Robert Strom
Nagui Halim

A New Programming Methodology for Long-Lived Software Systems

A new software development methodology based on the language NIL is presented. The methodology emphasizes (1) the separation of program development into functional specification and tuning phases, (2) the use of a fully compilable and executable design, (3) an interface definition and verification mechanism. This approach reduces life-cycle costs and improves software quality because (a) errors are detected earlier, and (b) a single functional design can be re-used to produce many implementations.

1. Introduction

NIL is a programming language and development methodology designed to support large and complex systems which are intended to be re-implemented in many product environments and which can be expected to last through several generations of hardware and several functional releases.

The NIL language is described more fully in [1, 2]. The application of NIL constructs to distributed software systems is discussed in [3]. This paper discusses the relevance of the NIL language and methodology to the software development process.

Our methodology distinguishes two kinds of change to software: (1) changes for which function is preserved but performance objectives or hardware technology is required to be different, and (2) changes which alter function.

The ability to support the first kind of change is called *portability*, and is achieved by (1) defining a high level of abstraction for the primitives of the programming language, and (2) permitting multiple *realization mappings* [3] to map these abstract primitives into an implementation. The ability to support the second kind of change is called *extensibility* and is supported by enforcing modularity within the language.

The features we require to permit extensibility include information hiding, module interchangeability ("plug-replacement"), and dynamic reconfigurability.

Section 2, "Separation of architecture and implementation levels," discusses the level of abstraction of NIL and how NIL programs serve as an interface between the phases of functional design and implementation. We include a discussion of how NIL can be used to support portability of a design. Section 3, "Support for modular architecture," explains how a NIL compiler can enforce well-known principles of good modular design which are normally left to human management. Finally, in Section 4, "Experience with NIL," we discuss some of the consequences of our design decisions and our experience in using NIL to construct a prototype of a portable SNA communications subsystem.

2. Separation of architecture and implementation levels

Our development methodology factors the implementation of complex system software into two activities—(1) *architecture:* the production of a functionally correct, machine-independent specification of the system, and (2) *tuning:* the generation of an implementation of this specification tailored

to the cost, performance, and physical hardware constraints of a particular product environment.

Since the same design may be re-implemented several times—either across the members of a family of products or over time—we call the machine-independent design a *software architecture.*

Central to our methodology is a very high-level language, NIL, which is at a level of abstraction appropriate to the boundary between architecture and implementation. The result of the architecture phase is a set of NIL modules representing the algorithms of the system.

On the one hand, NIL is an algorithmic language, rather than a pure specification language. On the other hand, the primitive data types and operations of NIL do not refer to the particular machine structures which are needed to implement these algorithms. Using NIL, the following sorts of implementation decisions, which affect only performance, are not visible in the architecture, but are decided by the compiler and modified during the tuning phase:

- Use of registers, main storage, and secondary storage for holding program states.
- The order of scheduling of logically independent activities, and the degree of actual concurrent overlap of operations.
- Whether memory resources are pre-allocated or allocated on demand.
- Whether data structures are packed, unpacked, contiguous, discontiguous, linked, or whether common data are shared.
- How many regions or address spaces are used, and whether communication between these address spaces uses data copying or shared buffers.

An architecture design specified in NIL is compilable and executable. Because the design is executable, it may be tested for conformity with functional and human factors requirements. Errors in the design of the system or of individual algorithms can be detected and corrected at this point. Corrections at this stage may be much cheaper than corrections occurring later in the product development cycle.

The initial executable design will not always meet the performance objectives of the product or products which are to be implemented. The evolution of a functionally correct design with inadequate performance into an efficient implementation is achieved by modifying the *realization mapping*, which converts NIL's high-level abstract primitives into low-level instructions and data structures. The realization mapping is embedded in a NIL compiler and a run-time environment specialized for the target design.

Our model for performance tuning involves the following steps:

- Instrumentation of the code to detect (1) which primitive operations are executed most frequently on which objects, (2) what are the optimum values for parameters such as buffer sizes, number of buffers, and scheduling priorities, (3) which modules of the program are candidates for hand tuning.
- Use of the results of instrumentation to select *pragmas* (compiler directives) which override the default implementation of NIL primitives either globally or for selected objects or selected modules.
- Modification of the compiler code templates for particular operations whenever the existing compiler lacks an adequate pragma option.
- Hand optimization of a few selected critical modules.

The tuning phase and the architecture design phase can be handled by separate organizations, since both the objectives and the required skills are very different. The emphasis of the architecture group should be on the details of the algorithms and protocols of the system, and on appropriate modularity to achieve extensibility. The emphasis of the tuning group should be on making most effective use of the underlying hardware. The tuning group need have only enough knowledge of the system functions to construct appropriate performance benchmarks.

The tuning phase must be repeated for each product or design point for which the architecture will be re-implemented, whereas the architecture design is performed once and is modified only as the function changes. When a design point changes due to technological improvements, new pragma options may be added to compilers to re-implement the NIL primitives to make appropriate use of new hardware. Because the primitive abstractions of NIL are application-independent, these new options may eventually be used in re-implementing many different functional architectures.

We believe our methodology to be cost-effective for software projects of sufficient magnitude that

- The function will be required on a family of products with distinct design points (e.g., IBM's Systems Network Architecture), or
- The function will be required to migrate over time to either different design points (e.g., fail-soft operation) or new hardware technologies, or
- The function is large and complex enough so that (1) it is advantageous to assign separate groups the responsibilities of understanding the functions and understanding the machine implementation, and (2) the cost of modifying a compiler to tune performance is less than the cost of redesigning the function.

3. Support for modular architecture

The second thrust of NIL methodology is facilitating functional modifications to large long-lived systems. These changes may occur (1) across the lifetime of a system, (2) across members of a product family, or even (3) from one installed system to another.

Although the impact of design-point changes tends to be greater in that the entire global system structure may change, the frequency of changes in functionality is much higher. Furthermore, certain kinds of changes can be more appropriately designed and installed by the user of the system than by the developer. In such cases, the ability of the system to be modified in particular ways becomes as much a part of the system specification as its functional and performance requirements.

The following widely accepted principles of structuring systems facilitate functional change.

- *Information hiding* The state of a system should be partitioned in such a way that each module is aware of only a small fraction of the state—its *secret* [4]—and that no module can depend on another module's secret.
- *Narrow, explicit interfaces* A system should be divided into modules with minimum coupling. Modules should know about other modules only via an *interface specification* which defines how the effects of one module are visible at another module.
- *Plug compatibility* A module should be replaceable by another module with a different algorithm which preserves identical interfaces to adjacent modules. In a so-called "open" system, the replacement can be made even by the users of the system.
- *Dynamic reconfigurability* Because systems can be reconfigured at run time as well as statically configured at system generation time, system configuration logic must be embodied in modules of the system, rather than in external binders and linkers.

Applying the above criteria ensures that (1) when a change is contemplated, it is easy to determine the relevant modules to change without having to inspect or even understand the whole system, (2) the number of modules needing to be changed is small, and (3) internal changes to a module do not require propagation of changes to other modules.

Communications architectures, such as ISO [5] and SNA [6], apply the above principles in a very disciplined way:

- The system is partitioned into *layers*, each having a certain generic function.
- At each layer, there can exist a number of alternative algorithms. Alternative algorithms may include installation-specific or even user-specific programs, depending upon the layer. For example, the data link control layer may include protocols such as Binary Synchronous and X.25.
- The system architecture defines a fixed *manager process* at each layer whose job it is to dynamically bring up and take down new instances of programs at that layer (the *control processes*) and to choose which of the alternative control process algorithms is to be executed.
- The architecture fixes the interface between any control process and the control processes and manager processes at that layer and at the adjacent layers.

While all these are widely recognized as good design methodology, enforcement of this methodology in actual implementations is difficult, particularly after a product has gone through a number of releases. We believe that modular design principles should be used to *constrain* the design while it is being produced, not merely to *evaluate* an existing design.

The core of the NIL approach is a methodology which (1) *forces* interfaces and private data to be made apparent prior to compilation, (2) enforces the consistency of a program with its interface, and (3) enforces the privacy of data. Every syntactically correct NIL design automatically has narrow, explicit interfaces and information hiding; furthermore, the description of the interfaces and the partitioning of the state space can be discovered statically without following program logic. To be sure, NIL designs may contain errors, and NIL designs may make too few module cuts or may make cuts in inappropriate places. However, NIL designs will not contain "pseudo-modules" (modules which are closely coupled in undocumented ways).

We believe that the costs associated with functional extension are reduced due to the fact that the designer can easily determine with complete confidence which modules of a system are affected by a change to one of them.

In this section, we show how the NIL semantic model enforces accepted modular precepts and how the NIL language and compiler allow these concepts to be documented and enforced.

- *Processes and ownership*

A major distinguishing feature of NIL is that partitioning of the system into modules simultaneously separates the data space and the program space. A NIL system is divided into program modules called *processes*. Every data object belongs to exactly one process. Processes may only operate on data objects which they own. All objects owned by a process are declared within the text of that process. Except for explicit communication over ports (to be discussed later), all data

objects are independent in that operations on one object cannot affect any other object.

The NIL view of data and processes contrasts both with conventional programming practices and with other high-level models of processes and access control.

In many system designs, collections of *control blocks* are connected by pointer (reference) variables. Modules are invoked with an environment containing access to certain control blocks. In general, examination of program algorithms is needed to determine which pointers are potentially used to access which other parts of the system.

However, even if it is possible to determine which control blocks *are* accessed by a module, it is not possible to determine which control blocks *may not* be accessed by a suitably modified version of the module. No distinction is made between a module's potential access and its actual access. Since in many systems there are paths from almost every control block to every other one, a change to nearly any module may entail a change to nearly any other.

Various proposals exist to restrict the potential scope of access from a module to dynamic data. Data abstraction, as in SIMULA [7], CLU [8], ADA [9], and ADAPT [10], allows some or all of a control block (the *private* part) to be made inaccessible except within a collection of modules associated with the control block's type. Guardians, as in Argus [11], are an abstraction of the notion of virtual memory spaces. Processes within a single guardian can share access to common data structures, but guardians can affect one another only via message passing. Domains, or capability lists [12–14], are used to explicitly enumerate the potential access rights of all processes being executed within the domain. In general, a single object may be simultaneously accessible from several domains.

The NIL model is equally expressive, yet much simpler. Each process is conceptually a separate data space. Every object is owned by exactly one process. The set of objects owned by a process can be determined by examination of the declaration statements appearing in the process's source text. Sharing of objects is impossible, since there are no global variables, no nested scopes, and no pointers. There is no ambiguity over which process has responsibility to initialize or finalize data, since each object has exactly one owner. Access control in NIL deals not with the right to operate on objects, but with the right to bind communications ports to form communications channels.

Although NIL processes can be used to implement abstract data types [3], more flexible types of process

```
A: process uses(C)
declare
   AMSG: MTYPE
   APORT: PTYPE sendport
   S: EBCDICSTRING
begin
      . . .
   allocate AMSG;
   AMSG.DATA = 'Hello';
   send AMSG to APORT;
   . . .
end A;
```

```
B: process uses(C)
declare
   BMSG: MTYPE
   BPORT: PTYPE receiveport
   BPORT: PTYPE receiveport
begin
      . . .
   receive BMSG from BPORT;
   if BMSG.DATA = 'Hello'
      then
   . . .
end B;
```

```
C: definitions uses(D)
. . .

MTYPE is message
   (DATA: EBCDICSTRING)

PTYPE is send
   interface of MTYPE

end C;
```

```
D: definitions
. . .

EBCDICSTRING is string
   of EBCDIC_CHAR;
end D;
```

Figure 1 Example of asynchronous communication: Modules A and B are executable modules which are independently compiled. Module C is a type definition module used by modules A and B. Module D is a type definition module used by modules A and C.

interconnections are possible besides the hierarchical pattern required in the abstract data type model. In such configurations, no NIL process is more "abstract" than any other.

• *Communication*

Interaction between processes is provided in NIL, without compromising the principle of exclusive ownership, by means of operations which *transfer the ownership* of data from one process to another either temporarily or permanently.

There are two forms of communication: asynchronous (message passing) and synchronous (calling).

Asynchronous communication works as follows: Suppose process A is going to pass a message to process B (see Fig. 1). For this to occur, A must declare two objects: AMSG, a message object, and APORT, an output port. Similarly, B must declare two objects: BMSG, a message object, and BPORT, an input port. Sometime prior to the communication, a *connection* must have been made between APORT and BPORT, forming a queued communications channel. Process A issues operations to (1) **allocate** AMSG, which creates a message object with nothing in it, and (2) initialize the contents of AMSG. Process A then issues the **send** operation specifying AMSG and APORT. This causes the message data to travel to the other end of the communications channel and be queued inside BPORT. The variable AMSG now no longer denotes an allocated message object, and process A may no longer read or modify its contents.

Process B simultaneously (or more precisely, in its own local time, which is unordered with respect to process A)

issues a **receive** operation, specifying BPORT and BMSG. Process B then waits until one or more messages appear on the queue, and the **receive** operation is completed with BMSG denoting the first message in the queue.

Notice that AMSG (entirely local to process A) and BMSG (entirely local to process B) at different times denote the same data, but never simultaneously denote the same data. The data are always either owned by process A (as AMSG) or owned by process B (as the value of BPORT prior to the **receive** and of BMSG after the **receive**) until they are eventually explicitly destroyed.

(Notice that the semantics of message passing does not entail data copying in an implementation; between processes residing on a single machine, **send** and **receive** can be implemented using pointer chaining.)

Synchronous communication (*rendezvous call*) also involves ownership transfer, a pair of connected ports, and a message called a *call-message*. Process A (the *caller*) once again has an output port APORT bound to an input port BPORT in process B (the *acceptor*). Process A moves a set of objects (the *actual parameters*) into a call-message object, which is passed from APORT to a queue at BPORT. Process A then waits. Process B issues an **accept** operation, specifying input port BPORT, and a call-message BMSG. The **accept** waits until the call-message containing A's actual parameters arrives, and then this call-message becomes the value of BMSG, and the fields of BMSG (the *formal parameters*) can be used to refer to the call-message data passed from A. Eventually process B will issue a **return** operation, causing the call-message to return to process A, causing A to resume and B to no longer own the call-message.

As with asynchronous communication, because there is no data sharing, synchronous communication can be implemented by data copying ("value") or pointer copying ("reference").

• *Dynamic configuration*

Ports in separate processes are connected to form communications channels. This connection must occur before communication can take place. Because we want the set of processes and the set of connections to be decided at run time, we provide run-time operations for process creation and access control, rather than requiring that the identity of the partner be specified in the program text.

Connections are made in two steps: (1) The owner of an input port **publishes** an access right to that port into a **capability** object; and (2) the capability is then passed to another process which issues a **connect** operation that binds a specific output port to the input port designated by the capability. A process acquires its initial capabilities from its creator via *initialization parameters*, which are exchanged at the time of process creation. Thereafter, it may acquire and export capabilities via normal communication.

The principal distinguishing features of NIL's communication mechanism are the following:

- There is no communication via shared data. In that respect, NIL resembles CSP [15] and Gypsy [16].
- Unlike the above languages, the connection of output port to input port is made dynamically—the sender does not have to name the identity of the receiver.
- Ownership transfer is sufficiently abstract that it encompasses both value copying (as in Argus's inter-guardian communication) and reference copying.
- Passing an object does not entail passing other objects reachable by reference from the passed object. The caller therefore knows for certain that, on return from the call, only the passed parameters were manipulated. In other systems, a called program may correctly specify that it receives, for example, a "Data Control Block" as parameter, and yet this does not guarantee that other structures will not be manipulated.

• *Types and interfaces*

The requirement of plug-compatibility of modules entails that when A communicates to B over APORT, it does not know statically that it is communicating with module B, since B might be replaced by some similar module BPRIME. The requirement of dynamic reconfigurability entails that the choice of module might be made at run time. In fact, A may own an entire table of objects like APORT, each connected to a different module.

To support plug-compatibility, NIL supports the independent compilation of processes. To ensure that A and B are connectable despite having been compiled separately, NIL requires that A and B each refer to an *interface definition*, which is contained in a module C referred to by A and B.

Interface definitions are examples of *type definitions*. Every variable in NIL is declared with a type, which has previously been separately defined in a definitions module. Definitions modules are separately compiled into *definitions libraries* and are time-stamped, as in MESA [17], to permit version checking.

The type of a variable is permanent throughout its lifetime, while its value may change. In the case of a port, the type is its interface and the value is its binding, which may be a connection to any similarly typed port.

NIL defines a set of classes of types for which type constructors exist. These classes are called *type families*. The

type families of NIL include integers, booleans, enumerations, components (collections of processes), relational tables, messages, call-messages, ports (interfaces), and variants. A new type is defined by giving its family name and other information required by the constructor.

In Fig. 1, the type definition C referred to by modules A and B defines the following types: MTYPE, the type of the messages AMSG and BMSG, and PTYPE, the type of the ports APORT and BPORT. The definition of MTYPE itemizes the message fields, specifying a name and a type. The definition of PTYPE specifies that the port sends and receives messages of type MTYPE.

Compile-time type checking guarantees a number of properties statically, using only the text of the executable module being compiled and the type definitions it uses:

- The type of each variable name can be determined statically, e.g., AMSG.DATA is known to have type EBCDICSTRING, since AMSG has type MTYPE, and MTYPE's definition specifies that field DATA has type EBCDICSTRING.
- Each operation can be checked to see that its operands are of appropriate type, e.g., **send** AMSG to APORT requires that AMSG be a message type, that APORT be an output port of send interface type, and that AMSG's type be consistent with APORT's interface.
- Versions are checked for compatibility: e.g., if process A declares a variable of type EBCDICSTRING, whose type was defined in module D, then the version of D in effect while compiling module A must be the same as the version of D in effect while compiling C, which also uses EBCDICSTRING.

Typestate

In the earlier example, communication by message passing involved creating a single data object which was sometimes visible to process A under the name AMSG and sometimes to process B under the name BMSG. The "destructive" semantics of **send** specifies that fields in AMSG can no longer be read or written after a **send.** Similarly, fields in BMSG cannot be read or written before a **receive.**

From process A's viewpoint, variable AMSG has three "states": (1) UNINITIALIZED: the state prior to **allocate** and subsequent to **send,** when data may not be written or read, (2) EMPTY: the state after **allocate** but before initializing any of the data, when data may be written but not read, and (3) FULL: the state after the message is fully initialized.

In NIL, an extension to type checking called *typestate checking* ensures that the compiler can determine the state of each variable at each point in the program and reject attempts to issue an operation from the wrong state.

For each type, the language defines a set of *typestates.* (For example, type MTYPE has three typestates: UNINITIALIZED, EMPTY, and FULL.) The language further defines which operations are permitted in which typestates and which operations cause typestate transitions. Typestate checking rules [18] permit compilers to statically determine the typestate of each variable at each point in the program and to verify that operations are issued only from correct typestates. Each synchronous interface definition must specify for each parameter not only its type but also its required typestate on entry to the service, on return from the service, and on each possible exception return from the service. (Certain frequent combinations, e.g., **input, output, inout,** can be specified with a single keyword.) As a result, interface specifications explicitly mention whether the caller owns the data before and after each call and whether the called process may update or destroy the data.

Typestate checking has the following benefits:

- It preserves the security of the system by rejecting "erroneous" programs which may produce unpredictable side effects in some implementations (e.g., by referencing de-allocated storage). This mechanism is powerful enough to replace run-time integrity checking of the sort taking place in many systems.
- It lowers maintenance costs by rejecting erroneous programs before they cause errors in the field. The typestate algorithms check all possible paths through a program, not just those taken during the product testing period.
- It permits the clean termination of processes, since at any point in a program where a fatal error might be detected, it is known statically which variables correspond to owned data objects which must be finalized.
- It forces procedures which change the ownership of data or which receive uninitialized parameters and initializes them to declare that fact on the interface. (Conversely, it forces any procedure using such an interface to fulfill the obligation expressed on its interface.)

4. Experience with NIL

A full NIL compiler and run-time system have been written for the IBM VM/370 system (approximately 80 000 lines of code). Two SNA prototype systems have been written by a team of five Research Staff Members during a period of eight months. The prototypes consist of approximately 20 000 lines of executable code and 6000 of interface and type definitions. The experience to date has been extremely positive, both in terms of programmer productivity and the quality of the resultant implementations.

Designing openness and dynamic reconfigurability into the system from the beginning virtually forced a suitable process decomposition, because all components with separate lifetimes were required to be implemented as separate processes. As a result, their data were forced to be disjoint, and explicit interfaces were required to be designed between the layers. Once the interface definitions were written they became a fixed point of reference, and it was possible to assign the programming of adjacent layers to different individuals.

Typestate considerations, both internally within a module and at the module boundaries, forced the careful planning of data initialization and use. In no case did we find that the constraints imposed by typestate made it impossible or even awkward to write an algorithm. We believe that requiring typestate correctness results in improvements in readability analogous to those brought about by the use of structured programming.

Many errors in design and coding were detected at compile time or at bind time before any execution took place. These errors include failure to fully initialize data aggregates, failure of a called procedure to create or discard data as required by its interface, initializing data on one path to a program statement and not on another, unmatched interfaces between communicating modules, and inconsistent versions of an interface shared by an executable module and a type definition which it used. Had these conditions gone undetected, they would have resulted in over-written storage, program checks, and other symptoms whose detection and correction would typically require use of a core dump and expertise in the underlying implementation.

Debugging proceeded in two steps: (1) individual modules were exercised with dummy test layers, then, when judged to be free of errors, (2) integrated with the real adjacent layers. Once a module was debugged with the test layers it rarely failed during integration. All debugging was at the "source" level, with the majority of errors being either (1) dynamic binding errors, (2) value errors, or (3) deadlocks. Debugging time was typically very small compared to the length of time needed to write and successfully compile the module. While testing the system, we discovered a rather unexpected result: Once the programs were through the initial testing period, the error density and error rates dropped to almost zero.

Any exceptional conditions (e.g., inability to complete a dynamic binding) automatically interrupted normal control flow and invoked exception handlers. Because the system console displayed the site of the exception, it was easy for these problems to be pinpointed during debugging.

Execution-time performance of the first NIL implementation was about three to four times slower than the same function performed in a comparable IBM product. An execution-time tracing facility was used to determine where the problems were, and we discovered that approximately fifty percent of the execution time was spent in a small subset of the run-time system. A second implementation is now nearly complete. The path length of NIL statements has improved to the point where we believe our implementation to be nearly competitive with production quality implementations.

5. Concluding remarks

The NIL language and its methodology provide a means of writing well-structured functional descriptions at a level of abstraction that permits true portability. This eliminates the intertwining of performance decisions, design, and algorithms so typical of systems that exist today. The result is a system that is easy to maintain and one that does not suffer from a gradual decay in code quality during the system life-cycle.

Because NIL is an executable design language which is tunable to produce production-quality code, it becomes possible to extend the clean, modular decomposition conceived by the high-level designers directly through the functional design coding and maintenance phases.

Furthermore,

- Identical versions of the system can be made to execute on a wide variety of machines, reducing the duplication of coding effort and acceptance testing.
- We speculate that more uniform external specifications of products running on different machines will reduce customer migration problems and improve the marketability of upgraded hardware.
- Extensions to the system can be developed and sized with respect to a single common design. By contrast, within IBM at present, extensions to SNA routing function involve sizings by four separate product organizations, and extensions to session protocols may involve dozens.
- By rejecting at compile time programs which attempt to use unowned data or which are inconsistent with their interface, we eliminate major classes of errors before the program goes into execution, including errors which may only occur in infrequently executed program paths. Errors detected earlier are less costly to correct.

6. Acknowledgments

Other members of the group contributing to the design of the methodology and language are Wilhelm Burger, Mike Conner, Francis Parr, John Pershing, and Shaula Yemini. The NIL implementation team also included Jim McInerny, Dan Milch, Francis Parr, John Pershing, and Wilhelm Burger.

References

1. "Draft NIL Language Reference Manual," *Research Report RC-9732*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, December 1982.
2. N. Halim and J. Pershing, "A New Language for Writing Portable and Secure Systems," *Research Report RC-9650*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1982.
3. R. E. Strom and S. Yemini, "NIL: An Integrated Language and System for Distributed Programming," *SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, June 1983.
4. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Commun. ACM* **15,** 330–336 (1972).
5. H. Zimmermann, "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection," *IEEE Trans. Commun.* **C-28,** 425–432 (April 1980).
6. *Systems Network Architecture Format and Protocol Reference Manual: Architectural Logic*, Order No. SC30-3112, available through IBM branch offices.
7. O.-J. Dahl, B. Myhrhaug, and K. Nygaard, "SIMULA-67 Common Base Language," Norwegian Computing Center, Oslo, Norway (1970).
8. B. Liskov et al., *CLU Reference Manual*, Computation Structures Group Memo 161, MIT Laboratory for Computer Science, Cambridge, MA, July 1978:
9. "Reference Manual for the Ada Programming Language," Draft Proposed ANSI Standard Document, ACM AdaTec (July 1982).
10. B. Leavenworth, "ADAPT: A Tool for the Design of Reusable Software," *Research Report RC-9728*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1981.
11. B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *Ninth ACM Symposium on Principles of Programming Languages*, Albuquerque, NM, 1982.
12. A. K. Jones and B. Liskov, "A Language Extension for Controlling Access to Shared Data," *IEEE Trans. Software Eng.* **SE-2,** 277–285 (October 1976).
13. R. B. Kieburtz and A. Silberschatz, "Capability Managers," *IEEE Trans. Software Eng.* **SE-4,** 467–477 (November 1978).
14. P. Ancilotti and M. Boari, "Language Features for Access Control," *IEEE Trans. Software Eng.* **SE-9,** 16–25 (January 1983).
15. C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM* **21,** 666–677 (August 1978).
16. A. L. Ambler, D. I. Good, and W. F. Burger, "Report on the Language Gypsy," *ICSCA-CMP-1*, The University of Texas at Austin, 1976.
17. J. Mitchell, W. Maybury, and R. Sweet, *Mesa Language Manual*, Xerox Palo Alto Research Center, Palo Alto, CA, April 1979.
18. R. E. Strom, "Mechanisms for Compile-Time Enforcement of Security," *Tenth ACM Symposium on Principles of Programming Languages*, Austin, TX, January 1983.

Received May 19, 1983; revised August 31, 1983

Nagui Halim *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Halim joined IBM in the Research Division in 1980. He is currently a research staff member with the distributed systems software technology group, where he is involved with proving the feasibility of using the NIL methodology and language in implementing product quality software. His interests include operating systems, programming languages, and compiler code generation techniques. His previous work includes two major operating system implementations and the design and implementation of the control software for a communications front end processor. Mr. Halim completed his undergraduate studies in physics at Yale University, New Haven, Connecticut, in 1978.

Robert E. Strom *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Strom's experience in programming languages dates back to 1958, when as a student assistant at the old IBM Watson Laboratory at Columbia University, he developed a two-pass version of the three-pass FOR TRANSIT compiler for the IBM 650. As a student, he worked on a number of large software systems, including a text-processing system for the translation of machine-readable text to Braille, a system for syntactic analysis of natural language using attribute grammars, and a real-time system for the control of psychology experiments requiring guaranteed response latencies. He completed his undergraduate studies in philosophy and psychology at Harvard University in 1966, and his doctoral studies in applied mathematics and computer science at Washington University, St. Louis, in 1972. Since 1977, he has been a research staff member at the Thomas J. Watson Research Center, specializing in software methodology and programming language design, particularly as applied to communications and distributed systems.