NIL: A high-level language for distributed systems programming

by F. N. Parr
R. E. Strom

Network Implementation Language (NIL) is a high-level programming language currently being used for the implementation of prototype communication systems. NIL is designed for writing executable architecture which can be compiled into efficient code for the different machines and run-time environments of a family of communicating products. NIL's distinctive features include (1) high-level primitive type families supporting constructs needed for concurrent systems, (2) facilities for decomposition of a system into modules which can be dynamically installed and interconnected, (3) compile-time typestate checking—a mechanism for enhancing language security without incurring large execution-time overhead.

Network Implementation Language (NIL) is a high-level programming language developed by the Distributed Systems Software Technology (DSST) Group at IBM's Thomas J. Watson Research Center. The language was originally developed to implement experimental software for new and proposed communications protocols. The logical details of protocol algorithms had to be programmed completely and correctly in order to demonstrate and validate the prototypes; also, the software structure and layering of the prototypes had to be correct to ensure that they could be expanded into fully functioning systems. Taking as a starting point a technique for the formal specification of protocols,[1] the DSST Group developed NIL as a programming language that is (1) at a suitable level of abstraction for communications architecture specification, (2) sufficiently general purpose to be suitable for programming the product-specific functions not defined as part of the architecture, (3) effective for defining the configuration functions by which processes are initiated and terminated, (4) secure enough to support "open" layered systems, and (5) compilable into efficient code for complete software systems. The DSST Group has also developed a compiler for executing NIL programs on the System/370 and is currently using NIL to produce the software for some communications prototype systems.

NIL is intended to support generic systems designs that are not "tuned" to any particular hardware or software execution environment.[2] The aim is to provide portability not just among different machines but among *design points*. Potentially this characteristic of NIL software offers enormous advantages, particularly in the field of communications and distributed processing.

For example, in IBM's Systems Network Architecture,[3] there are protocols that must be shared between data processing hosts and multiplexor computers, or between multiplexor computers and intelligent workstations. With the use of conventional software techniques, the communications code for such a variety of systems is expensive to design and, furthermore, converting design into an implementa-

Figure 1 Definition of network routing example problem

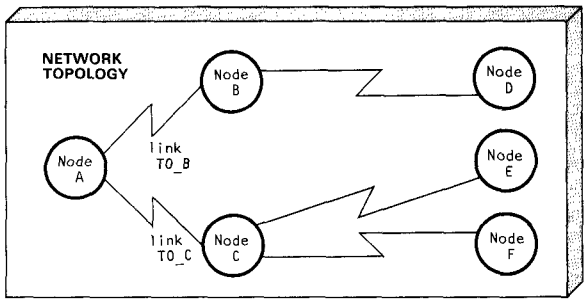
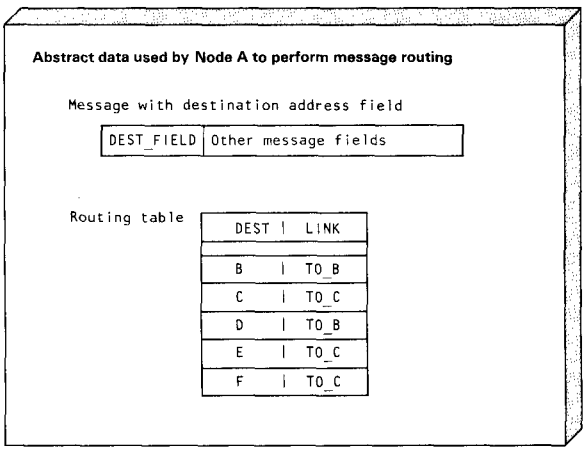


Figure 2 Example: The message routing problem



tion requires individuals who are knowledgeable both in communications architecture and in the target machine environment. Since the implementations are controlled by distinct product organizations and different development teams, it is extremely difficult to improve or extend the protocol algorithm without forcing major modifications on some of the implementations. When an architectural modification is agreed on, products based on independent designs will have difficulty in implementing the modification simultaneously. This leads to a profusion of subset implementations and an obligation to test all possible combinations.

These problems can be minimized if the architecturally defined functions can be specified as a single product-independent source system developed and tested within a group specializing in communication protocols and interfaces. Ideally, a system of NIL source programs would define the architecturally required functions common to all implementations in the product family, and would define the interfaces between those functions and external environments. A product implementation would be generated by combining the architecture-defined programs with product-specific NIL programs and by compiling them to produce directly executable code. In this way, both software development costs and the "inertia" of software evolution can be reduced.

Such a methodology, while producing the greatest payoff in communications software systems, is also likely to be valuable in other domains, such as work-flow management ("job entry") subsystems, operator interfaces, command languages, etc.

Language design principles

NIL attempts to meet its goals by means of the following strategies.

Representation-independent data description: Data in NIL is manipulated in a way that is completely independent of its physical organization and access. The language semantics exclude assumptions about the representation of data from having an effect on the correctness of a program. The particular representation compiled from a given NIL source program is controlled by the compiler, assisted by *pragma* annotations to the NIL source. Pragma annotations can be used by the compiler to choose an implementation whose performance best suits the design point. Selecting an alternate implementation affects only performance, not the meaning of the program.

Typical NIL systems are composed of independent layers.

Security: Typical NIL systems are composed of independent *layers*. Each layer defines interfaces to adjacent layers while hiding the internal data and algorithms constituting its implementation. A system may contain alternative implementations of a

single layer running concurrently, e.g., X.25, Synchronous Data Link Control (SDLC), and user-defined data link controls. It is a design objective of NIL to facilitate the coding of systems that allow user-supplied and system-supplied layer implementations to coexist. NIL's security mechanisms guarantee that a user-supplied module may interact with other modules only through their interfaces, and that each layer's private data shall remain private, *even in the presence of programming errors.*

NIL achieves its security guarantees without requiring expensive memory-protection hardware or run-time checks. This is possible chiefly because (1) NIL forbids direct manipulation of pointers by source programs, and (2) NIL supplements full type checking with *typestate checking*, which is a form of compile-time guarantee that invalid operation sequences will not occur.

Full set of system concepts: There is limited value in providing representation independence or security in a high-level programming language if the user is forced to escape into an underlying operating system for essential services such as tasking, dynamic introduction of new code, and the binding of ports between process instances. In NIL all these services are provided through language primitives whose semantics are consistent with a wide variety of underlying implementations. It is particularly important to have the passing of access rights to a newly created NIL process instance performed explicitly by the program creating the new process. Each program has statically defined interfaces, but the connections made to those interfaces are not fixed until that program is instantiated during execution.

Defining data without reference to its representation

NIL's approach to portability can be illustrated by means of a simplified programming problem. The message routing layer of a communications network node with multiple links must determine for each message it receives which link, if any, to use for the next hop. In Figure 1, Node A must forward messages that it receives for destinations B, C, D, E, and F. They can be forwarded either on link TO_B or on link TO_C. This decision is made using the destination address in the header of the received message and the routing tables saved in the node (Figure 2). This algorithm must be duplicated in every node of the network. However, particular implementations of the routing tables must reflect the requirements of the different network node machines. A simple end-user workstation might communicate only with a fixed, predefined number of host applications; in these circumstances an array representation of the routing information would be appropriate. Alternatively, a high-performance message-forwarding node might be required to keep track of thousands of final destinations and to dynamically respond to information about new destinations and new routes. In the latter context, a hierarchical, tree-based organization of the data would be more appropriate.

Figure 3 Routing table lookup in PL/I using an array implementation

```
data

dcl MSGPTR pointer;              /* message to be forwarded  */
dcl FOUND bit(1);                /* terminates scan loop     */
dcl X      integer;              /* index for array scan     */
dcl 1 ROUTTAB (10)               /* array of 10 entries      */
     2 DEST  integer,            /* a destination address    */
     2 QCBPTR pointer;           /* link queue for this dest */
dcl 1 MSG based,                 /* message data structure   */
     2 DEST_FIELD integer,         /* destination address    */
     2 DATA_char(255) var;         /* message data           */

code

FOUND = '0'B;
LOOP: do X = 1 to 10 while(¬FOUND);
  if ROUTTAB.DEST = MSGPTR->DEST_FIELD
    then FOUND = '1'B;
  end;
if FOUND
  then call ENQUEUE(MSGPTR,ROUTTAB(X-1).QCBPTR);
  else
    ... /* error action for unknown address */
   ;
```

If the routing algorithm module is designed using a programming language such as Pascal or PL/I, both the declarations of the routing data and the algorithms for access to routing tables necessarily reflect a decision as to whether to organize the data using arrays or trees. In Figure 3 and Figure 4, sample codings of the two representations are shown. It is apparent that even though these code fragments implement the same function, there is hardly a line of code in common among them. There is little chance of being able to use one source-level definition of the abstract algorithm to generate the other.

The above example illustrates the loss of portability through *overspecification.* If the properties of a data object are defined by relying on a sample representation of it, it will be extremely difficult to

Figure 4 Routing table lookup in PL/I using a sorted tree

```
data

dcl MSGPTR pointer;          /* message to be forwarded  */
dcl FOUND bit(1);            /* terminates scan loop     */
dcl X     pointer;           /* cursor for tree scan     */
dcl ROOT  pointer;           /* locates routing table    */
dcl 1 TABLEREC based,        /* tree node record         */
     2 DEST  integer,        /* a destination address    */
     2 QCBPTR pointer,       /* link queue for this dest */
     2 LEFT   pointer,       /* to smaller dests subtree */
     2 RIGHT  pointer;       /* to larger dests subtree  */
dcl 1 MSG based,             /* message data structure   */
     2 DEST_FIELD integer,   /* destination address      */
     2 DATA char(255) var;   /* message data             */

code

FOUND = '0'B;
X = ROOT;
LOOP: do while(¬FOUND & X ¬= null);
  if MSGPTR->DEST_FIELD = X -> DEST
    then FOUND = '1'B;
    else
      if MSGPTR->DEST_FIELD < X->DEST
        then X = X->LEFT;
        else X = X->RIGHT;
  end;
if FOUND
  then CALL ENQUEUE(MSGPTR,X -> QCBPTR);
  else
    ... /* error action for unknown address */
   ;
```

Figure 5 Routing table lookup in NIL

```
data

     declare
       ( ROUTTAB : table
           ( DEST  :  integer key,
             LINK  :  MSG1_TYPE sendport),
         AROW :   row in ROUTTAB,          -- temporary for lookup
         MSG1 :   MSG1_TYPE  )             -- message to be routed

code

      block
        find AROW in ROUTTAB key(MSG1.DEST_FIELD);
        send MSG1 to AROW.LINK;

      on(NOT_FOUND)  ...
                                  -- error action for unknown address
      end block;
```

show that implementations using an alternative representation are correct.

The NIL program for the routing algorithm described above is insensitive to whether the data is represented as a tree or an array. (See Figure 5.) Either the compiler will choose the appropriate representation, or it will offer the user a menu of choices which can be selected by specifying a pragma.

The type constructor used to cover both array and tree structures is an abstract table associating a unique "link for next hop" with any valid "destination address." There is a single set of language-defined operations for accessing data in abstract tables: the verbs FIND, INSERT, and DELETE.

NIL's primitive abstractions are organized into a set of type constructors called *type families*. Each type family is designed to provide considerable latitude to the compiler implementer while at the same time providing a simple set of rules by which the NIL programmer may reason about his programs. For example, *message* and *table* are NIL type families representing movable data records and extensible collections of data associations, respectively.

Relationship of NIL to data abstraction languages

Every programming language provides a set of base data types and data type constructors. Data abstraction languages[4–6] concentrate on mechanisms for defining new types from more primitive ones. In practice, these languages have chosen the conventional memory-oriented types, such as arrays, as primitive types. In these languages the programmer has to define higher-level types appropriate to his needs using the memory-level constructs.

Although data type definition may be useful as a data-hiding and program-decomposition mechanism, the failure to define suitable high-level types as part of the programming language has disadvantages:

- The design of a type family such as NIL's abstract table is not easy—the type family must have broad applicability and must be parameterizable to produce useful individual types; the semantics defined for the type should not excessively constrain future implementers. Individual users are more likely to develop specialized types with specific problem areas and even with specific implementations in mind.
- The semantics of user-defined types are normally inherited from an encapsulated implementation; any additional information about the abstract specification of properties of the type is optional and usually omitted.

Above the level of the NIL primitives, NIL and other languages supporting abstraction have much in common. All these languages support the decomposition of complex programs into layers that hide their internal structure from adjacent layers and provide access only through narrow interfaces.

(Some of the differences in NIL's approach are discussed in a later section covering run-time mechanisms for binding adjacent layers.)

Other data abstraction languages differ from NIL chiefly in that

- Users are not required to program at any particular level of abstraction; they may choose to program using pointers and other low-level primitive types, whereas in NIL, pointers are never directly visible.
- NIL specifies a complete set of operations, operation sequence rules, and semantics for a recommended set of type families for use in systems programming.

System implementation using NIL is factored into two phases. During the *functional design phase*, algorithms are coded in NIL, checked, and debugged. Functional design considers issues of modularity, extensibility, and algorithmic complexity, but not data representation. During the *optimization phase*, appropriate data representations are selected so that the system meets appropriate performance criteria, such as code size, system throughput, response time, fault-tolerance, etc. The choice of representation can be made either (a) automatically by the compiler, (b) manually by selection of an appropriate pragma, (c) by *a posteriori* determination of the best pragma based upon instrumentation of the executing code, or (d) in the worst case, by augmenting the compiler by adding support for an alternative representation.

System implementation using NIL is factored into two phases.

The NIL type families

There are 15 predefined type families in NIL. These are briefly summarized below.

- *integer* — The natural numbers. Individual implementations will vary in their capacity to implement all sizes of integers.
- *boolean* — The truth values *true* and *false*.
- *enumeration* — Each type defines a discrete set of values with user-defined names. There is ordering defined, but no arithmetic.
- *string* — Sequences of elements, which may be of boolean or enumeration type. Strings may be copied, compared, or concatenated. Contiguous substrings may be extracted or overlaid.
- *table* — A set of zero or more row objects. Each row is a set of fields whose name and type may be defined by the user. All rows within a table are of the same type. The user may define certain attributes to be *keys* (meaning that their values are unique within a table), and may define the collection of rows to be *ordered* or *unordered*. Rows may be inspected, updated, inserted, or deleted. The entire table, or a *selection* (subset) of the table, may be enumerated.
- *row* — Used to access individual row objects within a table.
- *tableset* — A collection of related tables, which are created or destroyed as a unit. Tables within a tableset may be related as follows: one of the tables may be the domain of an attribute in another table. For example, if one table is a PART-SUPPLIER relation, and another a SUPPLIER-CITY relation, the domain of SUPPLIERs in the first relation may be constrained to come from the members of the SUPPLIER-CITY relation.
- *variant* — A collection of fields of user-defined name and type, grouped into mutually exclusive *cases*. When a variant is initialized, exactly one of these cases is initialized. A *select block* is available to check the current case of a variant by branching to an appropriate *case clause*. Within each *case clause*, the programmer may access fields within that case.
- *message* — A collection of fields, whose names and types depend on the message type. Messages may be dynamically allocated and destroyed as a unit. Messages may be communicated between processes by sending a message to a *sendport* and receiving it from the connected *receiveport*.
- *interchange message* — An object that can be accessed as a collection of fields, but for which a complete mapping into bit string format has been defined. Because interchange messages are bit strings, (1) they may be used as a universal medium of exchange to I/O devices or to other programs outside the NIL environment, (2) they may be *remapped* as a different collection of fields having the same total bit string length, and (3) they are restricted to contain those NIL types for which bit string mappings are defined, namely

Figure 6 An ADA example of language integrity failure

```
task body A is
  type SESSION is access PROTOCOL_3;
  THIS_SESSION: SESSION;
  begin
    THIS_SESSION:= new PROTOCOL_3;
    ...
-----------------------------------

task body B is
  type R is access RCB;
  R1,R2: R;
  begin
    R1:= new RCB;
    R2:= R1;
    R1.F1:= ...
    . . .
    UNCHECKED_DEALLOCATE(R1);
    . . .
    R2.F1:= ...
    . . .
```

scalars, and for aggregates whose ultimate constituents are scalars.

- *call message* — A collection of fields called *parameters*. Call messages are used in synchronous communication (subroutine calls) to pass data from the caller to the called procedure or process, and to return data to the caller.
- *call_interface* — A queue of call messages of homogeneous type. Each call_interface is either a *callport*, an *acceptport*, or a *procedure*. One or more callports is connected either to an acceptport or to a procedure. Then calls are made over the callport. If the callport is connected to a procedure, a new procedure activation is created when the call is made and destroyed upon return to the caller. If the callport is connected to an acceptport, the call is queued. The caller waits until the acceptport owner dequeues the call, processes it, and returns it.
- *send_interface* — A queue of messages of a particular message type with all fields initialized. Every send_interface object is declared to be either a sendport or a receiveport. Communication is achieved by connecting one or more sendports to a receiveport. Messages sent over the sendport may be received by the owner of the receiveport.
- *catalog* — Stored access rights defining the ability to connect a callport to a specific acceptport or procedure, or to connect a sendport to a specific receiveport.
- *component* — A dynamically loaded collection of processes and procedures, as seen from the creator of the component. The owner of a component has no ability to see objects belonging to any modules in the component. Owners may test for completion and completion status of a component and may cancel a component.

Motivation for security

The term "security" is sometimes loosely applied to any programming language and run-time system with facilities for detecting and reporting program errors. In this paper, the term is used with a more precise meaning specifically relevant to the reduction of integration and testing effort. Our use of the term is consistent with its use by Hoare in his Turing Award Lecture:[7] *security* is the principle that every syntactically incorrect program should be rejected by the compiler and that every syntactically correct program should give a result or an error message that was predictable and comprehensible in terms of the source language program itself.

In NIL this principle of security is extended by also requiring that no system of modules that has been demonstrated to be working correctly can be made to fail by executing in the presence of other modules that have no language-defined interface to the original system.

Many cases of unauthorized interference between programs result from performing operations on data in the wrong representation. Type checking has widely become recognized as the means to overcome this form of security failure. However, type checking alone is not sufficient to prevent the reading of uninitialized data and access to dangling or uninitialized pointers and entry variables. The ADA program in Figure 6 illustrates how these errors can destroy security. Task A is assumed to be a correctly written task which may have been formally validated. Task B is an incorrect program executing anywhere in the same system, which deallocates a dynamic object and *subsequently* accesses it. The error in Task B will not in general be detected by an ADA compiler and will cause Task A to behave in a completely unpredictable way. Although Task A has been shown to be "of itself" correct, it cannot be trusted to execute correctly unless certain properties of all other software executing in the same system are verified.

Errors associated with the use of uninitialized data and with pointers aliasing data are extremely hard to detect in a programming language that permits

direct manipulation of pointers. Faced with this problem, most high-level programming languages have resorted to some combination of the following remedies:

- Omitting the language constructs that can give rise to the problems
- Requiring an elaborate or expensive run-time system—typically garbage collection or interpretive execution
- Abandoning language security and passing on the problems to the compiler writer or user of the language

NIL is a secure programming language in the sense defined by Hoare. NIL programs may interact with each other only via their defined interfaces, and these interfaces do not include sharing of an abstract memory. The rules of NIL permit secure implementations without the usual performance penalties. Specifically, NIL run-time environments

- Need not perform garbage collection
- Need not explicitly check for uninitialized data before use
- Need not worry about the possibility of concurrent access to a data item from within two processes
- Need not worry about the possibility that a process may terminate or be canceled without freeing its resources

It is extremely difficult to provide a single completely general security mechanism that can apply to all program abstractions or to all uses of pointers. In providing security, NIL takes advantage of the fact that each type family can be given a set of security rules appropriate to the programming technique which that type family embodies. We shall use the semantics of the *message* type family in NIL to illustrate this point.

How security is enforced in the NIL message type family. A standard technique in systems design is to communicate between processes via queues of messages. A process wishing to send information first allocates a message object, then fills its fields with information and sends the message off to a queue owned by the process that is supposed to receive this information. That process may dequeue the message at any subsequent time and process its fields.

Since messages may contain many fields, and copying large amounts of data is expensive, most efficient implementations of message passing within a single physical memory allocate the message object from a global heap and perform *send* and *receive* by copying a pointer to the sender's message into a queue owned by the receiver. A key requirement of this implementation is that after a process has sent a message to some other process, *the sending process must lose the capability to read or update fields in the message.* If this restriction is not enforced, it becomes extremely difficult to write correct programs in a multiprocess environment, since sender and receiver may have concurrent access to the same data. If the receiver then disposes of the message, the sender will have a dangerous "dangling reference." Although a single message may become accessible to different processes at different times, at any one time a message should be accessible to at most one process. When a multiprocess system is coded in an insecure language such as assembler, the restriction on message use is preserved either by informal agreement among the programmers not to access messages after they have been sent or discarded or by a run-time system that physically moves messages across address space boundaries.

A message type is defined by giving a list of field names and types.

The NIL message type family facilitates the definition of message objects with these characteristics. A message type is defined by giving a list of field names and types. Each variable of that message type will have those fields. In a NIL program, message variables may be declared with a specific message type. A message variable is "initialized" by an ALLOCATE operation or a RECEIVE operation, receiving it from a queue associated with some receive port; the RECEIVE waits until a message object is available on that queue, then dequeues it and makes it accessible via the message variable operand. A message variable is "uninitialized" by a DISCARD operation, or by a SEND operation sending it through a particular port to some other process. While a message variable is initialized (but not at

Figure 7 Examples of correct and incorrect message-handling code

```
Segment A - valid program

-- MSG1_TYPE is a message type defined elsewhere with integer field F1

-- APORT is a declared variable of type "sendport of MSG1_TYPE"

      block
      declare ( MSG1 : MSG1_TYPE)
        allocate MSG1;
        MSG1.F1 = 3;
        send MSG1 to APORT;
      end block;

Segment B - invalid program

-- MSG1_TYPE is a message type defined elsewhere with integer field F1

-- APORT is a declared variable of type "sendport of MSG1_TYPE"

      block
      declare ( MSG1 : MSG1_TYPE)
        send MSG1 to APORT;
        MSG1.F1 = 3;
        allocate MSG1;
      end block;
```

any other time), any message field qualified by that message variable name may be read or updated. The advantage of offering an access-destroying, nonblocking SEND operation and a blocking RECEIVE is that copying a pointer to the message data and copying the actual message data are both valid implementations, indistinguishable from the perspective of the NIL programmer. The implementer is therefore free to select whichever strategy is more efficient in each case. Adopting other semantics for message communication tends to force the more expensive data copying implementation on all sending operations.[8]

The restriction that gives the message type family its special flavor and results in system-wide assurance that no message can be simultaneously accessible to more than one process is the implied rule on the *ordering* of operations on each message variable. To see that this restriction cannot be enforced by type checking rules alone, consider the two program segments in Figure 7.

Segment A is a valid code segment allocating a message, initializing its only field and sending it to some other process. Segment B differs from Segment A only in that the message operations are performed in the reverse order. Segment B has correct syntax, and every operation has operands of the correct type. Therefore, a conventional strong type checking compiler will not reject it at compile time. However, Segment B presents significant problems for a secure language implementation and in NIL would be rejected at compile time. For a programming language to permit Segment B and still provide security, the implementer would be required to choose from one of these approaches:

- Detecting and generating a run-time error
- Forcing a message object to be allocated and sent even though no ALLOCATE operation was issued
- Treating the SEND as a null operation in this case

Following the SEND statement, Segment B initializes a field in an uninitialized message. Once again if the program has not been rejected at compile time, nonobvious choices will have to be made by the language designer either to cope with the problem as an error at execution time or to invent semantics that make all operations valid in all contexts.

The program segments in Figure 7 show that the operations on messages have simple obvious meanings only when they are performed in the correct order. The NIL philosophy is that both the semantics of the programming language and the run-time system which supports it can be drastically simplified if rules for correct *sequences of operations* are given as part of the language definition and enforced at compile time—or to be more precise, at a time when individual modules are processed *one at a time*, without the "compiler" having access to the complete system in which the modules will run. Even in an interpretive implementation of NIL, "compilation" will occur, taking the form of a check of each module before any statements from that module are interpreted. NIL emphasizes the special role of compilation in this sense, because separate compilation is an essential property for software systems.

Separate compilation is an essential property for software systems.

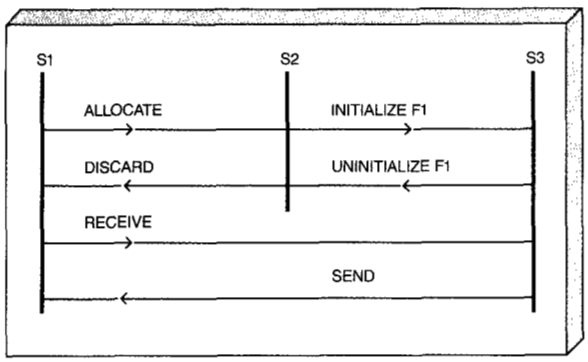
Typestate as a general mechanism for security

The compilation technique used in NIL to check the sequences of operations on variables is called *typestate* analysis. For each user-defined data type, the NIL compiler will construct a state machine whose states (called *typestates*) determine which subset of the defined operations on objects of the type are currently permitted. During compilation of a program, a typestate machine is maintained for every declared variable, tracking its typestate from statement to statement in the program.

The typestates of message variable MSG1 introduced in Figure 7 are

S1	=	MSG1	uninitialized
		MSG1.F1	unaccessible
S2	=	MSG1	initialized
		MSG1.F1	uninitialized
S3	=	MSG1	initialized
		MSG1.F1	initialized

and the typestate transition graph for this variable is



Every valid sequence of operations on a message variable corresponds to a valid transition sequence on the above graph. In reading this diagram, it must be remembered that typestate is a property of *variables* and not of *values*; when a message is sent, the message *value* continues to hold its fields and flows to some other process, whereas the message *variable* on which the send was performed becomes "uninitialized" since the sender has lost access rights to that message. Note also that a SEND operation is permitted only when the fields in a message are all initialized, and similarly the RECEIVE operation assumes that it will always receive a fully initialized message. It is technically possible to relax this pair of restrictions. However, preventing processes from exchanging partially initialized messages simplifies coding and fosters the design of systems with clearer interfaces.

Figure 8 Example of a typestate error

Segment C

```
block
declare ( X:      integer
          MSG1:  MSG1_TYPE )
  X = AFUNC;   -- initializes X

  C1: if X = 1
      then
         allocate MSG1;
      end if;

  C2: ... ;

  C3: if X = 1
      then
        MSG1.F1 = 3;
        send MSG1 to APORT;
      end if;

end block;
```

Given typestate transition graphs for every declared variable in a module, a complete checking algorithm can be defined.[9] Although run-time checking of typestate is possible, it carries a performance overhead and prevents code generation from exploiting typestate knowledge. Compile-time determination of typestate is not possible for general programs. In NIL, a compile-time determination of typestate becomes practical because of the addition of the following critical requirement: the typestate of all variables must be a *program invariant* at every statement in a program; i.e., the typestate of a variable at a given statement must be independent of how that statement was reached. The NIL type families, control structures, and exception-handling mechanisms are designed specifically to ensure that the invariance of typestates can be maintained without loss of power or flexibility for the programmer. The typestate invariance principle is an important new program structuring concept that gives NIL its unique flavor.

Program Segment C in Figure 8 illustrates the effect of requiring the typestate of variables to be a program invariant. Even though all paths through the program are meaningful, this segment would be rejected by a NIL compiler as having a typestate error even if Block C2 does not alter the value of

Figure 9 The typestate error example corrected

```
Segment D

block
declare ( X:      integer,
          MSG1:   MSG1_TYPE )
  X = AFUNC;   -- initializes X

  C1: if X = 1
      then
         allocate MSG1;
         call BODY_OF_C2;
         MSG1.F1 = 3;
         send MSG1 to APORT;
      else
         call BODY_OF_C2;
      end if;

end block;
```

Figure 10 Routing table lookup with an exception flow

```
Segment E

E1: block
   E2: block
       declare ( AROW : row in ROUTTAB )
         find AROW in ROUTTAB key(MSG1.DEST_FIELD);
         send MSG1 to AROW.LINK;
         . . .
         allocate MSG2;
         MSG2.F1 = 3;
         send MSG2 to APORT;

      on(NOT_FOUND)  ...
            -- error action for unknown address
      end E2;

    on(DEPLETION)
         -- recovery action from allocate failure

         -- AROW will be uninitialized and use counts
         -- in ROUTTAB cleared at this point
    end E1;
```

variable X. The problem is that the compiler has to decide on a typestate for variable MSG1 on entry to Block C2 so that it can determine whether access to MSG1.F1 should be permitted inside that block. To ensure that the typestate of MSG1 at that point does not depend on the path taken through the preceding IF BLOCK, the compiler will actually generate a DISCARD operation on MSG1 at the end of the *then* path and issue a warning message on the listing. Automatically generated *typestate coercion* operations may be generated at any point where control flows converge and will always force objects to the "least initialized" of the states on the different flows. When Block C2 is later encountered, MSG1 is therefore in the uninitialized typestate (S1), and hence, the access to MSG1.F1 is a typestate error.

A correct NIL implementation of the intentions of the program in Figure 8 is given in Figure 9. The correction is to collapse the two IF blocks with the same test into a single IF block. We claim that the resulting program is more readable because it does not rely on inferring that an identical test is repeated. Also, the correct version is more robust because it prevents some program maintainer who only partially understands the system design from inserting code into Block C2 which would alter the value of X and hence invalidate the original program logic.

Typestate coercions and exception handling. The typestate coercions mentioned in the preceding example become beneficial in the presence of exception flows. One consequence of the level of abstraction of the NIL type families is that many of the defined operations can raise *exceptions*. The language reference manual defines for each operation the names of the exceptions that may be raised, as well as the effect on the values and typestates of each operand. When an exception is raised, control flows outward through the static block structure to the nearest exception handler which lists that exception name. If no handler is found in the current procedure, the exception name is converted to an exception name known by the caller of the procedure as part of its call interface.

Figure 10 contains an example of exception flow and an automatically generated typestate coercion to clean up after it. One of the defined exceptions of the ALLOCATE operation is DEPLETION. The temporary variable AROW used to access the routing table has Block E2 as its scope. So if the DEPLETION exception causes control to flow from the ALLOCATE operation to the DEPLETION exception handler, a DISCARD operation will be coerced on AROW. This automatically generated operation is necessary since some implementations of ROUTTAB may have unpacked records or set use-counts when AROW was initialized to a particular row in this table. In other table implementations, discarding a row variable may have no execution time effects (discarding a row variable does not delete the underlying row object from the table—it merely makes it unaccessible until some subsequent FIND retrieves it).

The effect of typestate coercions is to provide automatically generated recovery code after an exception, which will "clean up" to the state expected by the recovery handler. Writing this recovery code explicitly after every possible exception would clut-

ter up the normal path through algorithms and make the system design hard to read. Graceful backout from exceptions also tends to be extremely difficult to program correctly without automated assistance. It should be noted that the NIL recovery—forcing uninitialization of program objects as necessary to enter a common exception handler—is considerably simpler than the problem of backing off a partially completed *transaction*, in which case all value changes caused by the transaction have to be reversed. See Reference 10 for a classic paper on this more difficult topic.

The general aliasing problem: other aspects of typestate

So far in this paper, typestate mechanisms have been described and illustrated entirely in terms of message allocation. Each of the different NIL type families has its own set of typestate transition rules achieving security in a slightly different way. For each family, it must be shown that the typestate of variables of the family can always be statically determined, and at the same time shown that the typestate restrictions are powerful enough to ensure that a wide range of implementations of the family is possible.

The effect of typestate coercions is to provide automatically generated recovery code after an exception.

The typestates of variables with no contained fields (e.g., integers, booleans, etc.) are

- *unaccessible* — The variable is a field inside a compound variable which is itself either uninitialized or unaccessible.
- *uninitialized* — The variable has no defined value.
- *initialized* — A defined value exists, which may be read, modified, or destroyed.
- *constant* — The variable's value may be read but not modified: in this state it is safe for an implementation to choose to share the variable's storage with that of other variables of the same value. Because updating is not permitted, this potential for aliasing cannot be detected.
- *permanent* — The variable may be read or updated, but not destroyed. Fields in compound objects that are required to be in an all-initialized state will have this typestate.

The possible typestates of a compound variable are generated by defining the structure tree showing the relationship of all fields in the variable, and labeling each node in the tree with one of the five basic typestates.

The particular type families used in NIL and the associated typestate rules combine to achieve a total separation of the variable spaces of each module. Specifically,

- There is no detectable aliasing—within a module, every distinct variable name has an independent value. Changes to one variable have no effect on values of other variables.
- There is no sharing—every variable is local to some module. Values of accessible variables of a process cannot be altered by actions of other processes. The only way in which a process sees the interaction of other processes is via the ability to receive messages and accept calls on ports.

Aliasing and sharing impede modularity, since understanding how a particular module can be affected from the outside requires the analysis of not only that module but also of all others coresident in the system. Sharing and aliasing are endemic in languages that permit uncontrolled pointer copying or that allow modules to inherit shared access to variables in statically containing scopes. NIL resolves these problems by

- Avoiding shared scopes
- Having a limited set of type families and thereby avoiding direct visibility of pointers
- Using typestate to prevent references to fields in messages already sent away or not yet allocated
- Forbidding a variable from being bound to two different formal parameters of a single procedure call unless the formal parameters were declared as read-only (CONSTANT typestate)
- Using a combination of compile-time and run-time checking to guarantee that two different row

Figure 11 Functional layers in a communications architecture

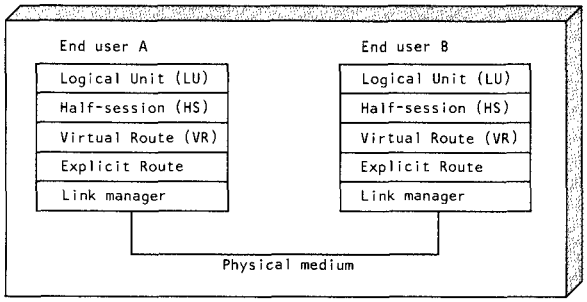
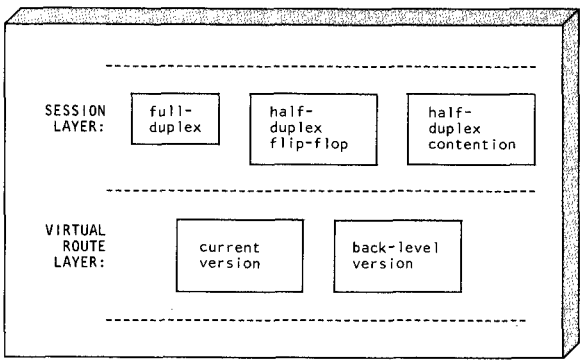


Figure 12 Examples of options within a layered architecture



variables may not access the same row of the same table, unless they are both requesting read-only access

In summary, by combining type families that hide direct memory access operations with a new programming language concept—typestate invariants, NIL succeeds in providing security and a more tightly constrained interaction of modules than heretofore possible.

Support for open systems: Access control

Communications systems tend to be long-lived and must frequently have their function extended to support new protocol algorithms, new communications media, and new execution environments. Hence, it is important for NIL to support the design of modular systems. The fact that a NIL system is portable increases the need for modularity since some users of the software will want to omit some functions and add others of their own.

The role of modularity in a communications system is best illustrated with reference to a communications architecture such as Systems Network Architecture (SNA). SNA partitions the communications function into distinct layers. A simplified diagram of SNA layering is given in Figure 11: the Logical Unit (LU) layer represents a communications end user; the session layer captures end-to-end communications protocols; the Virtual Route layer defines logical pipes between pairs of end points in a network; the Explicit Route layer is aware of actual intermediate nodes in the logical path; and the link manager is responsible for driving the link at each step in the path.

According to the NIL methodology, implementations of SNA products should reflect this layering in their internal structure. The architecture provides a useful organization of the various communication functions, and end users can write software interfacing to the communications layers—thus insulating themselves from the details of lower layers.

If one looks in detail at each individual layer in an architecture, a finer structure becomes visible. Figure 12 illustrates how within the SNA session layer there are several permissible "flavors" of a session, and within the virtual route layer options exist either for historical reasons or because particular products wish to extend or to subset the standard virtual route functions. Any particular communications thread must choose which flavor of protocol to use at each layer. The decomposition of communications function into layers derives its usefulness from the fact that the choices of which flavor to use at each layer are independent. The two end users of a communication thread will have to coordinate their choices so that at each layer they have agreed on matching protocol sets, but they should in general be able to make any session type work over any

Communications systems must frequently be functionally extended.

logical routing protocols using any physical link medium. The problem of selecting a total protocol between end users is factored in this way into a small number of independent choices.

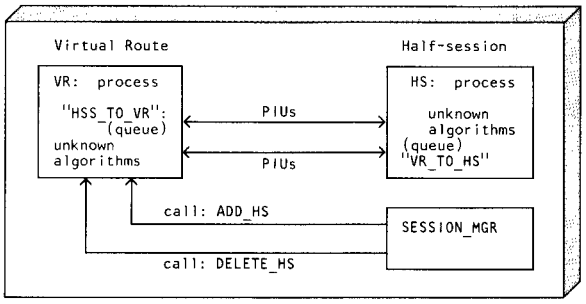
The features of NIL that support this sort of layered and open design are

- The security mechanism, which permits user-written layer options to coexist with "system-supplied" options without the risk that an erroneous user-written layer will corrupt the private data of other layers, and without reliance on special hardware.
- The *strong typing of interfaces* allows the programmer to specify how a particular layer communicates with adjacent layers, without requiring him to know any of the internals of those adjacent layers.
- The *dynamic binding* concept, which permits layer options to be added or removed and connected to their adjacent layers, while the system is running. The combination of statically typed interfaces and dynamic binding supports execution time selection from among a set of "plug-compatible" modules.
- The *access control* functions of NIL, which allow the creator of a layer process to determine that layer's access rights by restricting its potential set of port bindings.

Access control in NIL allows the creator of a layer process to determine its potential set of port bindings.

In an implementation of a layered communications architecture, the software components that are most critical in getting any choice of function at one layer to operate with any choice of function in adjacent layers are the *management functions* responsible for creating new process instances and binding their ports to other already-existing software components. This management function is often the most complex part of a reconfigurable system. We shall illustrate why with an example of a simplified manager to create *half-sessions* in a communications system.

Figure 13 Virtual route to half-session interface



A half-session (HS) is the collection of functions at one end of a communications thread which, when balanced by a matching half-session at the other end, implements the session protocols. A new half-session is created when a particular end user (LU) requests a new communications path to some other LU. We may assume that there is one process per network node called SESSION_MGR responsible for creating new half-sessions. That process, on being presented with a new session request from an LU in its node, will select a particular logical path (VR) to the correct destination on which the session will flow. SESSION_MGR will also determine the type of half-session to install since we assume that several session flavors are available. It then creates new instances of the processes and procedures for that session type. Now before these new programs can start executing, they must be bound to the particular LU component that requested the session and the particular VR component over which the session will flow. The binding actions will typically comprise

- Connecting sendports to receiveports across the layer interface so that messages can be sent between adjacent layers
- Connecting call interfaces across the layer so that synchronous requests can be made to adjacent layers
- Possibly binding the newly installed program to node-wide services
- Initializing other local variables in the newly installed programs

The nature of the connection between an arbitrary VR process and an arbitrary half-session process is illustrated in Figure 13. We are assuming that

half-sessions belong to a single VR at any time, and have one message queue in each direction between it and the VR, while a VR must be capable of "fan-in" of data from many half-sessions and "fan-out" of data into many half-sessions.

In Figure 13, it is shown that all flavors of half-session and virtual route control protocols must share a common interface. Specifically, each HS has a sendport over which it sends messages (called PIUs or "path information units" in SNA) to a VR queue which we call "HSS_TO_VR." For each half-session using the VR, the VR will have a sendport to that half-session's receive queue, which we may call "VR_TO_HS." Since the number of half-sessions per VR varies during the life of the VR, the VR supplies procedures to be invoked to add and delete half-sessions.

If a language is to support the writing of a session manager, it must (1) allow new bindings between half-session and VR processes to be created without exposing the private data structures of either VRs or half-sessions, and (2) allow calls to the add and delete functions even though the specific code being executed is not known until the VR identity is established at run-time.

Every NIL module can be compiled and proofread in isolation.

In an assembly language or PL/S implementation, the binding actions are done by giving SESSION_MGR access to control blocks in all three involved components and having it initialize pointers to make the desired connections. This approach relies on the self-discipline of SESSION_MGR to access only the intended variables in these control blocks. In practice, it usually results in the manager having considerable awareness of which flavors of each of the three layers are actually running. It is then not surprising that SESSION_MGR becomes complex since it has to be able to bind *all possible flavors* of VR with *all possible flavors* of half-session. The architectural decomposition of functions into layers will go to waste if management modules in the implementation have to consider all possible combinations of function in adjacent layers.

Higher-level (but still memory-model) languages such as Pascal and ADA, although they make many advances in other areas, actually make the problem of implementing the SESSION_MGR module for the above example more difficult rather than easier. When CALL statements or abstract SEND operations appear in a program, these languages have a property that requires an answer to be given at compile time to the question of which *program* will be executed in response to that command. But SESSION_MGR's problem is that it must establish a binding so that code in a VR component can send messages up to the half-session component without knowing which of the possible session algorithms is actually running for that half-session. It is the exclusion of procedure-valued variables and the interpretation of abstract operations on the basis of type alone in these languages which creates the problem. In practice, communications systems written in languages such as ADA will probably "escape" and use an underlying operating system primitive whose semantics are not defined in programming language terms to make dynamic connections. Because of the need for escapes, the languages with static binding cannot offer strong guarantees of security.

In NIL the problem is solved by providing both facilities for abstraction and a complete set of language primitives that support all essential operations such as dynamic connections. Processes with private local memory and the ability to accept synchronous calls from other programs allow arbitrary abstractions to be defined and implemented. At the same time, the sendport, receiveport, callport, and acceptport type families permit programs to have statically defined interfaces for ports which can be dynamically connected to ports with matching interfaces in other program instances—a type compatibility check being made at connection time. Every NIL module can be compiled and proofread in isolation without knowing which other modules it is going to be connected to. The language operations in NIL that provide the creation and binding functions are

- *publish* — Makes a receiveport (queue of messages) or acceptport (queue of rendezvous calls) available in a catalog for other program instances to connect to.

- *connect* — Connects a sendport or callport variable to a receiveport or acceptport of matching type using a published entry in a catalog.
- *create* — Creates a new instance of a named module (possibly loading the code), exchanges some creation parameters, and then (if no exceptions have been raised) allows the creator to continue and the created module to start executing independently with its own private memory only; all sharing of objects is achieved by establishing bindings during the exchange of creation parameters. The installed module has access to these parameters for the duration of its "installation" phase.

Figure 14 shows how the part of the SESSION_MGR program responsible for creating a new half-session would be written in NIL. This program sketch deals only with the creation of the half-session and its connection to a virtual route—the connection to a logical unit is similar and is omitted. It is assumed that SESSION_MGR maintains a table of half-sessions. A new row in that table will be created for the new half-session. By some means, the program will have initialized THIS_VR to identify the particular virtual route on which the session will flow. THIS_VR.CAT1 is a field of type catalog in THIS_VR. This catalog will contain an entry with key "HSS_TO_VR" which is the receiveport on which the virtual route receives messages from all its half-sessions. THIS_VR.ADD_HS is a second field of type callport. When a call is made on this port, the virtual route will connect to the port it will use to send messages to the newly created half-session (Figure 15).

It is the integration of these system building constructs into a secure high-level programming language that is the basis for NIL's claim to support modular system design.

Figure 14 Code in SESSION_MGR to create new half-session

```
HS_FLAVOUR_A: module interface(INTF1_TYPE)
        -- this interface makes formal parameters EXPORT_CAT
        -- and VR_CAT accessible during publish and init phases

  MAIN: process
  static    -- the ports to and from VR are declared here
    ( MSGS_TO_VR : MSG1_TYPE sendport,
      MSGS_FROM_VR: MSGT_TYPE receiveport)

  publish phase
        -- this "phase" is executed when an instance of MAIN
        -- is created; it has access to the creation parameters.

    publish MSGS_FROM_VR in HS_FLAVOUR_A.EXPORT_CAT key("VR_TO_HS");

        -- builds a "potential" connection in EXPORT_CAT for
        -- the receiveport from VR into this half session

  init phase
        -- this "phase" is executed just after the publish
        -- phase and has access to the module creation parameters

    connect MSGS_TO_VR in HS_FLAVOUR_A.VR_CAT key("HSS_TO_VR");

        -- actually connects the sendport out of the half-
        -- session to a potential connection in VR_CAT

  main phase
        -- code executed after creation is complete
        -- to handle the session traffic
        . . .
  end MAIN;
end HS_FLAVOUR_A;
```

Figure 15 Code in half-session to bind to a virtual route

```
SESSION_MGR:
   . . .

  block
  declare (NEW_HS: row in HALF_SESSIONS)

    allocate NEW_HS in HALF_SESSIONS key(THIS_HSID);
        -- start a new entry in the HALF_SESSIONS table

    create NEW_HS.COMP1
      module( "HS_FLAVOUR_A",interface(INTF1_TYPE)
              (NEW_HS.EXPORT, THIS_VR.CAT1) );

        -- "HS_FLAVOUR_A" is a string expression naming the module
        -- to be installed (in general not a manifest constant.
        -- INTF1_TYPE is the type of the list of create parameters.
        -- Actual parameters are (i) NEW_HS.EXPORT in which the
        -- half-session will publish potential connections (ii)
        -- THIS_VR.CAT1 which passes potential connections into
        -- the virtual route

    call THIS_VR.ADD_HS(NEW_HS.EXPORT);

        -- this call will cause VR to define a new sendport for
        -- feeding messages to this half session. VR will
        -- "connect" this sendport to potential connection
        -- "VR_TO_HS" in NEW_HS.EXPORT.

    insert NEW_HS;
        -- new entry in HALF_SESSIONS table is complete
        . . .
  end SESSION_MGR
```

Each of these operations has typestate effects and possible exceptions defined by the NIL language reference. It is the integration of these system building constructs into a secure high-level programming language that is the basis for NIL's claim to support modular system design.

Concluding remarks

Experience with NIL. A draft NIL reference manual is available[11] describing all the type families, the

operations on them, the exceptions which may be raised by these operations, and their typestate effects. A compiler for the full language is in use at the IBM Thomas J. Watson Research Center; this compiler generates PL/S code which can be executed in run-time systems under OS/VS2 (MVS), and the Conversational Monitor System (CMS). In the language tool area, work is under way to generalize pragma support in the compiler and to explore an interpretive execution scheme so that NIL systems can run on small machines. NIL is being used by its developers to implement prototype systems for SNA intermediate network node functions and for a set of higher-level intelligent workstation protocols. Although several thousand lines of NIL source code have been written and tested, it is not yet possible to offer any quantification of the software productivity gain achieved.

Other related work. The ideas in NIL are in many ways influenced by current trends in high-level programming languages, particularly ADA, Pascal, and PL/I. The use of type families rather than a memory-model-based encapsulation scheme seems to be quite new and philosophically different from ADA. Although some authors have proposed analysis of typestate-like properties of programs for error detection, to our knowledge there is no previous attempt to formalize typestate and require it to be a general program invariant. The emphasis on dynamic connection primitives in NIL is influenced by the structure of SNA architecture and the meta-implementation with which it is formally specified.

The ideas in NIL are in many ways influenced by current trends in high-level programming languages.

Acknowledgment

The team responsible for developing NIL and its tools consists of W. Burger, M. H. Conner, N. Halim, F. N. Parr, J. A. Pershing, R. E. Strom, and S. Yemini.

Cited references

1. G. D. Schultz, D. B. Rose, C. H. West, and J. P. Gray, "Executable description and validation of SNA," *IEEE Transactions on Communications* **COM-28,** 661–677 (April 1970).
2. M. H. Conner, F. N. Parr, and R. E. Strom, "Portable, secure communications software," *Proceedings of the International Conference on Communications*, Denver, CO (June 1981), pp. 9.4.1–9.4.6.
3. R. J. Cypser, *Communications Architecture for Distributed Systems*, Addison-Wesley Publishing Co., Reading, MA (1978).
4. *Reference Manual for the ADA Programming Language*, United States Department of Defense, Washington, DC (July 1980).
5. B. Liskov et al., *CLU Reference Manual*, Computation Structures Group Memo 161, M. I. T. Lab for Computer Science, Cambridge, MA (July 1978).
6. N. Wirth, "MODULA-2," Bericht Nr. 36, Eidgenoessische Technische Hochschule, Zurich, Switzerland.
7. C. A. R. Hoare, "The emperor's old clothes," *Communications of the ACM* **24,** No. 2, 75–83 (February 1981).
8. B. Liskov and R. Scheifler, "Guardians and actions: Linguistic support for robust distributed programs," *Ninth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, NM (January 1982), pp. 7–19.
9. R. E. Strom, "Mechanisms for compile-time enforcement of security," *Tenth Symposium on Principles of Programming Languages*, Austin, TX (January 1983).
10. "System structure for software fault tolerance," *IEEE Transactions on Software Engineering* **SE-1,** 220–232 (June 1975).
11. *Draft NIL Language Reference Manual*, Research Report RC-9732, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (December 1982).

Reprint Order No. G321-5186.

Francis N. Parr *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Parr is a Research Staff Member in the Computer Sciences Department of the Research Center. He received the Ph.D. in applied mathematics from Harvard in 1974. Before joining IBM in 1978, Dr. Parr was a lecturer in computing at the Imperial College of Science and Technology in London. He is currently manager of the project developing the NIL methodology.

Robert E. Strom *IBM Research Division, Thomas J. Watson Research Center, 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 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 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.