system services control point
LU = logical unit
SM = services manager
HS = half session
DLC = data link control
LC = link connection

Figure 2 Overview of the SNA node.

machines have finite input sets and state spaces and thus can be modeled by the finite-state machines (FSMs) of classical automata theory [7].

Like hardware implementations, the implementations of a network architecture define logical structures that perform the specified functions; the elements are data structures, algorithms, modules, and processes. Different realizations may be distinguished by the language used to build the implementation (e.g., assembler or PL/I). Each SNA product constitutes a realization of one or more layers of the architecture. The designers of these implementations must understand the inputs received by the product and the outputs to be generated. The natural viewpoint is to consider how the product must function when it is operating as a node in the network. Moreover, the architects can prescribe the behavior of the network by specifying the behavior of a node in the most general case. Thus the network node is a useful focal point for both architecture and product designs.

A node can be represented formally as a composition of protocol machines. Figure 2 (from [8]) provides an architectural overview of an SNA node in the most general case. Each block in the diagram represents a composite protocol machine that can be successively refined until the most primitive protocol machines are reached. Routing and checking logic is defined to show how the protocol machines are interconnected and the signaling paths between them. Specification of the format, content, and ordering of information units flowing on a signaling path constitutes a *protocol boundary* between protocol machines.

SNA meta-implementation

The ensemble of the node elements just described constitutes a rigorous definition by reference to a model or *meta-implementation.* A meta-implementation closely resembles an actual implementation in that it has a well-defined structure and is expressed using explicit data structures and control flow. The underlying architecture is revealed when the meta-implementation can be interpreted (executed) and its resulting behavior (output) observed. An early example of a formal architecture specification in a form that was potentially machine-executable was the IBM System/360 computer architecture in APL [9].

The problems and payoffs in using a meta-implementation form of architecture specification are discussed by Brooks [10]. Of fundamental importance is the benefit that every question about the system behavior has a precise answer that can be determined quickly by executing the meta-implementation. Further, the requirement for machine execution forces greater attention to details; there are fewer overlooked cases. This also leads to a problem. If the meta-implementation is executed, overlooked cases produce answers that may not be what the architect intended. Because these unintended answers are produced by the meta-implementation they may be credited with a false precision.

In order to obtain a machine-executable model, many details must be filled in, including those that are options for the implementer. Thus, the meta-implementation "overspecifies" the architecture. For example, the architecture may require only that data elements be stored and retrieved in a certain order. In a meta-implementation, a particular data structure and representation must be chosen in order to execute the function (e.g., a doubly linked list). A product implementation may choose a different representation (e.g., a fixed-size array) to meet its cost or performance objectives. Much care must be exercised to clearly separate essential function from optional detail. On balance, however, rigor and completeness outweigh the problems of overspecification and false precision.

In the early versions of the SNA meta-implementation, line-diagram representations of FSMs were combined with prose and other graphic tools (block-structure diagrams, flow

charts [11]). These formal methods were developed to make the specifications precise, complete, and capable of being accurately communicated and understood; they were executable by people but not by machines. The availability of this form of meta-implementation, however, indicated the potential benefits of execution by a machine.

The most useful form of a meta-implementation is obtained when the reference model is expressed conveniently for use by implementers. Its structure should be generally useful so product developers do not have to reinvent the basic organization. A good global design used over and over yields great leverage from the architecture development process. Further, the audience for the architecture now consists mostly of programmers because the ubiquitous microprocessor has led to programmed implementations of almost all functions. Therefore, the medium for expression should be a high-level programming language. Natural language prose, even when augmented with diagrams and sequences, has proven too ambiguous (or redundant) for the job. As a consequence, the SNA architects have developed a high-level programming language tailored for the meta-implementation of an SNA node. This language, the Format and Protocol Language (FAPL), is discussed in the following sections.

Format and protocol language (FAPL)

As FAPL evolved, several objectives guided the architects' deliberations on formulating the basic language structure. Foremost among these was to make the meta-implementation accessible to a broad audience of product implementers in IBM, as well as to customers and others needing a complete, precise definition of SNA. This objective led to rejection of proposals for "new" languages in favor of extensions to a widely understood programming language, PL/I [12]. Another major objective was to preserve the basic node structure and notation already defined with the line diagrams and flow charts of the graphic representation. In particular, the use of FSMs to model protocols was considered essential. Another requirement was to provide language elements that model messages and their flow from one SNA-defined layer to another within and among network nodes. Other considerations such as language structure for top-down design and a rich set of data-definition facilities for format descriptions were also important.

A relatively small subset of PL/I was chosen to provide essential programming language capabilities for control flow, data definition, and operations on data. The data types of the language are integers, pointers, bit strings, and character strings, with appropriate rules for operations and expression evaluation using these types. Data aggregates may be defined by structures or one-dimensional arrays. The statements of the language include assignments, calls, declarations, do groups, if statements, procedures, returns, and select groups. Extensions to this subset that provide the functions needed in FAPL are in two areas: 1) finite-state machines, and 2) modeling messages and message flows.

- *Finite-state machines*

Finite-state machines are represented directly as objects in FAPL. A two-dimensional graphic form was chosen for the FSM definitions. In this form, horizontal and vertical lines delineate rows and columns. The rows represent possible inputs and the columns represent states. At each row-column intersection is the definition of the next state and output resulting from receiving the input (row) in that state (column). Inside the matrix, mnemonic symbols are used to name inputs, states, error conditions, and output-generating procedures. Output procedures can be defined using any FAPL statement. The matrix form has two desirable properties. For the reader it is a concise representation that allows complex information to be referenced and easily verified for completeness (every intersection must have a defined next state and output). It is also a form that can be automatically translated into executable PL/I statements.

The actions of an FSM are invoked through an extension to the PL/I call statement. When "called," the FSM input and current state are used to evaluate the next-state and output-generating functions. The new state is saved in storage local to the particular FSM instance and becomes the current state. The output may be represented by new values of any variables in the scope of the FAPL statements used to define the output function. Other operations are provided to test the current state of an FSM and to perform specialized validity checks.

- *Messages and message flows*

In the SNA meta-implementation it is desirable to distinguish the flow of messages from one layer to another from the flow of execution control among program statements. The flow of messages conveys vital architecture information; the processing order among statements is needed only to understand how the model executes. Using the normal PL/I execution-control mechanism of procedure calls and returns (passing the message as a parameter) prevents a separation of these functions. In many cases a message is sent to another component of the node for processing at some later (but unspecified) time, and execution continues in the sending component to handle related housekeeping. It is also desirable to show in the meta-implementation how layers and components can operate as concurrent processes.

In order to permit a more natural model of message flow and processing in the meta-implementation, three concepts were introduced: 1) a SEND statement that transfers a message to another component, 2) independently executable

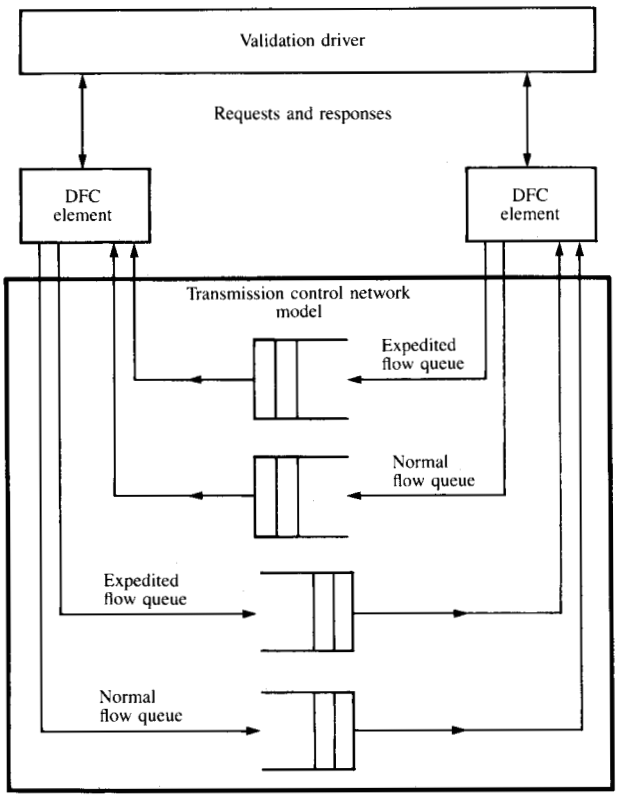


Figure 3 The system used to validate the DFC layer.

processes, and 3) queues that act as triggering mechanisms for process execution. The queues are used to model communication and unspecified order of execution among processes.

To model the messages and queues, two additional data types were added to the PL/I base: lists and entities (an entity is a data aggregate that may be placed in a list). Originally the only planned use of entities was to represent messages. It was soon found, however, that the generality of the list data structure encouraged users to represent control blocks, tables, and other forms of data internal to the node model. Lists and entities can be created and destroyed, entities may be added and removed from lists, all entries on a list may be processed, and a list may be searched for a specific entity. List-handling operations that treat lists in insertion or priority order are provided (specialized lists may also be manipulated directly by operations on the representation). There are also a number of built-in functions to manipulate entities on a list and perform simple tasks such as testing whether a list is empty. For a more detailed description of the language the reader is directed to Appendix N of Ref. [8].

Making FAPL executable

FAPL programs are translated by a preprocessor program that generates PL/I code which is then compiled by a PL/I compiler and executed. The preprocessor handles only those statements that are extensions to PL/I, simply passing the others through to the PL/I compiler. Most of the translation is straightforward; translation of matrix-form finite-state machines into PL/I procedures is, however, a non-trivial process. Details of the implementation can be found in Ref. [13].

Validation

In the SNA meta-implementation, the internal consistency of the executable form is ensured by using automated validation methods. Protocol validation is a form of state-reachability analysis in which the compiled FAPL code for a meta-implementation component is executed in an environment that simulates the remainder of the network. These validation techniques were developed at the IBM laboratory in Zurich [14, 15], as an extension of the techniques used to validate the X.21 protocol [16] recommended by the International Telegraph and Telephone Consultative Committee (CCITT).

The first large-scale application of validation was an investigation of the design for data-flow control functions in SNA. Data-flow control (DFC) is an SNA layer that defines one of the peer-to-peer (i.e., DFC-to-DFC) protocols between the two halves of an SNA session. DFC provides functions for sequence numbering and logical chaining of user messages, for correlation of requests and responses between end-users, for control of send/receive concurrency between them, and for bracketing (or serially multiplexing) transactions on the session [8]. DFC is the major synchronization point for controlling the order of messages exchanged between end-users of an SNA session. Its specification includes 30 FSMs and 3000 FAPL source statements.

A detailed account of the DFC validation is given in Ref. [13], so we discuss it only briefly here. Figure 3 shows the main components of the validation system developed for DFC. The validation system simulates the environment within which DFC layers in different network nodes communicate. The network layers below DFC are represented by a system of queues that accurately model the network delays and message-delivery order. The DFC elements are the compiled architecture; they communicate with the network model and the validation driver via defined protocol boundaries.

The validation driver itself has a number of functions. Starting from an initial system state, the driver can drive the system into further reachable states until all states have been explored. In any state it can pass to either DFC element a

message to be transmitted and so drive the system into further reachable states that result when the executable DFC elements process the messages and send them through the network. It can also define and observe the state of the DFC elements and the network queues. As the reached-state tree is generated, the driver examines the state definitions to detect deadlocks, loss of synchronization, and other states that should not occur. The validation of DFC uncovered about 20 previously undetected synchronization errors, some involving long, interleaved sequences of 10 or more unique messages and an even larger number of state changes (Fig. 4 gives one such example). It is probably not possible to routinely find such errors by manual inspection, and they occur so infrequently that detection during tests or field operations is extremely difficult.

Validation techniques provide a powerful tool for detecting errors in the design of communication protocols. They also provide a systematic means for exhaustively testing the meta-implementation program. All reachable states are explored and thus all paths through the FAPL code are executed with all valid inputs. Very thorough component or module testing of the FAPL code is obtained as a by-product of validation. Our experience shows that ten or more meta-implementation programming errors are detected for every protocol synchronization error. Validation is applied on a layer or component basis; system testing with sequences is required to detect errors in the specification of protocol boundaries between layers.

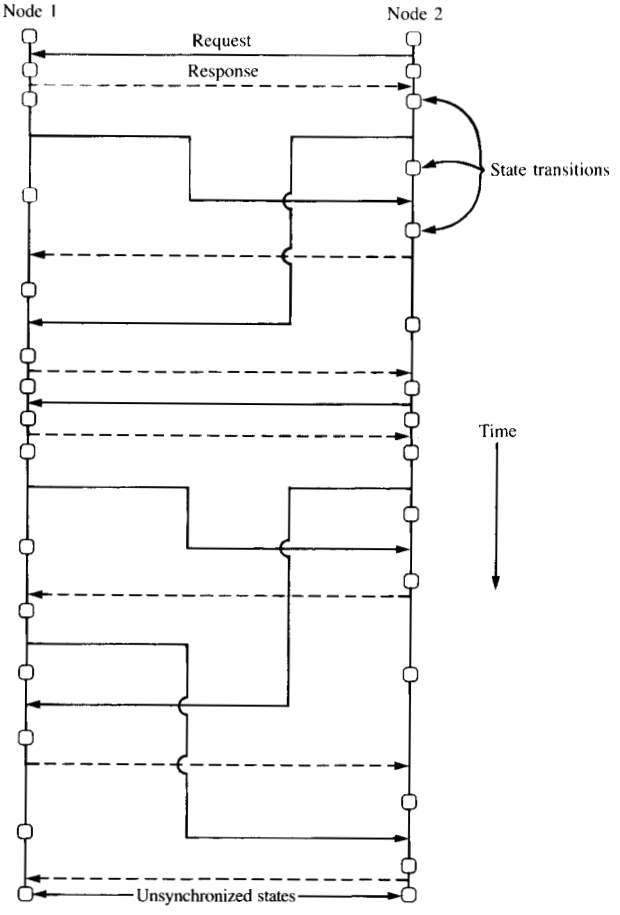


Figure 4 An architecture error found during validation.

Direct implementation

In principle, a correct implementation of the architecture can be automatically generated from a correct machine-interpretable representation of the architecture. When the architecture representation is a meta-implementation, as is the case for SNA, the generation process would appear to be a straightforward compilation. If this can be done, only very simple testing is required. In practice, a number of significant problems emerge when this is attempted, largely because of fundamental differences between an architecture representation and the design of a product derived from it. A product designer must optimize performance and cost; these are of lesser importance in an architecture representation, which must clearly and completely describe the functions of the system without giving unnecessary details that may restrict the optimal design of an implementation.

As a consequence of their differing purposes, it is often desirable for an implementation to have a structure that differs significantly from that of the architecture model. For example, an architecture designed for a multiprocessing environment may be considerably simplified when implemented in a single-processor system. The implementation of an allowable subset of the architecture functions may require much less storage than the full architecture. Such fundamental structural changes cannot be performed automatically and require significant design effort.

Another significant problem is that computer-network software is usually one component of a larger system (i.e., the control program for a distributed computer). Integrating code generated from the meta-implementation into such a system may require extensive modifications to the code to adapt it to existing data structure formats and content. Additional consideration must be given to adapting the code to use services provided by the system's control program (e.g., storage allocation and I/O).

The first attempt at direct implementation was modest in scope but sufficiently realistic to test practical application of the ideas. The DFC implementation in one component of the programming support for the IBM 8100 Information System was obtained by adapting the meta-implementation code to run on that system. Only a subset of DFC was required, so

the FAPL code was first manually reduced to a size of 10 FSMs and 1000 FAPL statements. Next, the meta-implementation was adapted to the data structures and control program services used in the product code. Several extensions to existing control blocks were required to provide DFC-related data. Control program services such as storage allocation were accessed by special macros generated for FAPL statements requiring these services (e.g., CREATE). Finally, the FAPL preprocessor was extended to generate PL/DS [17].

The results of this project were very encouraging. DFC was compiled, tested, and shipped with no major problems. Most errors were involved with adapting the FAPL code to a product environment. DFC functions worked properly largely because they had already been tested extensively by protocol validation. The final code was within the path-length and space budgets established for the project. Most important, the DFC component was built and tested for about one-third the programming resources that would have been invested otherwise. Most of this programming effort was spent in pioneering work associated with adapting and translating the FAPL for a specific product environment.

To learn more about the process of direct implementation, more experiments have been conducted outside the context of implementing a real product. In one experiment, several components of the meta-implementation were compiled and tested to implement a hypothetical network node containing only SNA functions. That is, no attempt was made to adapt the FAPL to a product environment; the environment was a raw machine with a given computer architecture (for concreteness, the 8100 architecture was chosen). Part of the experiment was to construct a simple control program to run the FAPL code on a real machine. The components of the meta-implementation that were included required 10 000 FAPL source statements, including 50 FSMs. This experiment confirmed that the key problems in automatic code generation from the meta-implementation are 1) rules and methods for automatic subsetting, 2) adapting the FAPL run-time system to different real machines, and 3) generating code that compares favorably in space and path length with tailored implementations.

In another experiment, automatic code generation is being used for building development and testing tools. Development teams building terminal devices that operate in SNA networks must test their implementation of protocols for communicating with other nodes. It may not be practical or economical for these groups to install and operate real networks just for testing. The solution is often a test tool operating in a virtual machine environment that acts as a surrogate for the real network node(s). Obviously, such tools must properly implement the architecture and be economical to develop and use. One such internal test tool has been built by compiling the SNA meta-implementation to generate code. This test tool uses about 10 000 FAPL source statements from the meta-implementation. An interesting sidelight is that the tool developers chose to use FAPL as a programming language to build testing and usability features that are outside the architectural protocols.

These early attempts at direct implementation have been successful on a modest scale. Because of the problems described above, the current methods for direct implementation can be applied only when development costs or time constraints are more important than optimization of function or performance. Advances in programming languages and compilers, especially those that provide efficient, portable software, may have a significant impact on the feasibility of direct implementation [18].

Architecture compliance and testing: principles

Direct implementation from an architecture solves two important problems, namely defining what constitutes an implementation of the architecture and how to test that any particular implementation complies with the architecture. If an implementation is derived directly from the architecture, the problem is reduced in essence to verifying a compiler. To the degree that the development of an implementation is not automated, and particularly when the structure of the implementation differs significantly from the architecture, verification of architectural compliance is much more difficult.

The internal structure of most implementations differs from that of the meta-implementation. Constraints of particular products and performance requirements dictate structural changes that may, for example, lead to the elimination of internal protocol boundaries or the assignment of functions to concurrent processes in a way that differs from that specified by the meta-implementation.

Consequently, it is extremely difficult to define architecture compliance in terms of the structure of an implementation. The only practical definition of compliance is to require that the external behavior of an implementation correspond to that of the meta-implementation. This definition dictates the nature of a test for compliance. An implementation can be tested for architectural compliance by applying a sequence of test inputs to both the implementation and the meta-implementation. Any difference in their behavior indicates that the implementation does not comply with the architecture.

Piatkowski [19] has indicated a problem of this approach: If it is not possible to directly observe the internal state of an implementation (as is generally the case), the upper limit of the length of the test sequence needed to prove equivalence is

approximately proportional to the cube of the number of internal states. Even modestly complex implementations have millions of different internal states, so it is not possible to perform even a reasonably complete partial test in a finite time.

It is interesting that experience has shown that most communication systems are too complex to describe by enumerating the sequences of messages that their components exchange. Their architectures are, therefore, now generally defined by the structure and algorithms in communicating processes that generate the message sequences [20]. On the other hand, compliance is defined in terms of exchanged message sequences and does not directly reference internal node structure. It may be that definitions of compliance and testing procedures that are closely related to the internal structure of processes in the reference model, rather than their external behavior, may be similarly advantageous.

Architecture compliance and testing: practice

The estimates of test-sequence length given above suggest that a thorough test of architecture compliance is impossible. The existence of many reliably operating implementations of the architecture demonstrates that this is not so, and there are a number of reasons why the test-sequence-length estimates provide an overly pessimistic view of testing.

The first is that they indicate the upper limit for a test designed to demonstrate exact equivalence. Demonstrating equivalence based on the external behavior of an implementation requires exercising and evaluating all possible message sequences that the implementation may receive. An implementation can exhibit good reliability yet contain errors if these are encountered only in rarely executed sequences. Tests of reliability need not address all sequences that have a very low probability of being encountered in normal operations.

Figure 4 shows an error found during validation of the architecture. After an exchange of 14 messages the layer being validated reached a state where the communicating elements were out of synchronization. The complex interleaving of the messages in the sequence leading to this error would only occur under particular network delay and loading conditions and would have a vanishingly small probability of being exercised in normal operations. Even if the error had not been found during validation, its presence in a derived product would not significantly degrade system reliability.

The second reason why the test-sequence length estimates are pessimistic is that it is not necessary to exercise all of the states because individual errors can manifest themselves in many states and must be detected only once. This can best be

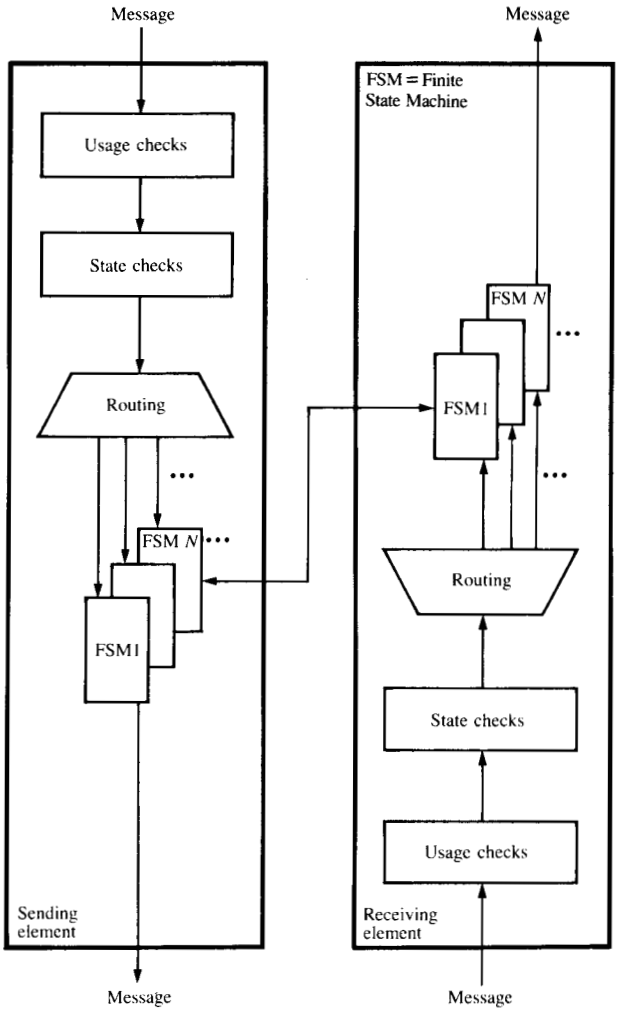


Figure 5 The general structure of a layer.

explained with reference to the architecture meta-implementation. Figure 5 (taken from Ref. [8]) shows the general structure of an SNA layer. The layer consists of two elements, handling respectively incoming and outgoing messages, each providing two levels of message checking: usage checks that perform global checks on messages and state checks that reject messages that are not acceptable in the current state of the layer.

Messages successfully passing the checks are routed to one or more finite-state machines that embody the layer functions. The sending and receiving elements cooperate by shared usage of appropriate finite-state machines. The number of reachable internal states of such a layer is a function of the number of states of the individual FSMs and the degree to which their states are correlated.

Experience in validating architecture models of this type indicates that the errors that are found can be characterized as 1) errors that occur in the checking or routing logic, 2) errors in particular states of individual finite-state machines, or 3) errors in particular state combinations of two or more finite-state machines. The errors are localized in their effect, independent of the state of many of the components of the layer, so that each error may be detected in large numbers of different internal states.

To summarize, our experience with the executable architecture definition shows that while the number of reachable states in a layer may be astronomical, the layer itself can be represented by a few hundred (or at most a few thousand) program statements, and the number of errors found by validation (the assumption is made that all are found) is a small fraction of the number of program statements. The errors found are localized in their effect and each may be detected in any one of a large number of internal states. Exercising a subset of the internal states therefore exposes a disproportionately large fraction of the errors.

The same may be concluded with respect to an implementation of the architecture. Much of the structure of an implementation is copied from the architecture. Where this is not possible, the design is still similarly structured, and the resultant localization of errors means that individual errors will manifest themselves in many states. The number of errors will be similar to those found in other large software systems; they are proportional to the number of program statements [21].

It is thus safe to conclude that the type of testing that is currently performed is considerably more effective than the type of theoretical calculations presented in Ref. [19] suggest. However, the arguments that support this are only qualitative. The way in which the external behavior of a system must be tested makes it extremely difficult to assess to what extent a test exercises the system and how reliable the system will be in field operations. The only guide to the significance of a test is prior experience from testing similar systems.

The increasing complexity of networks (measured in terms of size), the number of different types of communicating network components, and the sophistication of the distributed functions that are supported suggest that it will be important in the future to define compliance and testing procedures so that quantitative measurements of compliance are possible. Indications of the directions that may be taken are discussed in the next section.

Improved testing methodologies

Piatkowski [19] has discussed a number of ways in which a reference model may be used to evaluate the behavior of an implementation during testing. The SNA meta-implementation has been used in a number of experimental test tools to provide a check of implementation behaviors. It has also been compiled with special statistics-gathering facilities of FAPL so that the coverage of functions by test cases can be directly measured. Results indicate that special care in the development of test tools and the writing of test sequences is necessary if good coverage of error recovery situations is to be obtained.

The validation technique discussed in an earlier section makes a number of different testing techniques possible. The meta-implementation cannot directly be used as a generator of test sequences because it must be driven by external inputs. A by-product of the validation procedure is a single FSM model of the validated component in a form such that a simple exerciser can generate from it the sequences addressed by a validation run. When the number of sequences corresponding to a validation is extremely large, a variety of techniques can be used to generate a useful subset.

Sarikayi and Bochmann [22] have recently published results of applying a number of well-known sequence-generation techniques to a simple protocol. Such techniques could be applied to generate test sequences from a state machine derived from validation. Heuristic methods, possibly using randomly generated sequences, are more appropriate for complex problems where relatively complete coverage is not possible.

The use of the validation technique to generate test sequences has the advantage of producing more comprehensive tests in those cases where the resources available for test sequence production are limited. It also can probe areas of the implementation that manually generated test sequences may not address.

A more interesting possibility would be to replace an architecture-defined element in a validation run with an actual implementation. The validation system shown in Fig. 3 could equally well be used to validate the self-consistency of an implementation of DFC with the architecture, simply by replacing one of the DFC elements from the meta-implementation with its corresponding implementation. This would require that the implementation provide access to its internal state, but not that the implementation duplicate the architecture structure. Comparisons between validations of architecture and implementation components would permit more thorough testing than that obtained through observation of external behavior.

Conclusions

The development of an executable representation of SNA has significantly improved the quality of the architecture

definition. The requirement that the definition be compilable and executable encouraged the architects to produce a definition that is unambiguous and much more complete than was previously possible using less formal techniques. Many problems are exposed and corrected while the architecture is being defined that would otherwise stay hidden until subsequent product development.

An executable definition is not only a clear statement of the architecture, it is also a foundation upon which tools can be built that improve the quality and reduce the development cost of both the architecture itself and the products that implement it. It is the first step in the development of a software-engineering system for network product development.

Automated validation of the executable architecture definition has demonstrated that it is possible to detect (and thus correct) extremely complex errors using automated techniques. Results so far indicate that the number of residual errors in the architecture can be reduced by an order of magnitude in this way.

The experiments in automatic generation of implementations from the architecture definition have demonstrated feasibility and cost savings when performance is not a major constraint. There are a number of problems to be solved if a broad range of implementations are to be produced in this way. Subsetting and optimizing code that is intended as a general and easily readable definition of the architecture is not a simple task.

Finally, the executable architecture definition can be used as a reference model in testing products for architecture compliance. The increasing number and sophistication of network products are producing a need for tools that can provide general tests of architecture compliance. A number of test tools based on the executable architecture have been developed that provide this function. However, testing complex protocols is still an art. Deriving numerical estimates of test coverage and estimating reliability from test results is a challenging problem for all concerned with the development of network systems.

We expect further developments in architecture representation and implementation in the future. Higher-level architecture definition languages, improvements in compilers, and declining costs of large-scale integration will broaden the applicability of direct implementation techniques that promise cheaper and more reliable network software.

Acknowledgments

The development of the meta-implementation definition of SNA and its uses described here have involved many people throughout IBM. SNA itself involved the cooperative efforts of many architects and product designers. H. R. Barnes, R. F. Bird, J. P. Gray, D. B. Rose, G. D. Schultz, and R. J. Sundstrom played major roles in developing the first machine-executable (FAPL) version. D. B. Rose led the design of the FAPL language in collaboration with those listed above; many key contributions were made by J. P. Gray. Rose also wrote the FAPL-to-PL/I translator program. H. Rudin and P. Zafiropulo played major roles in developing the automated validation techniques. Pioneering work was done in direct implementation by S. Nash, G. Pursey, K. Soule, and F. Parr, and in compliance testing by R. M. S. Cork.

References

1. E. H. Sussenguth, "Systems Network Architecture: A Perspective," *Proc. ICCC 78*, Kyoto, Japan, Sept. 1978, pp. 353–358.
2. R. J. Sundstrom and G. D. Schultz, "SNA's First Six Years: 1974–1980," *Proc. ICCC 80*, Atlanta, GA, Oct. 1980, pp. 578–585.
3. G. A. Blaauw, "Hardware Requirements for the Fourth Generation," *Fourth Generation Computers: User Requirements and Transitions*, F. Greuenberger, Ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1970, pp. 155–168.
4. F. P. Brooks, Jr., "Architectural Philosophy," *Planning a Computer System: Project Stretch*, W. Buchholz, Ed., McGraw-Hill Book Co., Inc., New York, 1962, pp. 5–15.
5. *IBM System/370 Principles of Operation*, Order No. GA22-7000, 1976; available through IBM branch offices.
6. J. P. Gray, "Synchronization in SNA Networks," *Protocols and Techniques for Data Communication Networks*, F. Kuo, Ed., Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981, pp. 319–368.
7. A. Gill, *Introduction to the Theory of Finite-State Machines*, McGraw-Hill Book Co., Inc., New York, 1962.
8. *Systems Network Architecture Format and Protocol Reference Manual: Architecture Logic*, Order No. SC30-3112-2, 1980; available through IBM branch offices.
9. A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth, "A Formal Description of System/360," *IBM Syst. J.* **3**, 198–262 (1964).
10. F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley Publishing Co., Reading, MA, 1975.
11. T. F. Piatkowski, "Finite-State Architecture," *Proc. 7th Ann. Southeastern Symp. System Theory*, Auburn University, Auburn, AL, and Tuskegee Institute, Tuskegee, AL, March 1975; IEEE Cat. No. 75 CHO968-8C.
12. *OS PL/I Checkout and Optimizing Compilers: Language Reference Manual*, Order No. GC33-0009, 1976; available through IBM branch offices.
13. G. D. Schultz, D. B. Rose, C. H. West, and J. P. Gray, "Executable Description and Validation of SNA," *IEEE Trans. Commun.* **COM-28**, 661–677 (1980).
14. P. Zafiropulo, C. H. West, H. Rudin, D. D. Cowan, and D. Brand, "Towards Analyzing and Synthesizing Protocols," *IEEE Trans. Commun.* **COM-28**, 651–661 (1980).
15. C. H. West, "General Technique for Communications Protocol Validation," *IBM J. Res. Develop.* **22**, 393–404 (1978).
16. C. H. West and P. Zafiropulo, "Automated Validation of a Communications Protocol: the CCITT X.21 Recommendation," *IBM J. Res. Develop.* **22**, 60–71 (1978).
17. *Programming Language for Distributed Systems Reference Manual*, Order No. SC27-0446, 1979; available through IBM branch offices.
18. M. H. Conner, F. N. Parr, and R. E. Strom, "Portable, Secure Communications Software," *Proc. ICCC 81*, Denver, CO, June 1981, pp. 9.4.1–9.4.6.

19. T. Piatkowski, "Remarks on the Feasibility of Validating and Testing ADCCP Implementations," *Proc. Trends Applications: 1980, Computer Network Protocols Symp.*, Gaithersburg, Md., May 1980, pp. 94–109.
20. R. J. Sundstrom, "Formal Definition of IBM's Systems Network Architecture," *NTC '77 Conf. Record* **1**, 3A 1-1–3A 1-7 (1977).
21. A. Endres, "An Analysis of Errors and Their Causes in System Programs," *IEEE Trans. Software Eng.* **SE-1**, 140–149 (1975).
22. B. Sarikayi and G. Bochmann, "Some Experience with Test Sequence Generation for Protocols," *Proc. 2nd International Workshop for Protocols, Protocol Specification, Testing, and Verification*, Idyllwild, CA, North-Holland Pub. Co., New York, 1982, pp. 555–567.

Received July 13, 1982

F. D. Smith *IBM Communication Products Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709.* Dr. Smith is manager of the Architecture Technology Department and is responsible for the language (FAPL) and compiler used in the definition of SNA, and for software technology based on the executable architecture (e.g., automatic protocol validation). He received the B.S. in chemistry and the M.S. in industrial management from the University of Tennessee, Knoxville, in 1962 and 1964. He earned the Ph.D. in computer science from the University of North Carolina, Chapel Hill, in 1978. In 1965, he joined IBM as a systems engineer in the Chattanooga, Tennessee, branch office and was responsible for installing one of the first IBM System 360/50 systems using OS/360. He joined the Raleigh laboratory in 1968, working in modeling and performance evaluation of computer networks. Since 1971, he has been a member of the architecture organization. He has worked on a variety of projects in computer design, including storage hierarchy, cache memory analysis, microprocessor architecture evaluation, and the architecture of the IBM 8100 Information System.

C. H. West *IBM Research Division, Saumerstrasse 4, 8803 Ruschlikon, Switzerland.* Dr. West joined IBM in 1971 at the Zurich Research laboratory and has worked on laboratory automation, computer graphics, communications, and computer networks. He is currently active in the areas of communications protocol validation and image processing. He received the B.Sc. in physics in 1960 and the Ph.D. in elementary particle physics in 1965 from Imperial College, London, England. From 1961 to 1966 he was a visiting scientist at the European Organization for Nuclear Research (CERN) in Geneva, Switzerland, and subsequently held postdoctoral positions in the physics department at the Moore School of Electrical Engineering at the University of Pennsylvania.