MECHANISMS FOR COMPILE-TIME ENFORCEMENT OF SECURITY

Robert E. Strom
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, N.Y. 10598

ABSTRACT

This paper discusses features of a secure systems programming language designed and implemented at IBM's Watson Research Lab. Two features of the language design were instrumental in permitting security to be enforced with minimum run-time cost: (1) Language constructs (e.g. pointer variables) which could result in aliasing were removed from the programmer's direct control and replaced by higher level primitive types; and (2) traditional strong type checking was enhanced with typestate checking, a new mechanism in which the compiler guarantees that for all execution paths, the sequence of operations on each variable obeys a finite state grammar associated with that variable's type. Examples are given to illustrate the application of these mechanisms.

INTRODUCTION

A system is secure if every program's behavior depends only on its code and its inputs in a manner defined by the programming language semantics. A system is insecure if certain program executions can cause other programs not to behave according to the defined semantics. For example, a system which claims to support independent "virtual machines" is insecure if under particular conditions, a program running in one virtual machine can overwrite data belonging to another virtual machine.

It is potentially easier to provide security for a semantics defined in terms of a high-level language, because compilers can detect and reject illegal programs before they go into execution, and because compilers can avoid generating certain unusual execution sequences which otherwise would have to be anticipated by a run-time security mechanism. Furthermore, compile-time protection can extend to units as small as a single module, whereas such fine granularity is impractical for assembly level programming, at least on machines with conventional architectures.

Enforcement of security has been a goal of a number of high-level languages and proposals, ([DAH 70], [POP 77], [AMB 76], [EGG 61]), and its desirability has been emphasized by a number of writers ([HOA 81]). However, existing compiled languages still suffer from one or more of the following shortcomings:

- There are illegal programs which cannot be detected by the compiler and which if executed may violate security. Examples include "erroneous" programs in ADA (TM), the use of undiscriminated type unions in PASCAL, the dereferencing of uninitialized pointers in most languages.
- In some languages, constructs which are difficult to implement both securely and cheaply are omitted, even though these constructs can be useful in systems programming. (e.g. procedure variables in ADA, explicit deallocation in ALGOL-68, dynamic creation of process, dynamic connection of interprocess communications ports);
- Certain secure language features either require extra execution time overhead or special hardware (e.g. garbage collection, message passing by data copying, checks for dereferencing null pointers, or for dangling references).

The mechanisms proposed in this paper to deal with these problems include:

- The design of a set of abstract types which eliminate direct manipulation of pointers while providing all the useful functions supported by proper use of pointers.
- Augmenting type-checking with typestate checking, a static verification that a variable is in the appropriate state to perform an operation on it. These two mechanisms will be discussed together, since neither appears to be useful for security without the other.

The concepts discussed in this paper are taken from the programming language NIL, which has been designed and implemented at IBM's T.J. Watson Research Center and is being used to develop prototypes for communications subsystems and distributed operating systems. NIL's design objectives include (1) information hiding, (2) support for concurrency, (3) feasibility of secure implementations with minimal run-time cost, (4) efficient implementation on a wide range of machine architectures, (5) concealment of the underlying operating system by supporting process creation, access control, and module interconnection as language primitives. NIL is more fully described in [NIL 82], [PAR 81], [PAR 82], [HAL 82], [PAR 83].

HIDING POINTERS: EXAMPLE

The NIL approach to security can be illustrated by studying the design of the **message** type constructor.

The message type constructor defines individual message types. Each message type consists of a number of fields, which may themselves be of any type. Fields in messages may be named using the standard selected component notation. In this respect, NIL messages are just like records in PASCAL, ADA, or ALGOL-68. However, in these other languages, the user builds records dynamically by using pointers. The pointer has a different name from the record itself and may be assigned to other pointer variables, thereby introducing potential aliasing within a process, or sharing of data between processes. In NIL, messages may likewise be built dynamically, but no pointer is visible to the programmer. It is guaranteed that no two message variables reference the same data. The following operations are applicable to NIL message variables:

allocate obtains resources for an empty message

receive dequeues a message sent from another process over a port (another NIL type constructor).

send FIFO-enqueues a message to another process via a port.

discard disposes of an empty message

It is not permitted to read or update fields of a message after it has been sent. This rule ("destructive" sending), was deliberately chosen over an alternative semantics of **send** in which the sender could retain access to the message data. A language taking an alternative viewpoint must define either (1) that the sender keeps an independent copy of the data, or (2) that both sender and receiver share the message. Both definitions are unattractive: The first alternative requires that **send** copy the message, which on most hardware will be needlessly expensive. In addition, if the language has variables of types for which copying is not supported, these variables could not be passed in messages. The second alternative involves the introduction of shared data, and the danger that both sender and receiver may choose to discard the same message.

By contrast, the "destructive send" semantics allows flexibility in the choice of implementation. A very efficient implementation on a uniprocessor implements a message variable as a pointer to a block of storage containing: (1) the data fields, and (2) a chain pointer used when the message is enqueued on a port. The **allocate** operation initializes the pointer by obtaining storage from a heap or a quickcell list. Access to selected components involves dereferencing the pointer. Sending and receiving involve updating the chain pointer fields without physically copying any data.

TYPESTATE

DEFINITION AND EXAMPLE

The above definition of message semantics makes sense only if the above operations are performed in a particular sequence. For example: a field may not be written until the message has been allocated; a field may not been read until it has been written; a message once sent may not be read, written, or sent again, etc. It is undesirable to attempt to define the results of other orders of execution of the operations, such as **send** of an already sent message or **update** of a field in a message which has already been sent. In fact, the suggested "efficient" implementation of **send** will fail in the event that the same message is sent twice.

In NIL, typestate rules explicitly define the legal operation sequences. Conformity to the typestate rules is checked at compile time as follows: For every type in the language, there is a finite-state grammar (called a typestate grammar) which defines the valid operation sequences on variables of that type. The states of such a grammar are called its typestates. Every variable has a typestate which may vary from statement to statement within the program. **The typestate of each variable is required to be a program invariant at each statement.** Initially the typestate is UNINITIALIZED. Typestates of successive statements can be determined by applying the rules of the typestate grammar to the program. If the typestate on entry to a statement A is known, then the typestate on entry to statement B immediately following A can be determined by applying the typestate transition for the operation associated with statement A. If two statements have the same successor, then the statements must yield identical typestates for all variables. A program is illegal if either (1) it contains a statement containing an operation which is not permitted in the typestate known to hold on entry to that statement, or (2) some pair of statements S1 and S2 yield distinct typestate outcomes but have the identical successor statement.

For example, Figure 1 illustrates the typestate grammar for a typical message type containing two scalar fields, F1 and F2. The typestates are UNINITIALIZED, EMPTY, F1 INIT, F2 INIT, and ALL INIT. The state transitions are labeled with operations on a sample message variable M and its fields M.F1 and

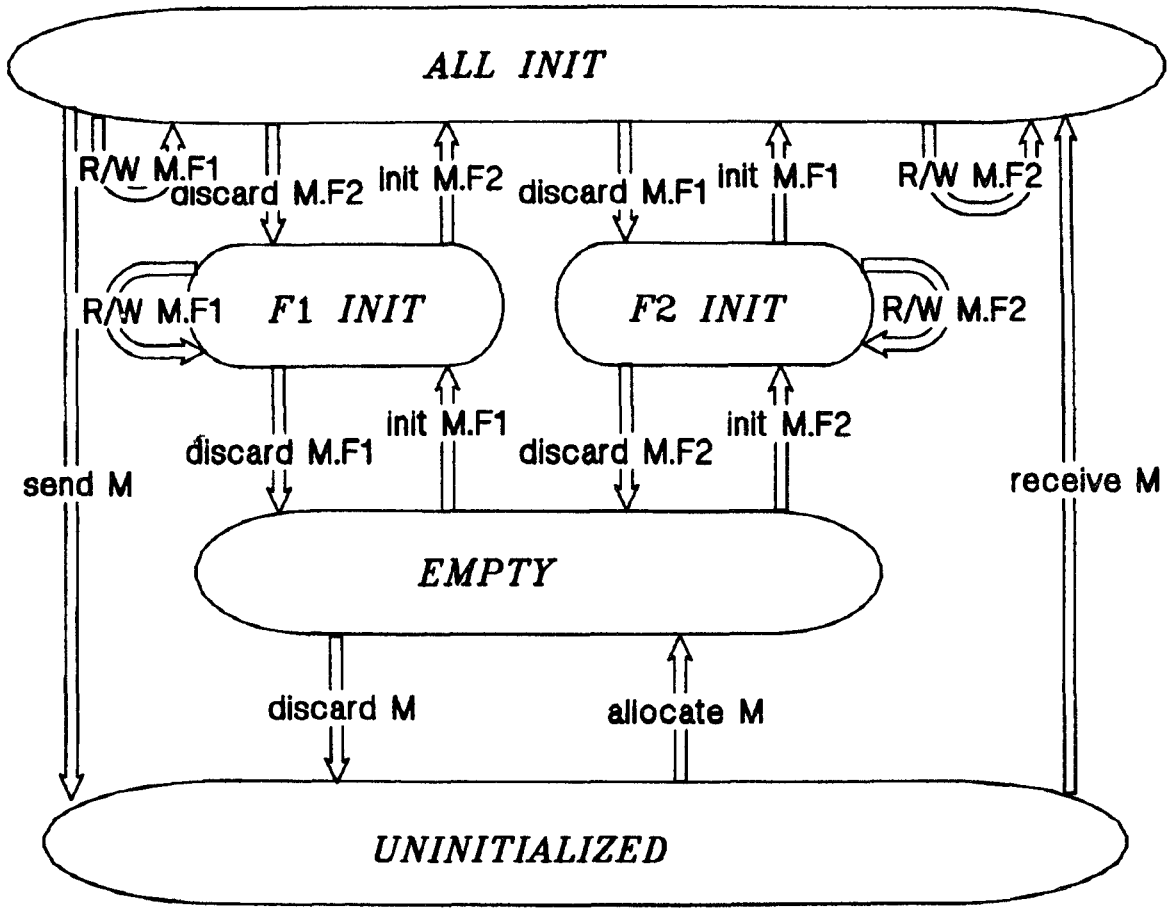

Figure 1. Representative Typestate Diagram: This typical typestate diagram shows the transitions possible for a variable M of a message type containing two fields, F1, and F2, each of scalar type (e.g. integer or string).

M.F2. For example, a message must be ALL INIT before it is sent, and the **send** operation changes the typestate to UNINITIALIZED. Typestate transitions associated with exception outcomes of operations are not shown on this diagram, but are discussed in a subsequent section. It should be noted that **discard** operations are provided for the scalar fields F1 and F2 as well as for message M, even though in a typical implementation space for F1 and F2 is preallocated with the message, and these "discard" operations generate no code. The abstract operations exist in order to permit greater implementation flexibility, and to provide a useful model of process termination, which will be discussed in a later section.

The definition of a sample message type, a sample port type, and the code which declares message and port variables and employs a valid sequence of operations is illustrated in Figure 2. Examples of illegal sequences are illustrated in Figure 3. In the particular representation of messages discussed earlier, these illegal sequences, if not explicitly excluded, would result in the dereferencing of uninitialized pointers, unreclaimed storage, dangling references, or damaged free space lists, each of which can cause a security breach or a failure to reclaim resources. With other choices of implementations of the operations, the effects of the illegal sequences may be different.

```
. .
Porttype is port type of Mestype;
Mestype is message type
   (F1: Charstring,
   F2: integer);
. . .
declare
   (A_message: Mestype,
   A_port: Porttype sender);
. . .
allocate A_message;
A_message.F1 = "hello";
A_message.F2 = 5;
send A_message to A_port;
. . .
```

Figure 2. Typical message type correctly used: The above program segment defines a message type, a port type, and objects of those two defined types. The segment code allocates the message variables, assigns its two fields, and sends the message off to another process. It is assumed that the port variable has been itself initialized earlier in the program.

```
Segment (1)
-----------
allocate A_message;
A_message.F1 = "hello";
A_message.F2 = 5;
send A_message to A_port;
discard A_message;

Segment (2)
-----------
allocate A_message;
allocate A_message;

Segment (3)
-----------
allocate A_message;
A_message.F2 = length (A_message.F1);
send A_message to A_port;
```

Figure 3. Illegal uses of message variables: These examples illustrate program segments which, though not violating any strong typing rules, misuse the message type by executing operations in the wrong order. In the absence of typestate checking, execution of these segments could cause program crashes in typical implementations. Segment (1) discards a message which is in use in another process. Segment (2) overlays a pointer and hence produces an unreclaimable message. Segment (3) accesses an uninitialized string, which if strings are implemented with pointers to the heap, may cause a program check.

RATIONALE FOR INVARIANCE OF TYPESTATES

NIL enforces security at compile time by:

- Forbidding all direct access to pointers, and supporting access to dynamic data only through the message types (and other secure data types).
- Requiring that the typestate of all variables be known as an invariant associated with each statement in the program, and guaranteeing that operations are issued only from correct typestates.

In order for typestates to be program invariants, whenever two or more branches of a program converge, the typestates immediately prior to the join must be the same. Programs which, for example, initialize a variable in the **then** branch of a conditional statement and fail to initialize that variable in the **else** branch, are illegal.

Although security could be guaranteed by run-time checking of typestate, the choice to check typestate at compile time has several advantages:

- It rejects at compile time programs whose erroneous code might not otherwise be detected until they had been widely distributed and used as components of a critical application.
- It avoids the space and time overheads associated with storing, checking, and updating typestate information.
- It rejects certain programs with dubious program structures.

The last point bears additional discussion, since it reflects the NIL designers' bias that software reliability can be gained by restricting the set of programs which a programmer is permitted to write. Consider the program segment in Figure 4 which would be rejected as illegal under the typestate invariance rule, even though a clever compiler might be able to prove that no typestate violation could ever occur. This program creates a half-initialized message whenever X has the value 2, and creates no message otherwise. The paths then join (making the message's typestate ambiguous and therefore illegal according to NIL). Some time later, the half-initialized message is processed, provided it is known to exist, which is determined by checking whether X has the value 2.

```
. . .
if X = 2
  then
    allocate A_message;
    A_message.F1 = "hello";
  end if;

. . .  ---- middle part of program

if X = 2
  then
    A_message.F2 = 5;
    send A_message to A_port;
  end if;
```

Figure 4. Program with ambiguous typestate: Assuming the middle part is well-behaved, this program will never execute an operation from an illegal typestate. However, it will still be rejected by NIL's strict interpretation of the invariance rule. It is the designers' contention that the exclusion of such programs does no harm and could even encourage better program structuring.

It is our contention that there is always a clearer way to write this program, which will provide better guidance to someone trying to modify it, and which will be typestate correct.

Under one interpretation of the program, it is intended to execute the second **then** clause whenever the first **then** clause was executed. Since in this case, the middle part cannot possibly alter the message, it would be safer to combine the two **if** blocks into a single test. The middle part can either be executed after the **if** block, or could be enclosed in a subroutine invoked from both branches. As currently written, it would be fatal to modify the value of X from within the middle part. (A programmer wishing to insert X = X + 1 into the middle part, and to replace the second test by **if** X = 3 deserves, in our opinion, to have his or her program rejected by even the smartest compiler.)

On the other hand, if the middle part of the program contains code which conditionally re-allocates or conditionally discards the message, updating the value of X to reflect its choice, then the programmer would do well to replace the message by a variant (discriminated union) variable, in which one case of the variant contains a message and the other is empty. The program then reads more clearly, since the programmer's test is explicitly asking whether the message exists, rather than asking some other question whose answer is presumed to correlate with the existence of the message.

By replacing direct use of pointers by indirect use via the type constructors message, table, and others not discussed here, and by enforcing typestate invariance, security can be maintained without impacting program efficiency or readability.

PARTIAL ORDERINGS ON TYPESTATES

The interaction between the invariance rule and NIL's treatment of exception handling and program termination lead to the additional requirement of a partial ordering relation on typestate transition graphs.

Consider the program fragment shown in Figure 5. The clause beginning with **on** (Depletion) receives control whenever the exception condition named Depletion is raised within the **begin** block. The Depletion exception is raised whenever storage is not available to perform the **allocate** operation. When Depletion is raised, the message remains in typestate UNINITIALIZED rather than making the transition to EMPTY.

Since typestate must be a compile-time invariant, the program fragment under discussion would be illegal if exceptions generated direct branches to the exception handler. A typestate ambiguity would occur since on the branch from statement [1] message M is UNINITIALIZED and on the branch from statement [2] message M is ALL INIT.

This ambiguity could be eliminated by providing a separate exception handler for statement [2]. This exception handler would discard variable J, the fields of M, and M itself, and then reraise the Depletion exception. There are several difficulties with such a proposal:

- In a language with abstract semantics, nearly every statement has the possibility of raising an exception, even though in some implementations the exception will never be raised. For example, the semantics of strings is flexible enough to permit implementations in which large string values are allocated dynamically from the heap, rather than being preallocated. In such implementations, string assignment could raise Depletion. Requiring separate **begin** blocks to contain the cleanup actions associated with every possible exception could cause the code to become so cluttered with exception handlers that the main path through the program could become obscured.
- If cleanup operations such as **discard** could themselves raise exceptions, then there is a danger of an infinite regress of exception handlers, since each handler would require another handler to deal with the possible failure of one of its cleanup actions.

Both of the above problems are solved by distinguishing between "higher" and "lower" typestates. Intuitively, moving to higher typestates commits machine resources and moving to lower typestates releases resources. The typestate graphs for all possible types can be partially ordered, with UNINITIALIZED a unique state lower than all other typestates. In Figure 1, the typestates closer to the top of the page are the "higher" typestates. The ordering is exploited in the following way:

- Guaranteed Downhill Operations: Between any pair of typestates A and B such that A is higher than B, there exists a sequence of one or more operations to convert an object in typestate A to typestate B. These operations do not require additional operands, may never raise exceptions, and may never deadlock.
- Greatest Lower Bound: Two or more statements may generate control transfers to the same exception handler even though some variable has different typestates in the exception-raising statements. The typestate used on entry to the exception handler will be the highest value which is lower than or equal to the typestates at the exception-raising statements. Typestate lowering operations (called typestate coercions) are inserted automatically between the exception-raising and the exception-handling statements whenever necessary to make the typestates agree.

```
. . .
I = 3;
begin
  allocate M; --- statement [1]
  J = 5;
  M.F1 = "hello";
  M.F2 = 3;
  allocate N; --- statement [2]
  N.F1 = M.F1 || M.F1;
  N.F2 = M.F2 + 10;
  send M to A_port;
  send N to A_port;
on (Depletion)
  call Print ("insufficient storage");
end begin;
```

Figure 5. Program with exception handler: Statements [1] and [2] can both raise the Depletion exception, and send control to the Depletion handler at [3]. Making the typestates invariant at handler [3] would require inserting additional "cleanup" code associated with statement [2]. This example motivates the automatic generation of "downhill" typestate coercions on entry to exception handlers.

In the language subset of our examples (only messages and scalars), the ordering rules are very simple. The typestate of a scalar is either UNINITIALIZED or INITIALIZED. The typestate of a message is either UNINITIALIZED or it is ALLOCATED (ts(1), ts(2), ...), where ts(i) is the typestate of the i-th field. (In the example, the states called mnemonically EMPTY. F1 INIT, F2 INIT, and ALL INIT would be called respectively ALLOCATED (UNINITIALIZED, UNINITIALIZED), ALLOCATED (INITIALIZED, UNINITIALIZED), ALLOCATED (UNINITIALIZED, INITIALIZED), and ALLOCATED(INITIALIZED, INITIALIZED).) The ordering of scalars is simply that UNINITIALIZED is lower than INITIALIZED. For messages, typestate A is lower than or equal to B if:

- A is UNINITIALIZED or
- A is ALLOCATED(tsa(1), tsa(2), ...), B is ALLOCATED(tsb(1), tsb(2), ...), and for all i, tsa(i) is lower than or equal to tsb(i).

The **discard** operation serves as the coercion operation.

In our example of Figure 5, the typestate at the handler for Depletion will have variables J and M UNINITIALIZED. When control is transferred from statement [2], the coercions necessary to discard J, M.F1, M.F2, and M will be generated automatically.

Program termination in NIL involves coercing all declared variables to typestate UNINITIALIZED. These coercions can all be generated by the compiler. The usual hazards involved in generating cleanup code do not exist in NIL: Since no aliases can be generated, the programmer cannot deallocate storage which is being referenced elsewhere under a different name. The programmer may not generate unretrievable storage by deallocating storage containing a pointer, since the typestate rules guarantee that messages may not be discarded until all the fields have been discarded. Any field implemented by a pointer (e.g. a long string, or another message) will automatically be discarded first if the containing message is coerced to UNINITIALIZED. There is no need for an implementation to rely on a garbage collector.

Implementations do have to be careful, however, that "downhill" coercions never raise exceptions --- for example, if **discard** is implemented by a call to a FREEMAIN service, some provision has to be made to avoid failing due to overflow of the call stack, for instance, by using the message itself to hold any temporary storage required by FREEMAIN.

When a process is terminated, all its variables are coerced to UNINITIALIZED after the process has executed its last wishes. The semantics of this coercion depends upon the type --- messages are uninitialized and discarded, message ports are unbound after discarding any waiting messages, rendezvous calls are returned to their caller, processes are terminated. As a result of this semantics, the programmer can know that cancelling a process will guarantee to return its resources within a finite time, and that no "black holes" (unaccessible data) or "white holes" (active uncancellable processes) are possible within a NIL system.

OTHER TYPES

Although typestate checking was illustrated using NIL's message and scalar types, similar ideas are carried out in all the type constructors.

TABLES

For example, the table type constructor is an abstraction for homogeneous collections of arbitrary size --- usually implemented in conventional languages with arrays when the a maximum fixed bound can be determined, and with lists or trees using pointers otherwise. Once again, in NIL the pointers are hidden and access is only permitted through table operations.

The following operations are supported on tables: insertion and deletion of rows, and read-only and read-write access to rows. (Table operations other than those operating a row at a time are not discussed in this paper.) The typestate grammar for rows in tables is shown in Figure 6.

The insertion sequence proceeds as follows:

1. An **allocate** operation is issued specifying key attributes, if called for by the table type definition. If the key does not duplicate an existing key in the table, and if storage depletion does not occur, then the row variable becomes DETACHED EMPTY. The non-key fields become uninitialized but writable.
2. Non-key fields are initialized until the row becomes DETACHED FULL.
3. An **insert** operation places the row into the table. The fields of the row are now no longer accessible, and the row variable itself is UNINITIALIZED. To access the data, a retrieval operation is required.

Deleting a row follows the reverse sequence: the row is first "detached", then the data in the row is discarded, and finally the row is itself discarded.

There are two ways of accessing data without detaching it from the table: **find** with the **read** option causes the data in the row with the selected key to become readable as the value of the row fields. **Find** for update causes the selected data to be readable or writable but not deletable. After examining or updating the data, the programmer issues the **lose** operation. After this operation, the data remains in the table, but is no longer accessible via the row variable, which becomes again UNINITIALIZED. Two new typestates: CONSTANT (read-only permitted), and PERMANENT (read or write only permitted), apply to fields in row variables.

Under certain circumstances a run-time check is required to avoid aliasing. No two distinct row variables may access the same table item unless both row variables are inspecting for read-only. The compiler's typestate analysis can determine when such a check is needed. The example in Figure 7 illustrates one of the rare cases in which a run-time check must be generated. The check is

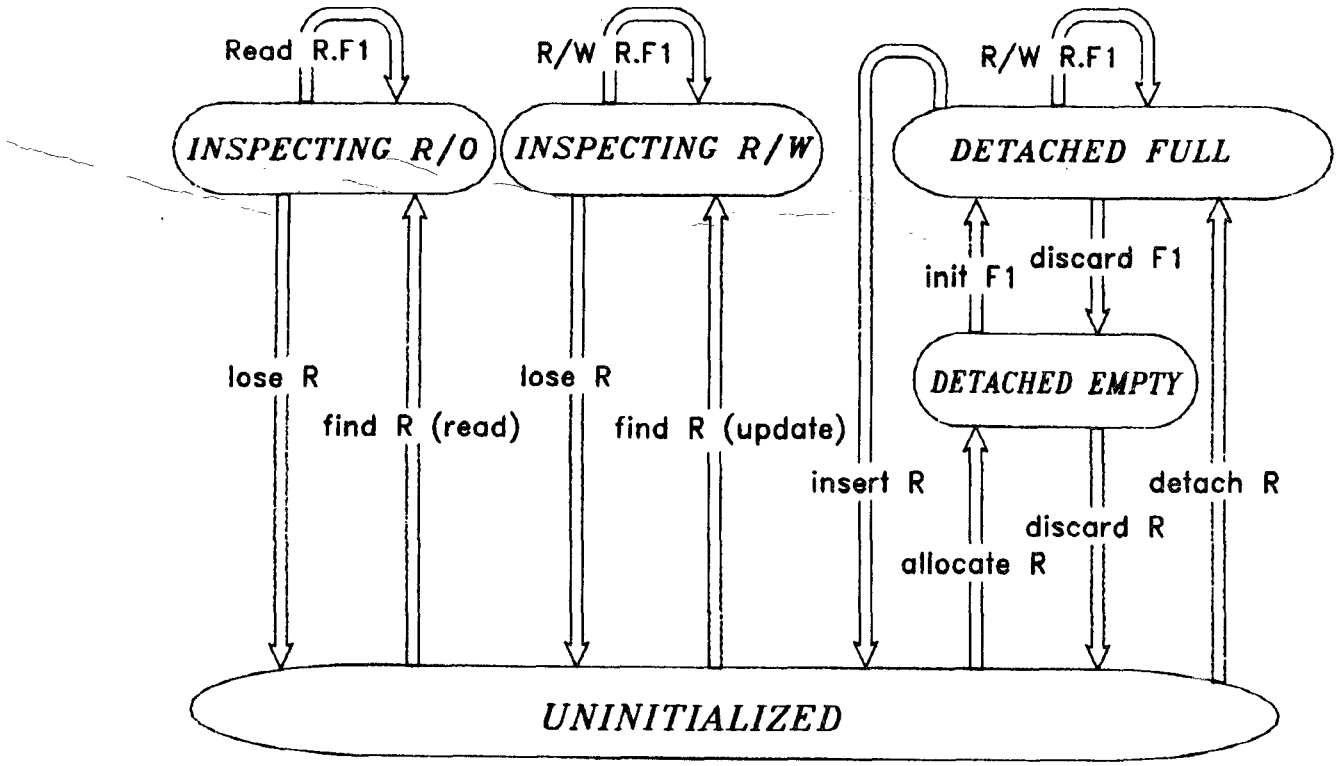

Figure 6. Typestate grammar for Rows in Tables: Possible transitions are shown for a row variable R having a single scalar field F1. The upward bend in the transition for **insert** indicates that an exception may be raised by this operation, whereas the operations shown by arrows pointing directly downward are "downhill coercions" and may not fail.

required on the **find** R2 operation, since another row of the same table has a typestate other than UNINITIALIZED at the time of a **find** for update. In a typical implementation, R1 and R2 will be implemented as pointers or as array subscripts, and the run-time check involves merely checking for equality of the pointers or subscripts.

```
declare
  T Tabletype;
  R1 row in T;
  R2 row in T;
  I: integer;
  J: integer;
  . . .
GetInputs(I, J);
find R1 in T key(I);
find R2 in T key(J) update;
R2.F1 = R2.F1 + R1.F1;
Print(R1.F1);
lose R1;
lose R2;
```

Figure 7. Potential Aliasing: Because R1 is INSPECTING when the **find** operation is performed on R2 for update, a run-time check will be needed to insure that R1 and R2 do not refer to the same row. In the absence of such a check, modifications to R2.F1 could (in some implementations) change the value of variable R1.F1.

VARIANTS

Variants are collections of fields partitioned into disjoint sets called cases. The typestate grammar for a typical variant containing two cases, Red and Blue, each with one field named R1 and B1 respectively, is shown in Figure 8.

The case is known at compile time and is part of the typestate under two circumstances: (1) during the initialization of the variant fields following an explicit allocation either to the Red or Blue case; (2) In the Red or Blue branch outcome of a **select** operation which queries the current state. Two other typestates exist: UNINITIALIZED, in which the variant has no case, and INITIALIZED, in which the case is part of the value, but is not known at compile time. Neither the R1 nor the B1 field may be accessed from this typestate.

PROCEDURE CALLS

Typestate interacts with procedure call semantics in the following ways:

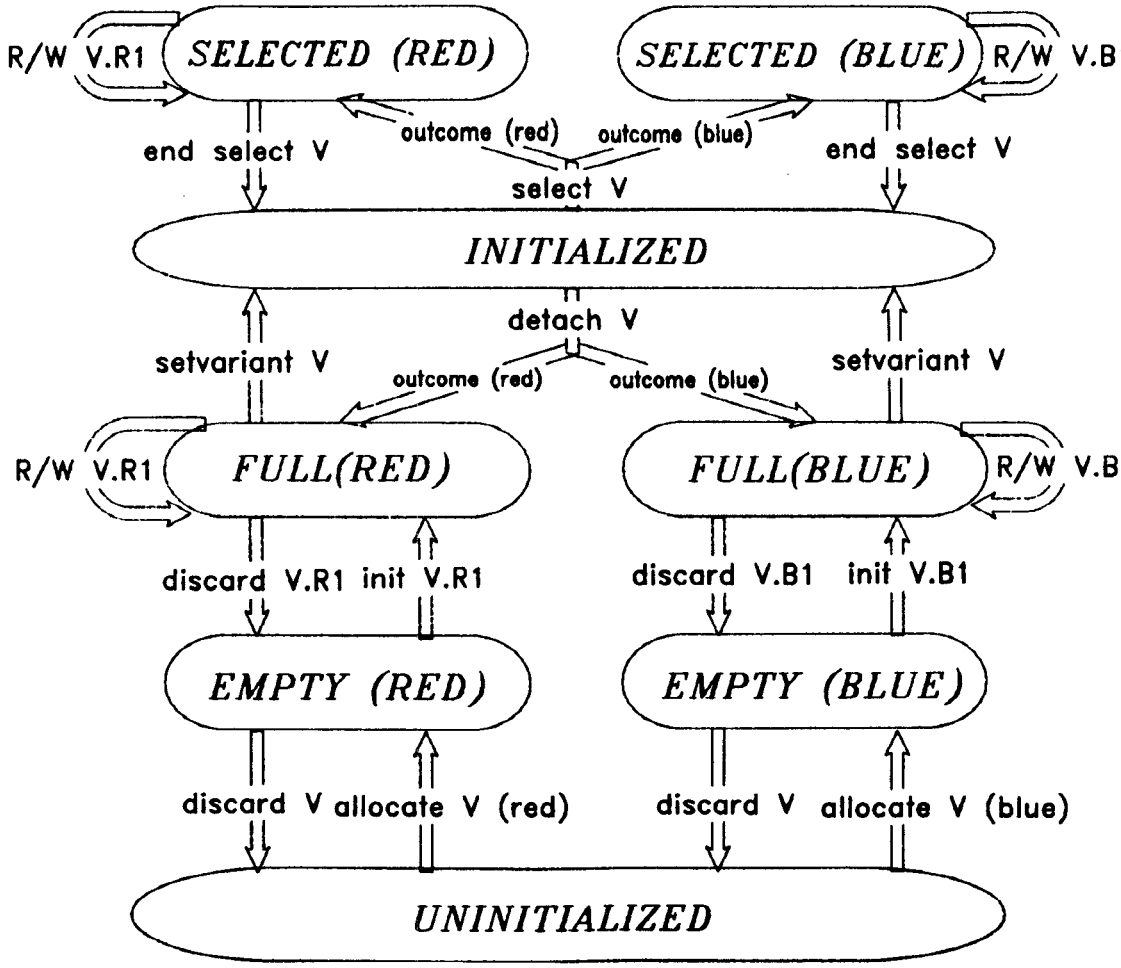

Figure 8. Typestate Grammar for Variants: This graph illustrates the permissible operations on a variable V of a variant type, containing the cases RED and BLUE. The RED case contains a field V.R1, and the BLUE case a field V.R2.

- Procedure call ports (entry variables) may be disconnected dynamically and reconnected to other procedures of conformable type. Tracking each port as either typestate UNINITIALIZED or INITIALIZED makes it possible to insure that uninitialized call ports will not be called.

- Each procedure type definition must contain not only the types of each formal parameter, but also the typestate of each parameter on entry, and on normal and exception returns from the procedure. This specification is useful for documentation, and is essential to permit the compiler to track typestate changes resulting from calling a procedure.

- Typestate information on interfaces helps the compiler to guarantee that procedure calls do not introduce aliasing. (As a result of the complete elimination of aliasing, either call by reference or call by value-result are valid implementations of the semantics.) Two actual parameters with overlapping names (e.g. M and M.A) may not both be passed in the same procedure call unless both formal parameters expect CONSTANT typestate.

RELATED WORK

Other authors have used finite-state models to represent the sequencing constraints inherent in the semantics of a type. (For example: path expressions ([CAM 74]), access path constraints ([CON 79], [KIE 79])). What is different in NIL is that each primitive data type is defined so that the typestate can be a compile time invariant. Typestates are program assertions which are simple enough that they can be automatically generated and proved invariant by a compiler, and yet powerful enough that proving them is adequate to avoid the "erroneous" programs which lead to insecurity.

The notion of partial orderings and "downhill operations" appears to be unique to NIL. It has been of enormous practical value in guaranteeing that aborted or cancelled processes clean up their private resources, including any processes which they may themselves own.

STATUS

A full set of type constructors is available in NIL to meet the general needs of systems programming. A compiler is available for VM/370, and an interpreter design to produce compact code for microprocessor environments is under development.

Prototype systems are being coded in NIL. The application areas to which NIL appears particularly well-suited include:

- "open" layered systems, such as communications systems, in which users of a system may be expected to add their own versions of certain layers, such as screen formatting, protocol conversion, or link control, and in which it is desired to protect vendor-supplied layers from errors in user-supplied layers.
- Highly portable subsystems in which NIL's ability to conceal the underlying data structures and the underlying operating system is very useful.

ACKNOWLEDGEMENTS

Francis Parr collaborated with the author in the original effort to turn NIL from a set of concepts into a viable language. Wilhelm Burger, Mike Conner, Nagui Halim, and John Pershing contributed to the subsequent design effort leading to the current NIL language. Shaula Yemini reviewed the drafts of this paper and contributed significant stylistic and technical improvements, as well as valuable critiques of our language design effort.

REFERENCES

[AMB 76] Ambler, A.L., Good, D.I., Burger, W.F. "Report on the Language Gypsy". ICSCA-CMP-1, The University of Texas at Austin, 1976.

[CAM 74] The specification of process synchronization by path expressions. Lecture Notes in Computer Science 16, New York, 1974.

[CON 79] "Process Synchronization by Behavior Controllers", Ph. D. thesis, University of Texas at Austin, December 1979.

[DAH 70] Dahl, O.-J., Myhrhaug, B., and Nygaard, K., "SIMULA-67 Common Base Language", Norwegian Computing Center, Oslo, Norway, 1970.

[EGG 81] Eggert, P. R., Detecting Software Errors Before Execution, UCLA Computer Science Department, Report No. CSD-810402, April 1981.

[HAL 82] Halim, N., and Pershing, J., "A New Language for Writing Portable and Secure Systems", IBM Research Report RC 9650

[HOA 81] Hoare, C. A. R., "The Emperor's old clothes", reprinted in Comm. ACM, vol. 24, pp. 75-83, February 1981.

[KIE 79] Kieburtz, R., and Silberschatz, A., "Access-Right Expressions", University of Texas, Technical Report, 1979.

[NIL 82] NIL Reference Manual, IBM T. J. Watson Research Laboratory, internal document.

[PAR 81] Parr, F. N., and Strom, R. E., "Portable, Secure, Communications Software", Proceedings, International Conference on Communications, Denver, June, 1981, also IBM Research Report RC 8875.

[PAR 82] Parr, F. N., and Strom, R. E., "NIL: A Programming Language for Software Architecture", Proc. IEEE 6th International Conference on Software Engineering, Tokyo, expanded version also available as IBM Research Report RC 9227.

[PAR 83] Parr, F. N., and Strom, R. E., "A High Level Language for Distributed Systems Programming", to appear in IBM Systems Journal, special issue on communication, 1983.

[POP 77] Popek, G. J., Horning, J. J., Lampson, B. W., Mitchell, J. G., and London, R. L. "Notes on the design of EUCLID", Proc. ACM Conf. on Language Design for Reliable Software, March, 1977.