Typestate: A Programming Language Concept for Enhancing Software Reliability

ROBERT E. STROM AND SHAULA YEMINI

Abstract—We introduce a new programming language concept called *typestate*, which is a refinement of the concept of *type*. Whereas the type of a data object determines the set of operations *ever* permitted on the object, typestate determines the subset of these operations which is permitted in a particular context.

Typestate tracking is a program analysis technique which enhances program reliability by detecting at compile-time syntactically legal but semantically undefined execution sequences. These include, for example, reading a variable before it has been initialized, dereferencing a pointer after the dynamic object has been deallocated, etc. Typestate tracking detects errors that cannot be detected by type checking or by conventional static scope rules. Additionally, typestate tracking makes it possible for compilers to insert appropriate finalization of data at exception points and on program termination, eliminating the need to support finalization by means of either garbage collection or unsafe deallocation operations such as Pascal's dispose operation.

By enforcing typestate invariants at compile-time, it becomes practical to implement a "secure language"—that is, one in which all successfully compiled program modules have fully defined execution-time effects, and the only effects of program errors are incorrect output values.

This paper defines typestate, gives examples of its application, and shows how typestate checking may be embedded into a compiler. We discuss the consequences of typestate checking for software reliability and software structure, and conclude with a discussion of our experience using a high-level language incorporating typestate checking.

Index Terms—Program analysis, program verification, security, software reliability, type checking, typestate.

I. Introduction

TYPESTATE is a refinement of the concept of type in programming languages. Typestate tracking is a compile-time program analysis technique which enhances program reliability by detecting type-correct applications of operations which are *nonsensical* in their current context. In this paper, the term "nonsensical" refers to syntactically well-formed but semantically undefined sequences of program statements.

In the following, we argue that there is a qualitative difference between simply incorrect programs—those which perform computations other than those intended, and "nonsensical" programs—which cannot satisfy *any* meaningful specification and which may produce unpredictable effects if executed. We then show that conventional error detection techniques based on type checking and static scope checking avoid some but not all nonsense. In Section II, we informally present the typestate concept, give examples of its use, and discuss the benefits which accrue from compile-time tracking of typestate. In Section III, we give a more formal definition of typestate, and present an algorithm for verifying the typestate consistency of programs. In Section IV, we discuss the interaction between typestate and other language design issues, such as composite user-defined types, independent compilation, and aliasing. We discuss our experience as designers and users of NIL—a secure programming language incorporating compile-time typestate tracking. Section V presents some conclusions and comparisons with related work.

Manuscript received February 18, 1985; revised September 30, 1985.
The authors are with the IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.
IEEE Log Number 8406865.

A. Type Checking

From the perspective of software reliability, one of the most important properties of the concept of type is that it supports the automatic detection of certain kinds of errors.

The *type* of a variable name determines the set of operations which may be applied to that variable. For instance, if X is of type **real** it is allowed to appear in the context

```
3.14 + X
```

but not in the context

```
Y := X
```

where Y is a variable name of type **boolean.** A language is *strongly typed* if each variable name has a type which can be determined statically. In a strongly typed language, it is possible to check each statement for *type-correctness*, i.e., to check that each operation is applied to operands of the correct type, and to reject any program which is not type-correct.

Type checking detects that sort of nonsense which is independent of the context of an operation relative to other operations. That is, if

```
Y := X;
```

is legal (illegal) at one point in the program it is legal (illegal) from the standpoint of type-correctness *everywhere* in the scope of the declarations of X and Y.

There are other kinds of nonsense, however, which are nonsensical only in particular contexts. For example, assume A and B are both of type **integer.** Then the following program segment:

```
declare
  A: integer;
  B: integer;
begin
  A := 2;
  B := A + 1;
```

makes sense, but this program:

```
declare
  A: integer;
  B: integer;
begin
  B := A + 1;
  A := 2;
```

does not, since the value of A + 1 is undefined if A has not been assigned a value. Therefore, even though A names an integer object, and + is defined for integers, A may be in a state in which the application of + is undefined.

The following is another example of a nonsensical sequence of statements (using Pascal pointer notation):

```
type msgptr = ↑ msg {pointer to a msg variable}
var P: msgptr
P↑.data {the data pointed to by P} := "hello";
```

Since **new** has not been applied, P is uninitialized, i.e., the value of P is undefined and the assignment to P↑.data is nonsensical. If the assignment is allowed to execute, the result will be unpredictable, and will likely result in the copying of the character string into a "random" location in memory, possibly damaging other supposedly "correct" programs running in the same system.

Type-checking does not reject the above examples of nonsense, because the operations being applied (+ and ↑) are valid operations on integers and msgptrs. What makes the above examples nonsensical is the *order* of the statements—that is, + is applied to A when A is in an inappropriate *state* (value not yet defined), and P is dereferenced in an inappropriate state (storage not yet allocated). As with type errors, in a theoretical treatment one can "define away" the errors by means of language extensions, e.g., by adding uninitialized values to the data domain. However since we are interested in enhancing software reliability by automating the detection of errors, we do not pursue this approach. Our goal is to expose nonsense rather than to assimilate nonsense to sense. By providing a narrow criterion for sense, we increase the likelihood that an incorrect program will contain a part which is recognizable as nonsense.

B. Static Scope

Another mechanism which can be used to enhance program reliability is one based on static scope rules in block-structured languages of the Algol family. Static scope rules enable a compiler to detect references to variables which are made outside the lifetime of these variables. For example, in this program segment

```
declare
  A: integer;
begin
  A := 3;
  Print(2*A);
end;
Y := A;
```

the final statement will be detected by the compiler as an illegal reference to variable A, thereby avoiding a potential reference to deallocated storage.

While static scope rules enable automatic deallocation of local block variables, they provide no such mechanism for heap variables. In Ada, for instance, dynamic variables are presumed to endure until the termination of the block in which the access type is declared. More explicit control of the lifetime of heap varaibles is possible only using an unsafe deallocation operation, compromising program reliability.

Here is an example of a set of higher-level operations in which static scope checking is inadequate: Suppose the operation Remove extracts an element from a set, thereby initializing a pointer referring to that element, and Insert performs the reverse procedure, finalizing the pointer after moving the element into a set. Consider the following program fragment:

```
Remove (elem, set0); -- (1)
if P(elem) -- (2) P is a predicate
  then
    Insert (elem, set1); -- (3)
    . . . -- (4)
  else
    Insert (elem, set2); -- (5)
    . . . -- (6)
  end if;
```

Static scope rules do not provide a mechanism for detecting that variable elem may be read at statements (2), (3), and (5), but that reading it before (1), or at (4) or (6) is nonsensical.

C. The Importance of Detecting Nonsensical Programs

Nonsensical executions are the most insidious programming errors, because they can result in arbitrary amounts of damage, and because they are the most difficult to debug. In particular, the erroneous state caused by a nonsensical execution may persist undetected for a long time, and then may manifest itself in the misbehavior of an "innocent" program, far removed from the "guilty" program which caused the error.

The cost of protecting critical components of a system—e.g., an operating system kernel—from the effects of nonsensical programs is high. For instance in the MVS operating system, a call from an untrusted user program to the system kernel must be checked to ensure that all parameters are valid addresses. Other user processes in the same address space must be locked out to prevent them from concurrently overwriting the parameter list while it is being checked. Changing from user to system domain is therefore an expensive operation, whose high cost is economically justified only because failure to check risks the integrity of the entire system.

Although nonsensical execution sequences such as those shown above are considered ill-defined in most procedural languages, e.g., PL/I, Pascal, C, and Ada® none of these languages contain mechanisms which prevent such execution sequences from occurring. Therefore, none of these languages can assure the reliability of any program executing in an environment which might contain nonsensical executions.[1]

A mechanism that automatically detects nonsensical programs at compile-time is extremely valuable because in addition to detecting errors, it ensures that the effect of any program error is confined to the erroneous module. This paper will present typestate tracking as such a mechanism.

II. An Informal Introduction to Typestate

A. Definitions

Typestate captures the notion of an object's being in an appropriate (or inappropriate) state for the application of a particular operation. Each type has an associated set of typestates. An object of a given type is at each point in a program in a single one of the typestates associated with its type.

In each typestate, it is legal to apply some operations of the type, but not others. Thus, for example, **new** may be applied to a msgptr which is in the *uninitialized* typestate (denoted by $\perp$) but not to a msgptr in the *initialized* typestate (denoted by I). Dereferencing, on the other hand, may not be applied to a msgptr in typestate $\perp$, but may be applied to a msgptr in typestate I.

A *partial order* can be defined on the typestates of a given type. Intuitively, a "higher" typestate corresponds to a larger amount of resources allocated to the object, i.e., a higher degree of "initialization." A "lower" typestate can be obtained from a "higher" typestate by discarding some information.

The typestate $\perp$ is ubiquitous, i.e., is defined on all types. $\perp$ corresponds to the state of an object before any operations are applied or following finalization. We organize the typestates of a given type as a *lower semilattice*, that is, a partially ordered structure in which every pair of typestates has a unique *greatest lower bound*. The typestate $\perp$ is the bottom element of the semilattice.

The application of an operation may or may not cause the typestate of its operands to change. For example, **new** causes the typestate of a msgptr to change from $\perp$ to I, but dereferencing does not cause the typestate of a msgptr to change. Thus, each operation of a type has an associated *typestate transition* for each of its operands. The typestate transition is defined by 1) a *typestate precondition*, which must hold in order for the operation to be applicable to that operand, and 2) one or more *typestate postconditions* reflecting the possible typestates of the operand after the operation is applied.

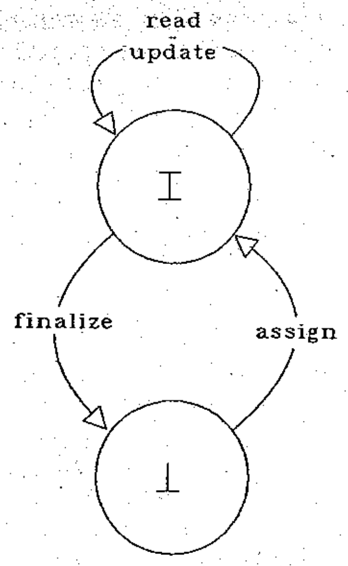

Fig. 1. Typestate transition graph for type **integer:** the scalar type integer illustrates the simplest nontrivial typestate transition graph. There are two typestates: $\perp$ (intuitively "uninitialized") and I ("intuitively initialized").

Some languages allow some operations to have more than one possible outcome. Typically, one outcome is called the *normal* outcome, while the others are called *exceptional outcomes*, signifying the inability to produce a normal outcome because of exception conditions [2], such as unavailability of resources to perform the operation, the value of an operand being outside the domain of definition of the operation, etc. There may exist a different typestate postcondition for the operand for each of the outcomes. For example, if there is no more available storage, **new** may fail with the **Depletion** exception, and the msgptr will then remain in typestate $\perp$. The typestate transitions can be depicted using a graph whose nodes are typestates and whose transition arcs correspond to operation outcomes. Fig. 1 shows the state machine graph for an object of type **integer.**[2]

To track typestate in a program at compile-time, we make typestate a *static invariant* property of each variable name at each point in the program text. That is, if a variable name has a particular typestate at a particular point

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

[1]The Ada reference manual [1] uses the term *erroneous executions* to denote executions which are semantically undefined but which are not guaranteed to be avoided at compile-time or runtime by language implementations.

[2]A typestate graph defines a set of abstract properties, and these properties may be realized in different ways by different implementations. For example, some implementations may preallocate storage for objects on entry to the program or the block, even if the objects are in the $\perp$ typestate. Other implementations may defer storage allocation until initialization. (Typically, large objects or small objects which are to be held in registers are given resources only when initialized, whereas small objects which are stored in main memory are usually preallocated so that several such small objects may be allocated at once.) The implementation of **initialize** and **finalize** will vary according to the storage management strategy chosen, but the abstract semantics and the typestate diagram remain the same.

in the program text, then the corresponding execution-time data object will have that typestate regardless of the path taken to reach that point in the program. Henceforth we will speak of typestate as a property of variable names rather than of data objects.

The typestate within straight-line code can be tracked at compile-time by successively applying the typestate transitions resulting from the application of program statements.

To preserve the static invariance of typestates, we define a rule for resolving the typestate of variable names at points where execution paths merge, such as the beginning of a loop, the end of a conditional statement, or the entry to an exception handler. The rule for determining typestate at a merge statement S is to define the typestate of each variable name as the *greatest lower bound* of the typestates of that same variable name on *all* paths merging at S. Intuitively, the greatest lower bound corresponds to the highest level of initialization of the variable that can be *guaranteed* to hold at S.

For each pair of typestates s_1 and s_2 associated with a given type such that s_1 is higher than s_2 according to the defined partial order, we assume the existence of a *typestate coercion* operation, which lowers the typestate of a variable from s_1 to s_2. The **finalize** operation in Fig. 1 is an example of a typestate coercion. We assume that each such coercion has a single outcome, i.e., it never raises an exception. This assumption is easily satisfied, since coercions never acquire resources.[3]

A program *execution* is *typestate-correct* iff 1) before the application of each operation in the program, each operand v_i has a typestate matching its typestate precondition for the operation, and 2) on termination of a program, all objects declared in the program are returned to the $\perp$ typestate. A program *text* is *typestate-consistent* iff it can be transformed by the addition of typestate-lowering coercions into a program each of whose points can be statically labelled with typestates so that any path allowed by the control flow is typestate-correct.

Whenever it is possible to resolve aliasing statically, as in languages which do not support pointer variables (e.g., Fortran or Algol 60), or in the nonpointer domain of languages which support pointers but do not allow them to refer to stack (static) variables (e.g., Pascal and Ada), it is possible to track typestate by a static examination of a program text. Typestate tracking requires only that procedure call interfaces be augmented to specify, in addition to the type of each parameter, the typestate transition performed by the call.

The restriction against uncontrolled aliasing might appear to exclude many existing languages and applications—in particular, those supporting the dynamic creation of objects by storing their names as values of pointer variables. However, we have embedded typestate within a language, NIL, which supports not only dynamic allocation of objects, but also dynamic creation and interconnection of processes. In a later section, we discuss our experience with NIL, and how the typestate concept influenced the design of NIL.

[3]It is occasionally necessary to exert some care in designing the implementation of downhill operations to guarantee that these are always exception-free. For example, it is not possible for the implementation of a downhill coercion to call library routines in the usual way, because stack overflow must not be permitted to occur.

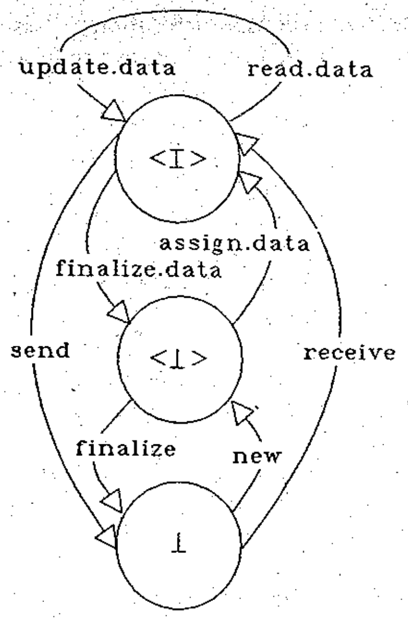

Fig. 2. Typestates for a message with a single character string field "data."

B. Benefits of Typestate Tracking

Although the concept of typestate, like the concept of type, has value independently of compilers, its most valuable aspect is that typestate tracking can be performed at *compile-time* by static examination of a program text. It should be noted that by "compiler," we mean here any algorithm which examines the static text of a program or program module prior to execution, whether or not it transforms this text into machine code. In this sense, we can speak of "compile-time checking" of programs even in interpreted environments.

1) Example: We use the following program example to demonstrate some of the advantages of typestate tracking. We assume that a data type MTYPE has been defined, whose structure is that of a message with a single character-string component, called Data. The operations on objects of type MTYPE are given by the state graph in Fig. 2. (We discuss in a later section how the typestate graph for a user-defined type such as MTYPE might be derived within a programming language.) To simplify the diagram, we have not depicted the exceptional outcomes. We assume in the following example that all exceptional operation outcomes result in the object's remaining in the same typestate it had before the operation was attempted. For example, if the application of **new** results in the StorageDepletion exception being raised, the MTYPE object remains in typestate $\perp$. We further assume that typestate $< I >$ is higher than typestate $< \perp >$, which is in turn higher than typestate $\perp$.

Note: The operation **send** is destructive, i.e., a message must be fully initialized before it may be sent, and once the message data have been sent to a queue, the data are no longer available as a value of the message object, and the message object is now in typestate $\perp$. Were the **send** operation not destructive, the data would exist both in the message object and in the queue after the **send**, which would have entailed either data copying or data sharing.

Consider the following program fragment:

```
 1  declare
 2    M: MTYPE;
 3    Q1: queue of MTYPE;
 4    Q2: queue of MTYPE;
 5  begin
 6    new M; -- Get a new message instance
 7    M.Data := ReadInput( ); -- initialize its field
 8    if F(M.Data) > 0 -- check the value
 9      then
10        send M to Q1; -- send the message
11      else
12        send M to Q2; -- send the message
13      end if;
14 on (StorageDepletion) -- control flows here if
     -- StorageDepletion occurs while attempting
     -- to execute any of lines 6, 7 or 8
     . . .

   end
```

By applying typestate tracking to the above program, the following can be determined.

- The variable name M has typestate $\perp$ at the beginning of the program, since it has been declared but not yet operated upon.
- M has typestate $< \perp >$ immediately after line 6.
- M has typestate $< I >$ immediately after line 7.
- M has typestate $\perp$ at line 13. Line 13 is a merge of the **then** and the **else** branches of the conditional at line 8. Since on both paths, the typestate of M has been reduced to $\perp$ as a result of the **send**, the computed typestate is also $\perp$.
- The typestate invariant for M at the exception handler on line 14 is $\perp$. The exception handler can be reached from lines 6, 7, or 8. When an exception is raised from line 6, M has typestate $\perp$. When an exception is raised from line 7, M has typestate $< \perp >$, since a StorageDepletion exception would mean it was impossible to satisfy the storage requirements for the assignment. When an exception is raised from line 8, the exception results from the inability to satisfy the storage requirements of F, but M is already in typestate $< I >$. The typestate of M computed at the handler is $\perp$, the greatest lower bound of the typestates at the three possible predecessors.

2) Compile-Time Error Detection: The primary consequence of typestate checking is the compile-time detection of nonsensical program sequences. For instance, had we omitted the **new** M statement on line 6, the subsequent assignment on line 7 would be a compile-time *typestate error.* If left undetected, this statement could have severe consequences at runtime as explained earlier.

By tracking typestate, errors not only are detected early, but also can be traced to the particular statement which introduced them, thus simplifying program debugging. Without typestate tracking, a problem resulting from omitting line 6, such as overwriting an arbitrary storage location, could remain undetected for a long time, and would be hard to trace back to the particular error which caused it.

It is important to note that typestate checking also detects errors which are unlikely to be detected by testing, such as those which would occur only on very infrequently executed program paths, since the checks encompass all paths.

3) Compiler-Guaranteed Finalization: In a language which supports dynamic creation of objects whose lifetime is not governed by block structure (usually called heap or dynamic objects), the issue of how to ensure finalization of these objects arises [12], [13]. Finalization may be required both on normal completion of a program, and when a partially executed program unit is terminated because of an exception. In existing languages, one of the following approaches to finalization of dynamic variables is taken:

- provide an explicit operation to finalize the object, e.g., **dispose** in Pascal, **free** in PL/I, and **unchecked_deallocation** in Ada. Explicit finalization runs the risk of accidentally forgetting to finalize some data, of finalizing the data too early, or of finalizing already finalized data—any of which may cause harmful side effects on other modules residing in the same system [12]. Explicit finalizations within exception paths clutter the code module, and make the main path harder to see.
- provide implicit finalization (garbage collection) within the execution environment. Implicit finalization requires additional runtime overhead (the cost of executing an algorithm to determine which objects are reclaimable and which are not), does not scale up to abstract types [12] and cannot always be performed in a timely fashion.

In a language supporting typestate, if any variable names declared within a program unit have typestates higher than $\perp$ upon termination, the compiler can generate the necessary coercions to return those variables names to typestate $\perp$. If an execution can be abandoned at any of several places due to an exception, then the typestates of each variable at the exception handler can be determined by the compiler using the greatest lower bound rule. The compiler can then insert appropriate coercion operations to perform the necessary finalizations prior to jumping the exception handler.

In our example above, the coercions which must be applied to M on the paths to the exception handler are determined by typestate tracking as follows. The typestate for M at line 14 is known to be $\perp$. If the exception is raised during the execution of line 7, after the message has been allocated but before its data field has been initialized, M will have typestate $< \perp >$. A typestate coercion $< \perp > \rightarrow \perp$ (**finalize** M) will therefore be inserted in the exception path from line 7 to the exception handler. If the exception occurs during line 8, after the message has been allocated and its data field has been initialized, then the MTYPE coercions $< I > \rightarrow < \perp >$, and $< \perp > \rightarrow \perp$ (**finalize** M.data; **finalize** M) will be inserted. If the exception occurs on line 6, before the message has been allocated, no coercion will be required, because the typestate will already be $\perp$.[4]

Because typestate tracking allows the compiler to 1) ensure that a finalized object cannot be mistakenly considered accessible, and 2) generate finalization in exactly those places where it is needed, a language designer may provide the user with safe and efficient operations.

4) Enhanced Execution Efficiency: Because typestate errors are detected at compile-time, the compiler implementers can proceed on the assumption that code generation need only be performed for typestate-consistent programs. In many cases, it will be possible to generate more efficient code than if typestate violations were possible, since the code will not have to be as defensive. Once the possibility of nonsensical programs is eliminated, critical systems modules may coexist with undebugged user modules without the need for expensive firewalls. In practice, user and system code will be able to share a single address space, resulting in lowered communications overhead. Since the compiler guarantees timely finalization, it is not necessary to track at runtime which resources have been allocated (as operating systems such as MVS do for files, locks, communication sessions, and other resources), in order to ensure that they are freed when the process terminates. Additionally, it is not necessary to run a garbage collector to reclaim storage. Typestate tracking can therefore significantly reduce execution overhead.

5) Compiler-Enforced Module Isolation: Although typestate checking does not detect errors in the logic of an algorithm, it qualitatively alters the debugging process. Since unpredictable and implementation-dependent side effects resulting from programming errors can no longer occur, it is always possible to debug at the source level, and one never needs to examine "core dumps" except in the case of failure of the hardware or compiler itself. If a module generates wrong answers in response to correct inputs, one can be confident that the error can be found by inspecting that module. Thus, typestate checking makes it possible for a language to satisfy a requirement urged by Hoare [9] and others, that a language enforce the *security* of programs. In nonsecure languages, the wrong answers may be the result of a module's state having been overwritten by "wild stores" from another module having a dangling pointer. To guess which module probably stored the data, it may sometimes be necessary to examine the low-level data representation.

Formal verification methods are motivated by the assumption that if a program has been verified, it will work correctly in all possible environments. In practice, formal verification techniques, e.g., Hoare logic [8] presuppose either that there are no nonsensical programs co-resident with the program being verified, or that none of these programs are able to affect the program being formally verified. Without that presupposition, the semantics of even a simple assignment statement such as X := 2 would have a very complex formulation, since a nonsensical program somewhere else in the system could conceivably overwrite the constant 2! Typestate checking complements formal verification by eliminating nonsensical programs, thereby allowing the simpler proof rules to be applied only to typestate-correct programs.

If it is known that *all* modules of a system are typestate-consistent, then it is possible to prove some properties of an individual module even though nothing is known about the other modules except that they are typestate-consistent. This supports the objective of module-at-a-time program verification [7].

6) The Effect of Typestate on Program Structure: The requirement of typestate invariance constrains the programmer, who is no longer free to code arbitrary control flows, or even arbitrary "structured" control flows (e.g., an IF statement whose THEN clause initializes an object, but whose ELSE clause does not). While some programmers may view this constraint as overly restrictive, our experience has led us to believe that the loss of freedom does no harm in practice, and often does a lot of good.

Consider for example the following program, which can be proved to be typestate-correct for every execution, but which will be rejected at compile-time as typestate-inconsistent because unique typestates cannot be assigned statically to each program point:

```
PROGRAM A

if X = 2
   then
      new M;
      M.data := ReadInput( );
   end if;
F (. . .);
if X = 2
   then
      send M to Q1;
   end if;
```

If we assume the same semantics of **send** used in our previous example (depicted in Fig. 2), the sender can no longer access message M after it has been sent. The program is not typestate-consistent, since the merging of the X = 2 and X ¬ = 2 paths yields a greatest lower bound

[4]In a straightforward implementation of typestate coercions, the same coercion may be repeated in many paths—for example, the **finalize** M coercion appears both in both the path from line 7 and in the path from line 8. To reduce the inefficient use of space due to multiple replications of seldom-executed program sequences, a compiler may apply *downward hoisting* optimizations in which the repeated operations are inserted into a common path.

typestate of $\perp$ for M, making the later **send** illegal. On first glance it might appear that typestate checking here constrains the programmer too much to be practical. However, it is easy to see that the program can be written equally well in the following way:

```
PROGRAM B

if X = 2
  then
    new M;
    M.data := ReadInput( );
    F (. . .);
    send M to Q1;
  else
    F (. . .);
  end if;
```

The two programs have the same semantics, and appear equally easy to write. However, we consider Program B which is completely typestate-consistent to be more readable and more robust for the following reasons.

- Program B makes it clear that every message which is initialized will later be sent. By contrast, in Program A, there is an association between the value of X and whether M has been created. There is a section of Program A in which the programmer must not change the value of X without also changing the state of M. This association has to be *remembered* by the programmer, since it is nowhere explicitly documented, it cannot be automatically enforced, and it is easy to violate if the program is subsequently modified. For example, if program A is modified so that X is incremented, it suddenly becomes nonsensical.
- In Program B, if any exception arises, the necessary finalization is a function only of the point at which the exception occurs. In Program A, the necessary finalization depends both on the point in the program where the exception is raised and, possibly, on the value of X, and therefore cannot be determined statically.

7) Summary: Typestate tracking detects nonsensical programs as well as other programming errors. It supports module isolation by compile-time checks rather than run-time checks. Additionally, it allows the compiler to automatically generate appropriate finalizations when programs terminate or when exceptions are raised. It allows more efficient code to be generated. It has been our experience that the typestate rules constrain programmers to produce better structured programs.

III. A Formal Presentation of Typestate

A. Definitions

We begin with a *strongly typed* language $\mathcal{L}$ containing 1) a set $\mathfrak{T}$ of types (either fixed in advance, or extensible by the programmer via a type definition mechanism), and 2) a set $\mathcal{O}$ of operations (likewise either extensible or not). Each operation $op \in \mathcal{O}$, has a *signature*, $\overline{T}(op) = \langle t_1, t_2, \cdots, t_N \rangle$, specifying the types of its *operands*, and the type of its *result* (if any). That is, for each operand or result position i, there is an associated type $t_i \in \mathfrak{T}$. We shall refer to the vector $\overline{T}(op)$ loosely as the *type* of the operation op. In a language not all of whose operations are pure functions, the distinction between *operands* and *results* disappears. Hereafter, we shall use the term *operands* to denote variables manipulated by an operation, whether the values are changed, read, or both.

We assume each operation in $\mathcal{O}$ has one or more *outcomes:* one normal outcome and zero or more exceptional outcomes.

A program in $\mathcal{L}$ consists of a sequence of statements. Each statement consists of an operation *application*, i.e., a statement is a pair $\langle op, \overline{V} \rangle$, where $\overline{V} = \langle v_1, v_2, \cdots, v_N \rangle$, and each v_i is a variable name.

In a *strongly typed* language, each variable name v has a unique type $t \in \mathfrak{T}$ throughout its scope of definition, which we shall designate as $\mathit{Typeof}(v)$. By extension, if $\overline{V} = \langle v_1, v_2, \cdots, v_N \rangle$ and $\mathit{Typeof}(v_i) = t_i$, then $\mathit{Typeof}(\overline{V})$ is defined to be $\langle t_1, t_2, \cdots, t_N \rangle$. A statement $\langle op, \overline{V} \rangle$ is *type-correct* provided that $\mathit{Typeof}(\overline{V}) = \overline{T}(op)$, that is, for each component v_i in $\overline{V}$, $\mathit{Typeof}(v_i) = t_i$, where t_i is the ith component of the signature $\overline{T}(op)$.

To extend a typed language to include typestate, we define for each type t an associated set $\mathcal{S}(t)$ of typestates. For each operation $op \in \mathcal{O}$, we define for each operand v_i of op, a *typestate transition*, $\langle Pre_{op,i}, \{Post_{op,i,k}\} \rangle$:

1) the *typestate precondition* for v_i, $Pre_{op,i} \in \mathcal{S}(\mathit{Typeof}(v_i))$, defines the typestate that v_i must have in order for op to be applicable, and

2) for each of the different outcomes $Outcome_k$, $k = 1, \cdots, m$ of op: the typestate *postcondition:* $Post_{op,i,k} \in \mathcal{S}(\mathit{Typeof}(v_i))$, specifies the typestate that v_i will have whenever op terminates with outcome $Outcome_k$.

There exists a ubiquitous typestate denoted $\perp$, corresponding to the initial state of a variable name before any operation has been applied. For every type $t \in \mathfrak{T}$, $\perp \in \mathcal{S}(t)$.

The set of typestates for each type is partially ordered by a relation $<$, and forms a lower semilattice, with $\perp$ being the unique lowest typestate. Since the typestates form a lower semilattice, every set of typestates has a *greatest lower bound*. Intuitively, the greatest lower bound typestate corresponds to the highest degree of initialization that can be guaranteed to be satisfied by *any* typestate in the set.

Between any two typestates A and B of a given type t, such that $B < A$, there is defined in $\mathcal{L}$ exists a unique sequence of operations called the *typestate coercion from A to B*, which when applied to an object in typestate A, reduces the object's typestate to B, thereby releasing, or finalizing, some of the resources associated with that object. We require typestate coercions to have the following properties: 1) each coercion operation has a single operand of type t, and 2) each coercion operand has only a single outcome, i.e., coercions cannot raise exceptions and thus always complete successfully.

The typestate tracking algorithm is applied to a *program* represented as a *program graph*, whose nodes correspond to operations, and whose edges connect operation nodes to other operation nodes which are possible successors according to the control flow of the program. There exists one such edge for each of the different outcomes of an operation at a node. Each edge from a node is labeled with its corresponding outcome name. Without loss of generality, we assume that the entire graph has a single entry node, and a single exit node, and is connected.

The particular transformations to be performed to achieve this canonical program representation depend upon the actual source language. In practice, one or more of the following steps may be needed.

- Any overloaded operators must be resolved to a particular operation: e.g., "+" may have to be converted to "IntegerPlus" or "RealPlus," based upon some overload resolution rule which selects a unique operation which satisfies the type-checking rules.
- Any implicit flow-control must be resolved to explicit flow-control: e.g., looping constructs, exception flows, etc.
- All aliasing must be resolved, e.g., by maintaining two mappings: an environment (mapping variable names to locations), and a store (mapping locations to values). For example, if identifiers A and B are aliases, i.e., both reference the same variable, then in the transformation they will be represented by a single variable name.

Note: The language NIL is an example of a programming language in which the above transformations are straightforward. More details of the design of NIL are discussed in Section IV. For the present, we merely assume that these transformations have been made.

Typestate tracking adds *typestate labels* to each node of the program graph. These labels associate each program variable v_i with its typestate s_i at that node. We shall denote these typestate labels by tuples $<s_1, \cdots, s_n>$ where s_i is the typestate of v_i. A labeled program graph is a *typestate-consistent program graph* iff

- on the entry and exit nodes of the graph, all variables have typestate $\perp$.
- at each node N, for each operand v_i of the operation $op(N)$ at N, s_i equals the typestate precondition $Pre_{op(N),i}$.
- for each node N in the graph, if $N_1, N_2, \cdots, N_m$, are the successors of the node corresponding to outcomes $Outcome_1$, $Outcome_2$, $\cdots$, $Outcome_m$, the typestate labelings at $N_1, N_2, \cdots, N_m$ are related to the typestate labelings at N as follows:
  - all variables which are not operands of $op(N)$ have the same typestate at $N_1, N_2, \cdots, N_m$ as they have at N.
  - all variables v_i which are operands of $op(N)$ have a typestate at N_k equal to $Post_{op(N),i,k}$. (If a variable appears more than once as an operand of the same operation, then for any outcome the postconditions for all occurrences of this operand for this particular outcome must be the same.)

A program is said to be a *typestate-consistent program* iff by adding typestate-coercion operations, it is possible to generate a corresponding program graph with typestate labels which is typestate-consistent. Otherwise, the program is said to be *typestate-inconsistent*.

B. An Algorithm for Typestate Tracking

The following algorithm takes a program graph without typestate labels, and inserts typestate labels and coercions to produce a typestate-consistent program graph if this is possible, or else determines that the program is typestate-inconsistent. During execution of the algorithm, the edges of the graph are also labeled with typestates.

We begin by factoring the program graph into 1) a directed acyclic graph between the entry and the exit nodes, and 2) a set of "back edges" representing "backwards branches." (For most structured programming languages, these graphs will be "reducible," in which case there will be a unique such decomposition.)

Pass 1:

1) Label the entry node with typestate $\perp$ for all variables.

2) While possible, select an unprocessed node N all of whose entry edges in the directed acyclic graph have been labeled (but which may have unlabeled back-edges). Let $op(N)$ be the operation at node N.

3) Label node N as follows: For each variable name v, compute the *greatest lower bound*, $s_{glb}(v)$, of the typestates of v on each of the entry edges into N. Then, if v is not an operand of $op(N)$ then its typestate at N is simply equal to $s_{glb}(v)$. If v is an operand $op(N)$, we distinguish two cases:

- if $Pre_{op(N),i}$ (the precondition typestate of that operand required for applying $op(N)$ is greater than the computed $s_{glb}(v)$, the program is typestate-inconsistent;
- if the required precondition typestate is less than or equal to the computed $s_{glb}(v)$, then for each entry edge labeled by a typestate of v higher than the precondition tyestate of v at N, insert a coercion operation on v to lower its typestate to $Pre_{op(N),i}$.

If a variable name appears in more than one operand position of the same operation, all occurrences must have the same precondition typestate.

4) For each edge exiting from N to N_k (corresponding to $Outcome_k$), including back edges, label the edge in the following way: for each variable name v, 1) if v is not an operand of op_N, its typestate in the label will be the same as its typestate at node N; 2) if v is an operand of $op(N)$, say the ith operand, its typestate in the label will be equal to the postcondition typestate associated with that outcome, $Post_{op(N),i,k}$. If a variable name appears as more than one operand of the same operation, for each outcome all occurrences will have the same postcondition.

The above iteration is guaranteed to terminate with each node labeled, since the directed acyclic graph is fully connected. It is now necessary to make a second pass through all nodes to deal with backward branches.

Pass 2:

1) While there remain nodes not processed in Pass 2, select a node N all of whose predecessors in the directed

acyclic graph have already been processed during Pass 2. (The entry node trivially satisfies this condition).

2) Examine all the back edges into node N. For each such edge, compare the typestate of each variable name v on the edge ($s_{edge}(v)$) to the typestate at N ($s_N(v)$). If for some variable name v, $s_N(v) \leq s_{edge}(v)$, then no change is made to $s_N(v)$, but a coercion to lower the typestate from $s_{edge}(v)$ to $s_N(v)$ is inserted if needed. If for some variable name v, $s_{edge}(v) < s_N(v)$, then $s_N(v)$ must be lowered to the greatest lower bound of $s_N(v)$ and $s_{edge}(v)$. If, as a result of inserted coercions, an operand is now in a typestate lower than its precondition for $op(N)$, the program is typestate-inconsistent.

3) After all nodes have been examined on Pass 2, the typestate coercions necessary in order to reduce the typestate of all variables to $\perp$ are inserted before the exit node.

The above algorithm processes each edge exactly twice, and hence is linear in the number of edges in the program graph.

C. A Modified Algorithm which Handles Overloading

In practice, it may be necessary to employ a variation of the above algorithm which takes into account *overloaded operations* intended to be disambiguated during typestate checking.

Many languages have operators such as "+" which are overloaded and cannot be disambiguated until typechecking occurs. A similar possibility for overload resolution occurs with respect to typestate checking. The most common example of such overloading is the *assignment* operator, as in "A := B". When A has typestate $\perp$, assignment means *initialize* whereas when A has typestate I, assignment means *update*. The two operations are distinct, having different typestate preconditions, although they have the same postcondition. Similarly, operations on structured objects are technically overloaded, since the typestate precondition for an operation may be independent of some of the components of the structure. For example, in the record with components "a" and "b" of type integer above, an assignment to component "a" can be made whether component "b" has typestate I or $\perp$. (See Fig. 5.)

If the typestate computed on the second pass is strictly lower than the typestate computed on the first pass, operations applied to that variable will no longer satisfy their typestate preconditions. When there is no overloading, it is obvious that the corresponding program is typestate inconsistent and must be rejected. However, if this occurs when there is overloading, the effect of a typestate relabeling will be to try an alternative resolution of the overloaded operation. Provided that all overloadings of a given operator differ only in the precondition, the revised algorithm still needs only two passes, since Pass 2 merely propagates typestate labels generated during Pass 1.

Even more complicated overloadings are possible, i.e., we can allow different overloadings to have different postconditions. A sufficient condition that the above algorithm terminates is that the overloadings are monotonic—i.e., that if two overloadings A and B of an operation exist such that A's operand's preconditions are each lower than or equal to B's operand's preconditions, then A's postconditions will be lower than or equal to B's post conditions. Each successive pass may lower the typestates of some variables, but because the set of the typestates has a bottom element, the algorithm must terminate.

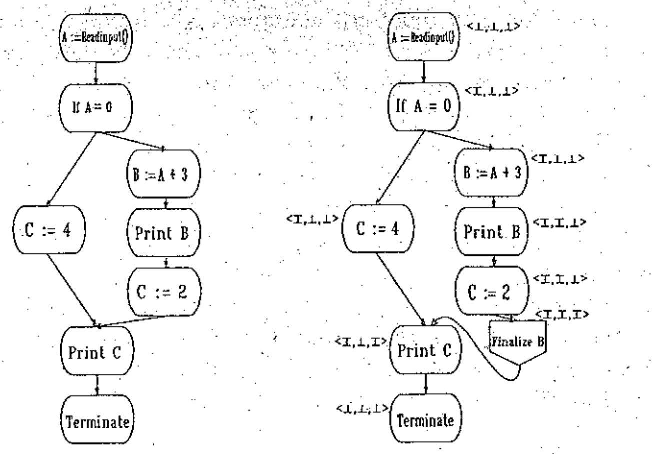

Fig. 3. Typestate checking for a loop-free program graph. The program graph is on the left, and the typestate labeling produced by typestate checking is on the right. The typestate labels contain the typestates of variables A, B, and C, respectively.

D. Examples

A simple example of the application of the above algorithm is given by tracking typestate in the following program:

```
declare
   A: integer;
   B: integer;
   C: integer;
begin
   A := ReadInput();
   if A > 0
      then
         B := A + 3;
         Print (B);
         C := 2;
      else
         C := 4;
      end if;
   Print (C);
end
```

Fig. 3 illustrates the program graph before and after typestate checking. In this example, type integer is presumed to have two typestates $\perp$ and I as in Fig. 1. Here, we are ignoring overloading, and will simply assume that the assignment statement means initialization only. The operation **finalize** is the coercion from typestate I to typestate $\perp$.

Notice that the typestates of A, B, and C, respectively, on the arc into the Print statement which came from the THEN path are < I, I, I >, but the typestates on the other arc coming from the ELSE path are < I, ⊥, I >. Variable B's typestate at the Print statement therefore resolves to ⊥—the greatest lower bound. A **finalize** operation (coercion) for B is inserted so that in the modified graph both paths yield the typestate < I, ⊥, I >. Similarly at the end of the program, variables A and C are finalized.

Notice further that if the operand of the second Print statement had been B rather than C, the program would be rejected as typestate-inconsistent, because the precondition for applying Print is that the operand be I, and B is ⊥.

In the above example, the entire program graph contained no backward branches, and therefore all typestate resolution was completed after a single pass. Suppose instead that a similar program fragment were embedded in a loop:

```
A := ReadInput();
B := ReadInput();
Print B;
while (A < 1000) repeat
   if A > 0
      then
         B := A + 3;
         Print B;
         Finalize B;
         C := 4;
      else
         C := 6;
         Finalize B;
      end if;
   Print C;
   end while;
   . . .
```

Here we show only a fragment of a complete program. In this example, we are assuming that assignment is overloaded. The typestates at the top of the loop evaluate on the first pass to < I, I, ⊥ >. The typestates after the END IF evaluate to < I, ⊥, I >, The discrepancy in the typestate of variable C is resolved by inserting a finalization of C in the branch back to the top of the WHILE loop. A one-pass algorithm could have inserted this finalization, since the top of the loop is guaranteed to be correctly labeled before the bottom of the loop. The discrepancy in the typestate of B, however, accounts for the need of a second pass. As a result of evaluating the greatest lower bound, the label on the WHILE loop test is reduced from < I, I, ⊥ > to < I, ⊥, ⊥ >, and this change must be propagated through the statements of the loop. Because the assignment operator is overloaded, the statement B := A + 3;, which was believed on Pass 1 to be an **update** operation on B, is now discovered on Pass 2 to be an **initialize** operation on B. Fig. 4 illustrates the labelings performed during the two passes of the checking algorithm applied to this program.

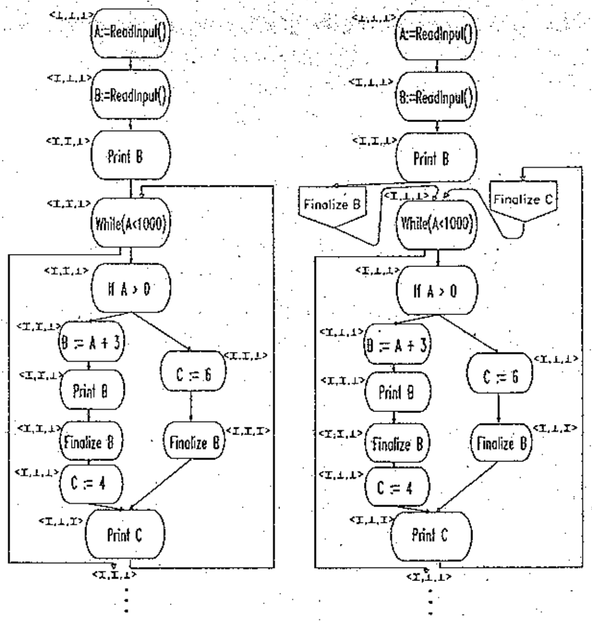

Fig. 4. Typestate checking of a program graph containing a loop. The two diagrams show the typestate annotations of the nodes of the program graph at the end of Pass 1 and Pass 2, respectively.

IV. Typestate and Language Design

Typestate interacts with other features of programming languages. In this section we discuss particular features of languages and how their design is affected by the typestate concept. We draw heavily on our actual experience in designing NIL. We conclude the section by discussing our experience as designers and users of NIL.[5]

A. *Typestates of User-Defined Types*

In languages which support user-defined types by providing type constructors, the typestate transitions for user-defined types can be defined by associating a *typestate construction rule* with each of the language's type constructors. The typestate construction rule defines the typestates of a constructed object as a function of the typestates of each of the objects from which it is constructed. Typestate construction rules may be defined for any type constructor, including recursive types.

For example, one type constructor found in many languages is the **record.** Each record type defines a structure consisting of an n-tuple of components where each component $i = 1, \cdots, n$, is associated with some type t_i.

Assuming that the operations on a record type are:

• **new,** which creates an empty n-tuple of uninitialized components (As discussed previously, the implementation may choose to pre-allocate storage even before **new** is applied),

[5]Other individuals who have participated in the design, implementation, and prototyping of the original NIL system include: Mike Conner, Nagui Halim, Jim McInerny, Dan Milch, Francis Parr, and John Pershing.

• **finalize,** which destroys an empty n-tuple,

as well as the following *derived* operations: if type t_i of the ith component supports an operation *op*, then the record type supports an operation *op.i.* denoting the application of *op* to the ith component of the record.

The typestates for the derived record type are:

• $\perp$ (the state prior to applying new or after applying **finalize**)

• $<s_1, s_2, \cdots, s_n> \in \mathbb{S}(t_1) \times \mathbb{S}(t_2) \times \cdots \times \mathbb{S}(t_n)$—the set of n-tuples of component typestates.

The ordering relationships and coercions for records are derived as follows:

• The typestate $\perp$ is lower than any typestate $<s_1, s_2, \cdots, s_n>$. The coercion sequence from any typestate $<s_1, s_2, \cdots, s_n>$ to $\perp$ consists of two steps: 1) for each i such that $\perp < s_i$, apply the operation to coerce component i from typestate s_i to $\perp$. The resulting typestate will then be $<\perp, \perp, \cdots \perp>$. 2) apply the operation **finalize** to the record.

• Typestate $r = <r_1, r_2, \cdots, r_n> \preceq s = <s_1, s_2, \cdots, s_n>$ iff for each i, $r_i \preceq s_i$. Of course, $r < s$ iff $r \preceq s$ *and* $r \neq s$. The coercion operation consists in applying for each i the coercion operation to lower component i from typestate s_i to typestate r_i.

The projection onto a particular type of the typestate transition for each of the operations which has an operand of that type can be depicted as a directed graph, as mentioned earlier.

Fig. 5 illustrates the typestate graph for a record with two components with field names "a" and "b", both of type **integer.**

Fig. 6 illustrates the typestate graph for a variant. It can be determined from the typestate graph that it is forbidden to access any field of an initialized variant (typestate I) without first performing the **inspect** operation. Other examples of typestate graphs appear in [14].

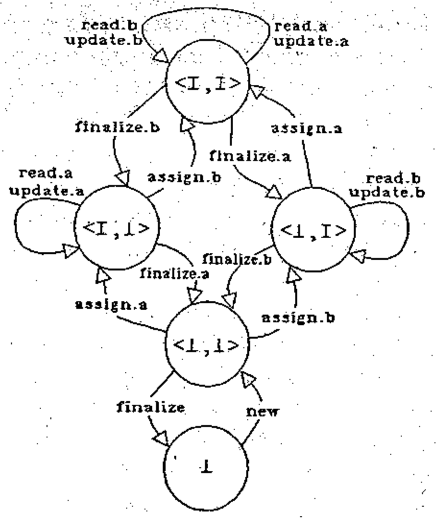

Fig. 5. Typestates of a constructed type: a record with two fields, a and b, each of scalar type.

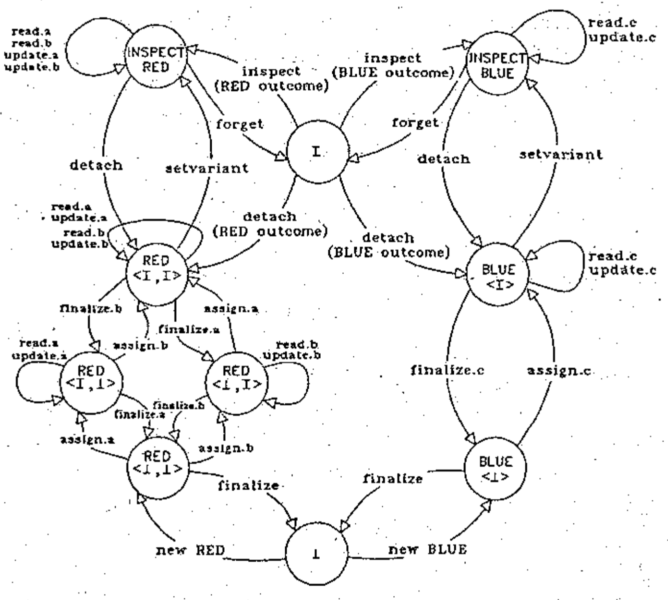

Fig. 6. Typestates of a variant: an object with two alternatives, RED with fields a and b, and BLUE with field c, all of scalar type.

B. Typestate Checking and Independent Compilation

In Section III, we described typestate checking as applying to entire programs. However, it is also possible to *independently* check individual modules of a large program for typestate consistency in such a way that if each module proves to be typestate-consistent, then the complete program, when linked and executed, will be typestate correct.

We illustrate independent typestate checking with respect to a common construct for intermodule interaction—calls.

By *independent* typestate checking of modules, we mean that the text of the calling program is not required to be available when compiling the called program, and conversely that the text of the called program is not required to be available when compiling the caller.

The following discussion applies equally well to either procedure calls or entry calls as in e.g., Ada or NIL. We assume that the language $\mathcal{L}$ has a construct for calling an entry (procedure) in another module and passing parameters, such as

```
call E (P1, P2, . . . Pn);
```

where E identifies the entry and Pi are the actual parameters of the call. We assume that $\mathcal{L}$ also has constructs for accepting and subsequently returning calls on a particular entry, such as:

```
accept (FP1, FP2, . . . FPn) on E;
. . . --body of call
return (FP1, FP2, . . . FPn) [exception (Ename)];
```

where FPi are the formal parameters of the call, and Ename is the name of the exception outcome, if any. The acceptance of the call binds the formal parameters to the actual parameters. The accepting process may operate upon the formal parameters until it issues the **return** operation, at which point the parameters are passed back to the caller and the caller resumes either normally, or possibly with an exception outcome if so specified by the **return** operation.

Note: The syntax shown is for rendezvous calls similar to those in Ada. The same analysis applies to ordinary procedure calls if one assumes that each procedure body begins by implicitly issuing the **accept** operation to receive its parameters.

To achieve independent typestate checking in the presence of calls, we introduce the concept of *interface definitions*—separate modules which specify the assumptions shared by the calling and called modules. The information which we require in an interface definition is precisely the information required for a primitive operation of $\mathcal{L}$: namely, the type of each parameter to the call (operands), the set of outcomes of the call (i.e., normal and exception outcomes), and for each parameter, its typestate transition for the call, i.e., its typestate precondition, and its typestate postcondition for each of the outcomes of the call.

The interface definition may additionally specify *typestate restrictions*. A typestate restriction removes edges from the typestate graph. For example, we may wish to forbid the called program from updating or from finalizing an object which is passed to it. We allow typestate restrictions provided that between any two typestates in the set of typestates reachable from the typestate precondition of the call, the edges representing the coercions to the greatest lower bound have not been removed.

The modes **in, out, inout,** defined on procedure interfaces in languages such as Ada and Pascal can in fact be viewed as special cases of the above formalism. For example, **in** means that both the pre- and postcondition typestates are I, and that all operations except reads are disallowed. The designation **out** means that the precondition typestate is $\perp$, and the normal outcome postcondition typestate is I. To be able to perform full typestate tracking, it would be necessary to extend the definition of **out** to specify the typestates on exceptional outcomes as well. In NIL, interface definitions include the set of exceptions and the complete typestate transitions.

Provided an interface definition for the call has been supplied, one can compile the calling and the called module *in any order*. During compilation, the call is treated exactly as a new primitive operation of the language. When compiling the caller, it is checked that prior to the call, each parameter is in the correct precondition typestate, and it is assumed that on return from the call, each parameter is in the postcondition typestate associated with the corresponding outcome of the call.

When compiling the called program, the information checked and the information assumed now reverse roles. The entire formal parameter list is in the $\perp$ typestate prior to the **accept;** after the **accept,** the formal parameters are assumed to have typestates according to the precondition specified on the interface. The precondition for **return** requires that all formal parameters be in the appropriate postcondition typestate defined by the interface description. After the **return** operation, the formal parameter list is once again in the $\perp$ typestate.

Notice that when compiling the calling program, the compiler is ensuring that the program is typestate-consistent with respect to *any* typestate-consistent called program compiled against the same interface. Conversely, when compiling the called program, the compiler is ensuring that the program is typestate consistent with respect to *any* typestate consistent calling program compiled against the same interface. Consequently, there may exist many different independently compiled calling and called programs which use the identical interface. The decision as to which calling program is to be bound to which called program may be made arbitrarily late. Thus typestate checking supports *compile-time* secure dynamic binding, which is very important for systems programs, which typically involve dynamic loading and linking of independently written modules.

It is possible for the typestate tracking algorithm to perform overload resolution for procedure calls, just as for primitive operations such as **assign.**

C. Pointers and Aliasing

Languages that allow unrestricted pointer assignment do not support tracking typestate at compile-time because the mapping between variable names and execution-time objects is not one-to-one. As a result, a typestate change resulting from applying an operation to an object under one name will not be reflected in the typestate of other variable names referring to the same object.

Another language feature which may interfere with typestate tracking is data sharing by multiple concurrent processes. If a language allows concurrent processes to share data, then unless other facts about synchronization are known, typestate-changing operations in different processes may be interleaved in arbitrary ways, making compile-time tracking impossible.

However, it is possible to provide the benefits of typestate tracking while still supporting important language features such as concurrency and dynamics.

In NIL, concurrency is supported by means of a *process paradigm*. In the process paradigm, there may be any number of independently executing processes in the system, but each data object is owned by exactly one process at a given time. Processes communicate by sending messages and making rendezvous calls over *ports*, rather than by sharing variables. Objects may change ownership by being sent in messages between processes, but the semantics of send is destructive, thereby preserving the single owner rule. Thus, typestate tracking can be applied to the program text of each process.

Dynamics is achieved in NIL by raising the level of the language to a more abstract set of operations which do not expose pointers. (Pointers are typically reintroduced by the compiler in the encapsulated implementations of the language primitives). In fact, NIL supports a higher degree of dynamics than other languages of its kind such as

Pascal and Ada, because in addition to having all data allocated dynamically, process and procedure bodies are selected and instantiated dynamically under program control [17].[6] In most other languages, intermodule bindings must be fixed statically before execution, when all the modules are linked.

An example of hiding pointers is in allocating records or variants. A record is allocated by the statement

```
new R;
```

where R is a variable name whose declared type is Rtype, rather than by the pointer assignment

```
Rptr := new Rtype;
```

Dynamically growable structures are implemented either with *relations*, which support **insert, delete,** and **find** operations, or by using recursive variant types as in the following NIL definition of Lisp S-expressions:

```
Sexpression is variant
  case (NIL)
  case (ATOM)
    Printname: Name;
  case (PAIR)
    Car: Sexpression;
    Cdr: Sexpression;
  end Sexpression
```

Relations are more abstract than arrays, linked lists, and other more traditional data structures used to implement aggregates, although the compiler may use traditional data structures to implement relations. They are safer than arrays or linked lists, because arrays might have uninitialized "holes," and linked list pointers can be operated upon in unsafe ways not detectable at compile-time.

D. Experience with Typestate in NIL

The concept of typestate was first included in the programming language NIL [14]–[16], which was originally developed as a language for prototyping systems software.

NIL was designed to support programming large long-lived systems. To support modularity NIL provides a processbased paradigm [17], in which loosely coupled modules communicate over point-to-point ports, rather than by sharing data. To support portability, NIL provides very high-level, machine-independent primitive data type constructors.

The following design decisions of NIL made the incorporation of compile-time typestate tracking particularly easy:

- Data are never shared between processes.
- There are no directly manipulable pointers, nor any other ways to generate aliases.
- NIL contains a recursively composable set of type constructors, which allow the straightforward generation of typestate graphs, coercions, and the pre- and postconditions of derived operations for constructed types.
- NIL's control flow is the standard "structured" control flow augmented by exception handling. The creation of the program graph is therefore straightforward, as the only back-edges result from loops.
- The interface type definitions for port types supporting the **call** operation contain exactly the information necessary to support typestate checking with independent compilation.

A compiler for NIL, incorporating typestate checking, was completed in 1982, after which time a significant amount of NIL code (about 25 000 lines of code constituting several hundred modules) was developed for a prototype communication system.

In our experience, we have not encountered any situations where the most natural, readable, and robust expression of an algorithm was one which was typestate-inconsistent. On the contrary, the habit of balancing typestate effects in alternative constructs often helped organize our thoughts about an algorithm and enhanced the readability of the result. Additionally, at one time we examined several hundred modules of a communications subsystem written in a PL/I-like language, manually checking for typestate errors. We discovered that all typestate-inconsistent programs in our sample had bugs, which led us to believe that the probability of violating typestate but nevertheless producing safe code was low.

The use of interface definition modules which documented typestate transitions of all parameters gave us very detailed documentation which enabled developers of different modules to work independently, relying on the compiler to keep the code consistent with the interface, rather than relying on knowledge of internals of modules developed by others. By contrast, in other languages such as Pascal and Ada, interfaces are less rigidly documented and enforced:

- the obligation to initialize **out** parameters is not enforced;
- a programmer has no means of specifying that a caller passes an object to a called program with the intention that the object be retained by the called program and not returned (that is, has a precondition of I and a post-condition of $\perp$).
- there is no way to specify whether parameters are initialized or not in the event of different exception outcomes from the call.

The most dramatic result of our experience with NIL was the fact that modules that were debugged independently (unit-tested) did not subsequently fail when integrated with other modules. This was not the case for the NIL compiler itself, which was written in PL/I, where debugged parts of our compiler frequently "broke" as a result of changes to other modules. With NIL, we acquired a confidence in the locality of the programming errors which made a qualitative difference in our approach to debugging.

[6]Preallocation and prebinding can always be performed as optimizations when the necessary information can be statically determined.

V. Summary

A. Related Work

Scope rules and strong typing have been extensively used for enhancing program reliability. Using these rules, it is possible to 1) detect at compile-time references to objects or program units which will be inaccessible at runtime (e.g., undeclared or not visible within the referencing scope); 2) support dynamic allocation of variables on the stack and ensure their finalization without supporting unsafe explicit storage deallocation; and 3) ensure that operations are applied only to objects of the appropriate types. However, as we pointed out above, block structuring and static scope rules are insufficient to detect references to uninitialized variables or to nonexistent heap storage.

Dijkstra, in his essay "On the Scope of Variables" [5], attempted to deal with uninitialized data, as well as to control the indiscriminate importation of variables from outer to inner scopes. Dijkstra's concept of a region containing an "obligation to initialize" was inspirational to the idea of typestate. Dijkstra did not treat inverse problem of finalization, nor did his mechanism generalize to other operation sequences or other flow control constructs.

Other research has involved the augmentation of type rules with finite-state sequencing constraints. For example, path expressions [3] and access right expressions [4], [10] are concerned with specifying scheduling constraints for operations on *shared data objects*. Neither of these approaches is intended to be enforced by a compiler, although either can be supported by a combination of compile-time and runtime mechanisms.

The typestate-tracking algorithms presented here are related formally to information propagation algorithms [6]. The objective of these algorithms is to label a program graph with "facts" subject to constraints. The constraints determine the "maximal" fact which can be placed on a successor node given the fact on the current node. Graham and Wegman's analysis of the complexity of graph labeling algorithms can be applied to typestate, by replacing set inclusion with the relation $\leq$ and intersection with greatest lower bound.

The use of lattice-structured sets of facts appears also in a number of algorithms for constant propagation such as [11] and [18].

B. Conclusions

We have shown that the concepts of typestate and typestate checking provide a significant enhancement to the compile-time processing capabilities of compilers for strongly typed programming languages. In particular, we have shown that compile-time typestate checking supports

- compiler-guaranteed security by the avoidance of *all* nonsensical execution sequences;
- compiler-guaranteed safe manipulation and finalization of dynamic objects;
- enhanced execution efficiency: typestate-correct programs can be considered "authorized," and be allowed to run "fast paths" since they are known to be secure; and
- modular testing and verification.

The example of NIL shows that typestate checking can be embedded in a "realistic" powerful programming language supporting a highly dynamic computation model as well as concurrency.

Typestate checking can be viewed as an automatable subset of program verification. While mechanical verification that a program meets all its specifications is not currently feasible, incorporating typestate checking into a compiler at least allows verified programs not to be corrupted by unverified ones. Although typestate checking is limited to finite-state properties, we believe that this small subset of verification nevertheless handles a most difficult and bothersome class of errors, and is therefore a valuable addition to software reliability.

Acknowledgment

We wish to thank A. Demers for valuable criticisms of this paper, and acknowledge the contributions of R. Bernstein, W. Burger, M. Conner, N. Halim, J. McInerny, D. Milch, F. Parr, and J. Pershing who participated in the NIL project during various stages.

References

[1] *Reference Manual for the Ada Programming Language*, ACM Ada-Tec, Draft Proposed ANSI Standard Document, July 1982.
[2] D. M. Berry, R. A. Kemmerer, A. von Staa, and S. Yemini, "Toward modular verifiable exception handling," *Comput. Lang.*, vol. 5, pp. 77-110, 1980.
[3] R. H. Campbell and A. N. Habermann, *The Specification of Process Synchronization by Path Expressions (Lecture Notes in Computer Science)*, vol. 16. New York: Springer-Verlag, 1974.
[4] M. H. Conner, "Process synchronization by behavior controllers," Ph.D. dissertation, Univ. Texas at Austin, Dec. 1979.
[5] E. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976.
[6] S. L. Graham and M. Wegman, "A fast and usually linear algorithm for global flow analysis," *J. ACM*, vol. 23, no. 1, pp. 172-202, Jan. 1976.
[7] B. Hailpern and S. Owicki, "Modular verification of concurrent programs," *IBM Res. Rep.* RC 9130, 1981.
[8] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, Oct. 1969.
[9] —, "The Emperor's old clothes," reprinted in *Commun. ACM*, vol. 24, pp. 75-83, Feb. 1981.
[10] R. Kieburtz and A. Silberschatz, "Access-right expressions," Univ. Texas, Tech. Rep., 1979.
[11] G. A. Kildall, "A unified approach to global program optimization," in *Proc. 1st ACM Symp. Principles of Programming Languages*, pp. 194-206.
[12] R. L. Schwartz and P. M. Melliar-Smith, "The finalization operation for abstract data types," in *Int. Conf. Software Eng.*, 1981.
[13] M. Sherman, A. Hisgen, and J. Rosenberg, "A methodology for programming abstract data types in Ada," in *Proc. AdaTEC Conf. ADA*, Oct. 1982.
[14] R. E. Strom, "Mechanisms for compile-time enforcement of security," in *Proc. 10th ACM Symp. Principles of Programming Languages*, Austin, TX, Jan. 1983.
[15] R. Strom and N. Halim, "A new programming methodology for long-lived software systems," *IBM J. Res. Develop.*, Jan. 1984.
[16] R. Strom and S. Yemini, "The NIL distributed systems programming language: A status report," in *Proc. NSF/SRC Seminar Semantics of*

Concurrency (Springer-Verlag Lecture Notes in Computer Science, Vol. 197), Pittsburgh, PA, July 1984.
[17] R. Strom, S. Yemini, and P. Wegner, "Viewing Ada from a process model perspective," *Proc. Int. Ada Conf.*, Paris, 1985.
[18] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *Proc. 12th ACM Symp. Principles of Programming Languages*, Jan. 1985.

Robert E. Strom, photograph and biography not available at the time of publication.

Shaula Yemini, photograph and biography not available at the time of publication.