DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY · STANFORD, CA 94305

SEL-78-024

THE FORMAL DEFINITION OF A REAL-TIME LANGUAGE

John L. Hennessy and Richard B. Kieburtz

Technical Report No. 155

July 1978

This work was partially supported by
National Science Foundation Grant No. 5042203,
and Joint Services Electronics Program
Contract No. N0014-75-C-0601

SEL-78-024

THE FORMAL DEFINITION OF A REAL-TIME LANGUAGE

John L. Hennessy and Richard B. Kieburtz[†]

Technical Report No. 155

July 1978

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California

[†]Department of Computer Science
SUNY at Stony Brook
Stony Brook, New York

This work was partially supported by National Science Foundation Grant #J042203, and Joint Services Electronics Program Contract #N0014-75-C-0601.

THE FORMAL DEFINITION OF A REAL-TIME LANGUAGE

John L. Hennessy
Stanford University

Richard B. Kieburtz
SUNY at Stony Brook

Technical Report No. 155

July 1978

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California

ABSTRACT

This paper presents the formal definition of TOMAL (Task-Oriented Microprocessor Applications Language), a programming language intended for real-time systems running on small processors. The formal definition addresses all aspects of the language. Because some modes of semantic definition seem particularly well-suited to certain aspects of a language, and not as suitable for others, the formal definition employs several, complementary modes of definition.

The primary definition is axiomatic in the notation of Hoare; it is employed to define most of the transformations of data and control states affected by statements of the language. Simple, denotational (but not lattice-theoretic) semantics complement the axiomatic semantics to define type-related features, such as the binding of names to types, data type coercions, and the evaluation of expressions. Together, the axiomatic and denotational semantics define all the features of the sequential language.

An operational definition, not included in this paper, is used to define real-time execution, and to extend the axiomatic definition to account for all aspects of concurrent execution. Semantic constraints, sufficient to guarantee conformity of a program with the axiomatic definition, can be checked by analysis of a TOMAL program at compilation.

Index Terms: formal definition, programming language semantics, axiomatic definition, denotational semantics, concurrent programming

1. Introduction

TOMAL is a real-time programming language designed for small processors operating in stand-alone configurations without the benefit of a standard operating system [Hennessy 75, 77]. In this section we will briefly and informally define the various elements of the language.

It is a language in which to compose programs to meet real-time response constraints imposed by an external environment. A TOMAL program is built in modules, with each module constructed as a set of procedures and concurrently executable components called tasks. The body of a procedure or a task is formed by the sequential composition of statements,

TOMAL is a strongly typed language, whose type structure is similar to, but somewhat less rich than,that of Pascal. There are three standard scalar types, boolean, integer, and char; and a real arithmetic types: Set and array types are defined, and a one-dimensional array of characters is given special treatment as a predeclared type string, with it own operators There are no file or record types, and no pointer types. The extent of any TOMAL type can be determined from its declaration.

The control structures of the language consist of standard constructs, such as: if..then..else, while, case, a compound statement, an integer for statement with directional and step clauses, and a procedure return statement. A repeat statement creates an iteration with no specified termination condition. The break statement, appearing in several programming languages as exit [Wulf 71], is used to exit from any statement block. The statement break L exists from the block labelled by L, which may be nested arbitrarily deeply. The for all statement iterates the execution of a statement block, while quantifying an iteration control variable over a finite set [Hoare 72a].

Three types of procedures are provided by TOMAL: proper procedures, function procedures, and assignable procedures. In order to maintain a static environment, procedures cannot be recursive. Procedure parameters are always passed by value except for strings and arrays which are passed by reference for efficiency. Since aliasing of variables is prohibited, parameters passed by reference have the same effect as if passed by value-result.

A function procedure returns a value of a scalar or arithmetic type and is not allowed to modify global variables or parameters of array or string types. Thus a function call can be embedded in an expression without producing any side effect. A proper procedure has no return values, may produce side effects, and is invoked by a call statement.

The assignable procedure has been introduced in conjunction with the operation of simultaneous, multiple-value assignment in order to reduce the need for side effects or var parameters of procedures. It yields a list of one or more return values having simple (i.e., not array or string) types. An assignable procedure is invoked by an occurrence of its name and a list of actual parameters, just as is a function. However, a call to an assignable procedure can only appear on the right hand side of an assignment statement.

Multiple-value assignment binds a list of values from the right hand side to the list of variables on the left hand side. The assignment is simultaneous and correspondence is by order of occurrence. If two variables on the left hand side are the same (i.e., have the same L-value), but the corresponding right-side values differ, the assignment is undefined. When the right hand side is an assignable procedure, the value list is that resulting from the procedure invocation.

The use of value parameters and simultaneous multiple assignment to replace the customary practice (with programs written in Pascal, PL/1, etc.) of using var parameters to secure value-result updating of variable parameters prevents the aliasing of scalar variables within the body of a procedure. The exception to this rule that was made for array and string variables requires compile-time checking to detect and warn of possiblitities for aliasing. However, since the language was designed in an attempt to provide a tool for real-time programming and replace the use of assembler language as a programming medium, some concessions to efficiency must be made.

The concurrent features of TOMAL are embedded in its multi-tasking capabilities. Every TOMAL program consists of globally declared semaphores, a number of modules, and an initial activity request. Each module contains declarations of variables and procedures local to that module, and a set of tasks. Each task consists of locally declared variables and procedures, and a statement block. The names of all tasks and of explicitly exported procedures are considered global to all modules.

A task is the basic unit of program activity; it may be currently active, suspended, requested for activation, or dormant. Tasks are requested for activation by means of the request statement; they become active when they are scheduled. When the task completes execution of its statement block it terminates and becomes dormant. Initial value parameters can be passed to a task in a request statement. At most one activation of a task and one request for a task can be simultaneously outstanding. Multiple requests have no effect on the task state, but merely update the parameters.

Synchronization mechanisms are provided so that access to shared resources may be regulated. Binary-valued semaphores are used within a synchronization statement of the form with $S_1, \ldots, S_n$ do A, where A is a single or compound statement. The effect of executing the first part of the statement is to suspend execution until all of the semaphores $S_1, \ldots, S_n$ are free, and then to lock the n semaphores and continue execution. This construct is the equivalent of the P-multiple operation on binary semaphores [Vantilborgh 72]; either all are successfully locked, or none are and the task attempting the lock becomes suspended at that point.

The semaphores in TOMAL are binary-valued, and are used to control access to shared data and procedures, and also to allow restriction of the otherwise impl icitlyconcurrent execution of a group of tasks. When a semaphore protects a group of tasks, each request for a task in the group is required to be preceded by a semaphore lock (P-operation). A task-protecting semaphore is unlocked by implicit action whenever one of the protected tasks terminates its execution. Thus the members of a protected group of tasks are guaranteed to execute mutually exclusively in time.

Semaphores may also be used to create critical sections, thus regulating access to other shared resources. If a with statement contains one or more semaphores not associated with tasks, then the compound statement headed by the with statement becomes a critical section for those semaphores. The semaphores protecting a critical section are implicitly freed upon termination of the critical section. Critical sections protected by a common semaphore execute in a mutually exclusive manner.

The features of TOMAL that are directed toward real-time applications are the ability to declare fixed priorities for task scheduling, the ability to specify minimum response times for the delivery of service requested by external processes, and the ability to declare external device characteristics, allowing the compiler to generate I/O routines. The specification of task priorities imposes constraints on the possible sequences of task activation that may be scheduled. The I/O and response time specifications introduce notions of time dependence, requiring the definition of a metric for time in an implementation. These are powerful, integral features of the language and deserve careful definition.

2. An Approach to the Formal Definition of TOMAL

The primary reason for giving a formal definition of a programming language is to supply concise and unambiguousmeaning, independent of an actual or proposed implementation. Historically, most programming languages have been loosely or inadequately defined, with the result that early implementations have often served as the language definition, or that the language has existed with a number of different interpretations. Other important reasons have been cited in [Hoare 73]; among these are: to give to the programmer a clear, unambiguous meaning for each language construct, and to provide a logical basis for verification of programs written in the language.

Two factors influence the form of the definition: the desire to support program verification, and the requirement that the entire language must be defined. In order to meet these requirements, we have employed multiple modes of semantic specification. This method of supplying complementary semantics was suggested and used by Hoare and Lauer [Hoare 74] and later by [Donahue 75] to define a subset of Pascal.

Axiomatic semantics [Hoare 69] are used for the primary definition of program statements. This mode of detinition offers several major advantages, including: conciseness, comprehensibility, and applicability in program verification. There are three major deficiencies of the axiomatic method for defining TOMAL. First, it is unable to easily express the semantics of expression evaluation, especially the type dependency and type correctness of expressions. Secondly, axiomatic semantics are extremely awkward to use in defining the bindings of names to types within a scope. This is because it is based on an underlying, uninterpreted functional calculus in which no distinction is made between a name and its value.

A recent paper presented an axiomatization of declarations, scope concepts, and the relationship of exit or escape statements [Fokkinga 77]. The approach utilized was the introduction of an environment component which is carried along within the proof. Although our approach treats declarations and scope rules with a different semantics, it allows the use of axiomatic semantics without the need to consider environment, nor does it introduce a new name producer into the axiomatic definition.

Lastly, the real-time features of priority and time-dependency introduce complexities for which the axiomatic method is not well suited. These complexities fall into two categories. The real-time aspects of the language allow specification of response time criteria and scheduling priorities. Because these two features determine the order of execution by a metric not expressible in the axiomatic definition (i.e., time), properties of statement scheduling are not axiomatizable.

Therefore, the semantics of TOMAL which are not specifically dependent on expression evaluation, scope and name-type binding, or scheduling are defined by the axiomatic definition; the other features are defined by complementary schemes of semantics.

In order to alleviate the shortcomings of the axiomatic method with respect to sequential language features, we introduce two forms of simple, denotational semantics. The two aspects of expression evaluation, namely data type coercions and operator evaluation, are defined by a set of simple

functions. These functions express the semantics in terms of well-understood operations over standard domains. The scope and name-type binding rules are defined by another set of functions and rules for their composition. These express the semantics of name-type b nding in a simple lambda-calculus. Together, the functional and axiomat c semantics define all the scheduling-
4 independent features of TOMAL.

The scheduling-independent semantics of TOMAL are self-consistent, but manifestly incomplete because they describe the transformations of data induced by all conceivable execution sequences of a program, including many that cannot occur. In order to account for the constraints imposed by priority and response-time scheduling, we have chosen to employ an operational mode of semantics. This choice is dictated by the natural definition of task scheduling (which is itself operational) and the ease with which a time metric can be introduced. Within the operational definition it has been possible to introduce the notion of execution time for a statement, as well as the concept of scheduling by a time-dependent priority scheduler. Thus the operational definition utilized defines a number of language features which have previously been left informal. The operational definition utilizes VDL (Vienna Definition Language) [Lucas 71]. As is the nature of an operational semantics, this definition has the form of an abstract implementation of the language. However, the definition is intended to constrain the implementer as little as possible and yet unambiguously define the language. The VDL model, its necessity, and its relationship to the axiomatic definition (i.e., consistency) are not presented in this paper but appear in [Hennessy 77].

The axiomatic definition relies on certain assumptions concerning sequentiality of access to shared objects. These assumptions may sometimes be violated during the execution of unstructured, concurrent programs. Syntactic restrictions sufficient to ensure the validity of these assumptions could have been imposed by the language design, but the designers chose not to do so. Such syntatic restrictions must also necessarily prohibit many programs that would satisfy the required access conditions during their actual execution. Therefore, in order to use the axiomatic definition, we give a set of computation-dependent constraints to which concurrent computations must adhere. If the constraints are not adhered to in a particular program, its semantics are defined by the VDL definitions but not necessarily by the axiomatic definition. A set of compile-time testable conditions, sufficient to ensure that the constraints hold, is checked by the program analysis module of the TOMAL language processor. These conditions are not unreasonable, but ensure that certain constructs are used as they are intended in an environment where concurrency is supported. The constraints are discussed in detail and the consistency of the operational and axiomatic semantics, under the constraints, has been proven in [Hennessy 77].

3. The Axiomatic Definition of Statement Constraints

In this section we give an axiomatic definition of a standard interpretation of the TOMAL language, guaranteed to apply when certain constraints on concurrent execution are obeyed [Hennessy 77]. The axiomatic definition is presented in three parts:

1) the definition of sequential statements:
2) synchronization operators and constructs that describe concurrently executable statements;
3) the data types.

The form of a verification formula was developed by Kieburtz and Cherniavsky [Kieburtz 76], and is an extension of [Nassi 74]. A verification formula describing the effect of executing a statement S is written;

$$P \{S\} <Q,Q'>$$

where P,Q,Q' are assertions. The interpretation given to the formula is:

> if the precondition P holds prior to the execution of S, then one of the two postconditions, either Q or Q', must hold following the execution of S. If case S terminates with normal, sequential flow of controls then Q is the postcondition. However, if S terminates by executing a nonsequential control operator, such as <u>break</u> or <u>return</u>, then Q' is the postcondition accompanying the control transfer.

Although the double consequent form of the axiom adds some additional complexity, it enables us to accurately define a number of language aspects, such as the <u>return</u> statement, the <u>break</u> statement and the <u>case</u> statement, in a totally formal approach. Thus although the language includes rich control structures, they can be neatly defined. In order to eliminate some complexity, we have omitted the second consequent, whenever it is obviously false, such as in an assignment statement, or where the second consequent in the hypothesis is identical to the second consequent in the conclusion.

Rules of inference have the form (due to Hoare [Hoare 69]):

$$\frac{V}{W}$$

where V is a hypothesis consisting of one or more verification formulae or assertions, and W is the conclusion consisting of either a verification formula, or a theorem expressed in the verification logic. The meaning of an inference rule is that whenever the hypothesis can be proved, the conclusion is said to be proved by inference. The axioms for sequential constructs and data type (but not those for operators or coercions) are based on those given by Hoare and Wirth for the language Pascal [Hoare 73]. The axioms for repeat, break, compound statement, and case are based on [Kieburtz 76).

The following abbreviations are utilized:

1) A is a statement;

2) A* is a sequence of zero or more statements;

3) D stands for $d_1,\ldots.,d_n$;

4) $D = f(C)$ stands for $d_1 = f_1(C_1),\ldots.,d_n = f_n(C_n)$;

5) If w is a variable or constant, then T_w is the type of w.

Sequential Statements

1) Empty statement

The empty statement has only a sequential termination condition, which reflects the fact that the statement alters no variables.

P {;} P

2)- Break statement, break L;

The break statement has no sequential termination condition. The nonsequential postcondition records the target of the break and reflects the fact that no program variables are altered. The label variable, ℓ , is a distinguished variable of the verification logic, used to record the target of a nonsequential execution. Thus in the statement break L, ℓ will get the value L, the target of the nonsequential execution.

P (break L;} <false, $P \wedge \ell = L$>

3) Compound statement, begin A* end L;

The rule of inference for the compound statement accounts for three distinct ways in which control can pass from the statement list A* during its execution.

a) if A* terminates sequentially, then so does the compound statement

b) if A* terminates nonsequentially, with a break target different from L, then the compound statement also terminates nonsequentially.

c) if A* terminates nonsequentially, and its break target is L, then the compound statement containing A* and labelled by L terminates sequentially.

$$\frac{P\ \{A^*\}\ \langle Q,\ R\rangle}{P\ \{\underline{\text{begin}}\ A^*\ \underline{\text{end}}\ L;\}\ \langle Q \vee R^{\ell}_{L},\ R \wedge \ell \neq L\rangle}$$

4) Repeat statement, repeat A

The inference rule for repeat indicates that termination can only occur by a nonsequential flow of control from the statement list, A. The rule also states that an assertion P, which is invariant for the statement block is unaffected by the repeat control structure. If the repeat structure is combined with the break statement to create either of the familiar control structures while or repeat-until, the invariant can be used to formulate the usual axioms for those structures.

$$\frac{P\ \{A\}\ \langle P,\ Q\rangle}{P\ \{\underline{\text{repeat}}\ A\}\ \langle \text{false},\ Q\rangle}$$

5) While statement, while B do A

The while rule embodies the usual rule for while statements.

$$\frac{P \wedge B\ \{A\}\ P}{P\ \{\underline{\text{while}}\ B\ \underline{\text{do}}\ A\}\ P \wedge \neg B}$$

6) If statements

The inference rules for the if statements embody the usual rules, adding only the possibility of nonsequential termination.

a) if B then Al

$$\frac{P \wedge B\ \{Al\}\ Q}{P\ \{\underline{\text{if}}\ B\ \underline{\text{then}}\ Al\}\ (P \wedge \neg B) \vee Q}$$

b) if B then A1 else A2

$$\frac{P \land B \{A1\}\ Q,\quad P \land B \{A2\}\ Q}{P\ \{\underline{\text{if}}\ B\ \underline{\text{then}}\ A1\ \underline{\text{else}}\ A2\}\ Q}$$

7) Case statement, case x of $[k_i : A_i]^*$ end;

The case statement is similar to that of Pascal except that subranges may also be used to form a finite list of constants for each label. The case statement has two inference rules. The first rule describes the effect of executing the empty case statement. The second rule is a recursive definition of the semantics of a list of case instances. The notation $[k_i : A_i]^*$ is used to indicate zero or more occurrences of a <label: statement> pair. The index i is a metasymbol used to distinguish between case instances. The case instance labels, k_i, are defined as subsets; for this reason a membership test determines if the case selector is associated with a particular instance.

a) case x of end;

$$P\ \{\underline{\text{case}}\ x\ \text{of}\ \text{end}\}\ \langle P,\ \text{false}\rangle$$

b) case x of $[k_i : A_i]^*\ k_n : A_n$ end L;

$$\frac{y \varepsilon \{k_n\} \land P_y^x \{A_n\} \langle Q_n, Q_n'\rangle,\quad P\{\underline{\text{case}}\ x\ \underline{\text{of}}\ [k_i : A_i]^*\ \underline{\text{end}}\ L\}\ \langle R, R'\rangle}{P\{\underline{\text{case}}\ x\ \underline{\text{of}}\ [k_i : A_i]^*\ K_n : A_n\ \underline{\text{end}}\ L\}\langle R \lor Q_n \lor Q_{nL}'^{\ell},\ R' \lor (Q_n' \land \ell \neq L)\rangle}$$

8) for statements

a) for all statement, for all e in Y do A

The rule for the for all indicates that the statement list is executed while a quantified variable ranges over the members of a designated set, in order, and that the set is evaluated once. This rule differs slightly from the rule for Pascal; the same approach is used in the integer for statements.

Let T_y be the smallest type that includes all elements of the set Y; denote T_y by a subrange a..b. Then $T_e \subseteq T_y$ must hold.

Define $\text{pred}_y(e)$ = if $\text{pred}(e) \varepsilon Y$ then $\text{pred}(e)$ else if $\text{pred}(e) \varepsilon a..b$ then $\text{pred}_y(\text{pred}(e))$ else undefined

$$\frac{(e \varepsilon Y) \wedge P[\{a..\text{pred}_y(e)\} \cap Y]\ \{A\}\ P[\{a..e\} \cap Y]}{P[\phi]\ \{\textbf{for all } e \text{ in } Y \textbf{ do } A\}\ P[Y]}$$

The rule says the statement increases for the set of values for which P holds on each iteration. Then the for all ensures P will hold on all elements of Y, upon completion.

b) integer for..to, for x := m to n step p do A

This statement and its rule are similar to the for statement of Pascal. The inference rule for the for..to statement illustrates that: evaluation of the control expressions occurs once, the control identifier takes on the initial value and is incremented by the step value each time, the step value must be positive, and the control identifier can not be updated in the statement block.

Let $Y = \{i \mid i = m + kp\} \cap \{m..n\}$; Y is the set of values the control identifier will be assigned:

$$\frac{(x \varepsilon Y) \wedge P[\{m..x - p\} \cap Y]\ \{A\}\ P[\{m..x\} \cap Y]}{P[\phi] \wedge (p > 0)\ \{\textbf{for } x := m \text{ to } n \textbf{ step } p\ A\}\ P[Y]}$$

c) integer for..downto, for x := n downto m step p do A

The downto for statement reflects the same properties as the to limit form. The only difference is the direction of the step, relative to the natural order defined on the domain of values.

Let $Y = \{i \mid i = n\ k*p\} \cap \{m..n\}$.

$$\frac{(x \varepsilon Y) \wedge P[\{x..p + n\} \cap Y]\ \{A\}\ P[\{x..n\} \cap Y]}{P[\phi] \wedge (p<0)\{ \text{ for } x := n \text{ downto } m \text{ step } p \text{ do } A\}\ P[Y]}$$

9) procedure or function body

This new rule for the statement body of a procedure or function has the effect of binding the break target for the return statement. This results in a neat axiomatization for the nonsequential control structure return.

Let B be a procedure or function of the form procedure B....; S_B end B, or function B....; SB end B.

$$\frac{P\ \{S_B\}\ \langle Q,\ R\rangle}{P\ \{\text{procedure B...};\ S_B\ \underline{\text{end}}\ \text{B};\}\ \langle Q \vee R^{\ell}_{\text{endproc}},\ \text{false}\rangle}$$

Likewise if B is a function.

10) return statements

The axiom for the return statement demonstrates that the statement has two effects: to store the return expression values, and to cause a break to the end of the procedure. Together with the procedure body rule, these two new rules parallel the rules for breakand the compound statement.

Let B be a procedure, with n return parameters, i.e.,

n = 0 if B is a proper procedure,

n = 1 if B is a function procedure,

n = number of return parameters if B is an assignable procedure.

Let $z_1,\dots,z_n$ be the implicit return variables for B.

$$P^{z_1 \dots z_n}_{e_1 \dots e_n}\ \{\underline{\text{return}}\ (e_1,\dots,e_n);\}\langle\text{false}\ ,\ P \wedge \ell = \text{endproc}\rangle$$

17) procedure declaration and invocation

There are two rules of inference associated with procedures. The first rule is associated with the declaration of a proper procedure and is called the rule of declaration. This rule defines the effect the procedure has on global variables and array or string parameters, by means of a pair of functions. The function f gives the effect on array and string parameters, while h gives the effect on global variables. The conclusion of the rule of declaration is a theorem about the functions f and h. This theorem may then be instantiated with actual parameters. It need only be established once for each procedure. The second rule, called the rule of invocation, displays the semantics for the <u>call</u> statement. Since variable aliasing is prohibited this rule states that the <u>call</u> statement is effectively an assignment to the array and string parameters, and the global variables. The values which these variables receive are given by the functions f and h defined in the rule of declaration with arguments instantiated by the actual parameters given in the <u>call</u>. Since the rule of invocation involves assignment of initial value parameters, coercion may be necessary. C_a is a function used for coercion in assignments; C_a is defined in detail in Section 4.

The rules for procedure declaration and procedure call are based on those for Pascal [Hoare 73], but differ in three ways. First, the parameter passing mechanisms are constrained by the language definition. Secondly, the absence of variable aliasing is ensured by the restrictions given below. Lastly, TOMAL procedures are not recursive.

Variable aliasing can be prohibited by the following three restrictions. Let P be a procedure containing references to global variables G_1 then $G_1 \cap T = \phi$ set of actual array and string variables passed in calls to P.

1) $T \cap G = \phi$

2) If t_1 and t_2 are actual array and string parameters passed in a single call, then t_1 and t_2 are distinct

3) If P calls Pl and P_1 updates global variables G

Let A be a proper procedure of the form:

<u>procedure</u> A $(w_1, \ldots, w_n, r_1, \ldots, r_j)$; S_A <u>end</u> A

Let A reference global variables $g_1,\dots,g_n = G$;
A has array and string parameters $r_1,r_2,\dots r_j = R$, and other (value) parameters $w_1,w_2,\dots,w_n = W$;

a) rule of declaration

$$\frac{P\ \{S_A\}\ \langle Q\ ,\ \text{false}\rangle}{P \supset Q^{R\quad G}_{f(W,R)h(W,P)}}$$

W does not occur free in Q.
for all values of W,R,G

b) rule of invocation

Consider the <u>call</u> statement for A:

<u>call</u> A(B,C);

where $B = b_1,\dots,b_n$ is a sequence of actual values corresponding to the W and $C = c_1,\dots,c_j$ in a sequence of actual variables corresponding to the R. The types of C must match the types of R exactly.

Let $D = C_a((B,T_B), T_W)$.

$$R^{C\qquad G}_{f(D,C)\ h(D,C)}\ \{\underline{\text{call}}\ A(B,C);\}\ R$$

(N.B. This requires restrictions which prevent aliasing)

12) assignment statements

There are two important cases to consider. The axiom for multiple assignment of a list of scalar or arithmetic expressions to a list of variables is a generalization of the familiar single-assignment axiom. A second case of assignment governs the invocation of an assignable procedure; here the possible modification of global variables or var parameters of array and string types must be accounted for. Special cases of the substitution rule govern an assignment that performs partial updating of an array or string variable, or requires coercion.

a) assignment of expressions to variables

$$P^{x1...xn}_{e1...en} \{x1,...,xn := e1,...,en\} P$$

The substitution $P^{x1...xn}_{e1...en}$ is not a composition of single-expression substitutions, but a simultaneous replacement of the variable names xl . ..xn by the corresponding expressions el...en. This reduces to the familiar rule for assignment to a single scalar variable when the length of the substitution list is one. The result of substituting two or more distinct values for a common variable is undefined.

b) assignment of the result of invoking an assignable procedure

$x_1,...x_n$:= A(B,C);

The rule of declaration specifies the effect of an assignable procedure in terms of three functions: the global variable function, g; the array and string parameter function, f; and a function denoted by the name of the procedure, which relates the values in the return list to the input parameters. The conclusion of the rule of declaration is a theorem defining properties of the three functions; such a theorem is proven only once for each procedure. The rule of invocation states that two substitutions of values for variables are composed. First, new values are substituted for the array and string parameters and for global variables updated by the procedure invocation; next, new values are substituted for the scalar variables that appear explicitly on the left side of the assignment operator. This new rule differs from previous rules for procedures in that it makes provision for returning any number of values. Consider an assignable procedure A, declared as:

procedure A $(w_1,\ldots,W_m,r_1,\ldots,r_1)$ <u>returns</u> $(t_1,\ldots,t_n)$; S_A; <u>end</u>

A has array and string parameters $r_1,\ldots,r_j$ = R; and A has other formal parameters $w_1,\ldots,w_m$ = W.

Let A reference global variables $g_1,\ldots,g_k$ = G, and SA have implicit return variables $z_1,\ldots,z_n$ = Z.

a) rule of declaration

$$\frac{P\ \{S_A\}\ Q}{P \supset Q^{R}_{f(W,R)}{}^{G}_{g(W,R)}{}^{Z}_{A(W,R)}} \quad \left\{\begin{array}{l}\text{where no variable of Z occurs free in P,}\\ \text{and no variable of W occurs free in Q.}\end{array}\right.$$

for all W,R,G,

b) rule of invocation

Consider an invocation of the form A(B,C).

B corresponds to the W, and C to the R. The types of C and R must match exactly.

Let D = $C_a((B,T_B),T_W)$

$$[[R^X_Y]^{R}_{f(D,C)}{}^{G}_{g(D,C)}]^{Y}_{A(D,C)}\ \{x_1,\ldots,x_n := A(B,C);\}\ R$$

Where Y denotes a list of dummy variable names that do not occur in R, B or C; and C A G=ϕ.

The assignment rules given in a) and b) utilize normal substitution and do not account for either subscripted references on the left hand side of an assignment or for possible coercions. These two possibilities are accounted for by an extension to the rules for substitution. These rules, given in Appendix 1, specify the necessary coercions and the effect of assignment to subscripted variables.

Rules Governing Concurrent Execution

A restricted form of a binary semaphore is used to synchronize concurrent activities. Semaphores are specified by declarations of the form

semaphore S protects $(v_1,\ldots,v_n)$;

S1 protects tasks $(a_1,\ldots,a_m)$

In the first case S is a semaphore which protects the variables (or procedures) $v_1,\ldots v_n$. The axiomatic definition imposes certain restrictions on access to protected variables and procedures. In the second form of declarat on S1 protects the tasks $a_1,\ldots,a_m$; protection of tasks as resources differs from protection of shared variables and procedures.

Semaphores protecting variables and procedures are used in a critical section, of the form:

```
with S1,...,Sn do
  A
```

This statement is called a critical section for $S_i,\ldots,S_{j_i}$, where $S_i,\ldots,S_j$ are semaphores protecting variables or procedures. (Note that if all of $S_1,\ldots,S_n$ protect tasks this is not a critical section.) The language requires that all updates to variables (or calls to procedures) protected by S must occur within the statement body of a critical section for S. (The formal definition of update appears in Appendix 2.)

The critical section structure (i.e., where $S_1,\ldots,S_n$ protect variables or procedures) is easily understood by the following semaphore implementation

```
P(S1,...,Sn);
    A
V(S1,...,Sn);
```

(remembering that semaphores are binary-valued).

Tasks differ from sequential resources; a task, once invoked, may not be reinvoked until the execution of its first invocation has been completed. A request for a task, on the other hand, may be performed (by another task) at any time. If requests for a task t are issued from more than one point in a program, one can ensure that a request is never overwritten by embedding each request for t in a with statement that locks a semaphore protecting it. Recall that a semaphore protecting a task is unlocked at the termination of that task's execution, rather than at the end of a with statement; thus when execution is suspended at a task-protecting semaphore, it awaits completion of the protected task.

In general a with statement may mix semaphores of both types. The resulting structure is a combination, where all critical section semaphores are freed at the end of the compound statement. Task protecting semaphores are freed at task terminations.

The rules enforced by the language syntax are not sufficiently strong to ensure that the proof rules are applicable. Instead, a set of dynamic constraints must be satisfied. These constraints, as well as a set of static, syntatic conditions sufficient to ensure them, are discussed in section 6.

In the Floyd-Hoare logic, the fundamental rule relating the effect of a statement to its environment is the rule of sequential composition. When the execution of a statement is not controlled by simple sequencing, but involves repetition or nondeterministic scheduling, the inference rules invoke the notion of an invariant assertion. An invariant assertion describes the program states in which control passes to or from a segment in all execution sequences.

If I is an assertion, I is said to be invariant for A if

$I\{A\}I$ is provable.

Let I_S be an assertion associated with semaphore S, and containing only those variables protected by S. I_S is an invariant for S if, hypothesizing that I_S

is true each time S is locked, it is provable that I_S is true each time S is unlocked.

A variable v is said to be <u>safe</u> at a statement A, if any of the following hold.

1. v is local to task or procedure M, and A belongs to M.
2. v is updated only in M, and A belongs to M.
3. v is protected by semaphore S, and A belongs to the critical section or task protected by S.

The rules for concurrency are largely based on [Hoare 71] and the extension by [Owicki 75]. Our rules extend the previous results by using the concept of task as a resource, and defining the meaning of variable initialization. It is not the aim of this definition to be complete for scheduling aspects. The effect of scheduling by priority (and response time) is defined by a nonaxiomatic definition appearing elsewhere [Hennessy 77).

1) Tasks, requests and task invariants.

a) Request statement, <u>request</u> t $(e_1,\ldots,e_n)$

Pr(t) is an assertion over the parameters of t, called the domain assertion,associated with t. Pr(t) must be proven as a precondition of each request of task t. If P is an assertion over variables safe at the request statement, then

$$Pr(t)^{a_1,\ldots,a_n}_{e_1,\ldots,e_n} \wedge P\{\underline{\text{request}}\ t(e_1,\ldots,e_n)\}\ P$$

where $a_1,\ldots,a_n$ are the formal parameters of the task.

The axiom of request indicates that the request statement assigns a sequence of values given as expressions to the formal parameters in the task. Any variables which are not <u>safe</u> at the request may be accessed by the newly requested task, thus destroying their values;

hence only variables which are safe remain invariant. Also the axiom ensures that Pr(t) will be true prior to the execution of a request for t; this requirement is used to strengthen the invariant.

b) Task Invariant

$$\forall t \epsilon T_S (pr(t) \wedge Inv_S \{A_t\} Inv_S)$$

where A_t is the statement body of task t, and T_S is the set of tasks protected by S.

The invariant must also satisfy an initialization constraint (given in a following section).

c) Task Initiation

The following axiom describes what assertion is known to be true when a task body begins execution. The assertion includes the condition established by all request statements for the task. If the task is protected by a semaphore, the condition that the invariant for that semaphore is true also becomes a part of the assertion.

i) true {T: task (a, ,...,a_n) } Pr(T)

ii) true {T: task (a,...,a_n) protected by S} Pr(T) $\wedge$ Ins_S

2) Initial Condition of the Invariant

In this section, verification formulae, which ensure that an invariant is initially true, are specified. These formulae are dependent on the initialization of global variables, since this specifies the initial system state for all the shared resources.

The declarations of global variables may also initialize values, and are treated as statements, according to the following cases:

1) P {var x : T} declaration without initialization

2) P {var x : T initial (c)} P $\wedge$ x=c simple variable initialization

3) P {<u>var</u> x : <u>array</u> (a..b) of T <u>initial</u> $(C_a,\ldots,C_b)$}

$P \wedge x(a) = C_a \wedge \ldots \wedge x(b) = C_b$ array initialization

where the initial values are required to be type-coercible to the declared types. Then if D is the sequence of all global variable declarations, and predicate P satisfies

<u>true</u> {D} P

we require that the invariant for any semaphore S must satisfy

$P \supset Inv_S$

3) Invariants for Critical Sections

In this section the requirements for the critical section invariant are given; the construction is similar to that for tasks.

Let Y = {critical sections protected by S}.

Then Inv_S must satisfy the initialization condition given above and also:

$$\forall y \varepsilon Y\ (Pcr(y) \wedge Inv_S\ \{A_y\}\ Inv_S)$$

where A_y is the body of critical section y.

Pcr(y) is called the environment assertion for the critical section y, and is over variables safe at y. It will appear in the synchronization axiom as a precondition.

4) Synchronization Axioms

Let B be the statement: <u>with</u> $S_1,\ldots,S_n$ do

A

a) Axiom for <u>with</u> clause

Let Pcr(B) be over variables safe at B; if B is a critical section then Pcr(B) must be the same assertion as was used to specify the invariant constraint.

$$Pcr(b)\ \{\underline{with}\ S_1,\ldots,S_n\ \underline{do}\}\ \bigwedge_{i:=1}^{n} Inv_{S_i} \wedge Pcr(b)$$

The axiom states that after a with statement is executed any pre-conditions about variables which were safe are retained. Other variables were subject to update during their possible suspension to await the synchronization condition. Additionally, the invariants associated with each of the semaphores are true.

Let Q be an assertion over:

{variables safe at B} U {variables protected by S1,...,Sn}

- {variables protected by semaphores in S'}.

$$\frac{\bigwedge_{i=1}^{n} Inv_{S_i} \wedge Pcr(B)\ \{A_B\}\ Q\ \langle Q,\ false\rangle}{Pcr(B)\ (\underline{with}\ S_1,\ldots,S_n\ \underline{do}\ A_B\}\langle Q,\ false\rangle}$$

The rule for the do with construct demonstrates the fact that at the end of a critical region the semaphores protecting that critical section are freed; therefore the variables protected by those semaphores are no longer safe. The postcondition only includes variables which are not protected by semaphores associated with the critical section. Note that a break statement is not allowed to exit a with statement.

4. Axioms for Data Types

If x is a constant or variable then T_x denotes the type of x.

Scalar Types

Scalar types are either predefined (in the case of integer, boolean, and char) or defined by enumeration:

$$\underline{\text{type}}\ T = (c_1,\ldots,c_p).$$

1) $c_1,\ldots,c_p$ are all the distinct members of T.

2) $(0<i<n) \supset (c_{i+1} = \text{succ}(c_i))$

3) $(0<i<n) \supset (c_i = \text{pred}(c_{i+1}))$

4) $\neg(x < x)$

5) $(x < y) \wedge (y < z) \supset (x < z)$

6) $(x \neq c_n) \supset (x < \text{succ}(x))$

7) $(x \neq c_1) \supset (x > \text{pred}(x))$

8) $(x < y) \equiv (y > x)$

9) $(x \geq y) \equiv \neg(x < y)$

10) $(x \geq y) \equiv (y \leq x)$

11) $(x \neq y) = \neg(x = y)$

Subranges can be used to define subtypes based on a scalar type. If m,n are constants of type T_0, then

$\underline{\text{type}}$ T = m..n is equivalent to the following scalar type:

$\underline{\text{type}}$ T = (m,succ(m),. . .,pred(n),n)

Predefined Scalar Types

1) integer

This type represents a subset of the integers; i stands for a member of type integer.

i) type integer = minint..maxint

ii) $(i < \text{maxint}) \supset (\text{succ}(i) = i + 1)$

iii) $(i > \text{minint}) \supset (\text{pred}(i) = i - 1)$

2) Boolean

i) type boolean $\triangleq$ (false, true)

3) char

The character type consists of a set of values, Tc, subject to the following restrictions:

i) 'A', 'B',...,'Z', '0',...,'9' are all members of Tc.

ii) 'A' < 'B' <...< 'Z' and

'0' < '1' <...< '9'

iii) $|Tc|$ = Charsetsize (a positive integer constant)

iv) minchar, maxchar ε Tc

v) $(x \varepsilon Tc) \supset (\text{minchar} \leq x \leq \text{maxchar})$

Real Arithmetic Type

The real type represents a subset, $\mathcal{R}_0$, of the real numbers with the following axioms, which specify the constraints on $\mathcal{R}_0$, and the ordering on $\mathcal{R}_0$. Let x, y, z be type real.

1) $x \varepsilon \mathcal{R}_0$

2) minreal $\varepsilon \mathcal{R}_0$ and maxreal $\varepsilon \mathcal{R}_0$

3) $\text{minreal} \leq x \leq \text{maxreal}$

4) $\neg (x < x)$

5) $(x < y) \wedge (y < z) \supset (x < z)$

6) $(x > y) \equiv (y < x)$

7) $(x \geq y) \equiv \neg (x < y)$

8) $(x \geq y) \equiv (y \leq x)$

9) $(x \neq y) \equiv \neg (x = y)$

Set types

A set type represents powersets of a scalar base type. The following axioms describe the members of a set type, and two methods for forming a set. Assume type T = set of W, where W is a scalar type. Let x,y belong to type T.

1) The subsets of W are all the distinct members of T.

2) $\{x_0, x_1, \ldots, x_m\} \equiv \{x_0\} \cup \{x_1\} \cup \ldots \cup \{x_m\}$.

3) $\{x \text{ } \underline{\text{in}} \text{ } y \mid p(x)\} \equiv \{x \mid (x \varepsilon y) \wedge p(x)\}$

where x is a bound identifier, y is a constant set expression and p is a recursive predicate.

Array types

An array is a structured homogeneous type. The axioms specify the members of an array type and the rules for indexing arrays.

An array, T, is specified by: type T = array (W) of S; where W is any scalar type, and S may be any type. T will be logically represented by a binary mapping (i.e., a set of ordered pairs) with cardinality n. Let R be the set of all values of type S and let $r \varepsilon R$; then define the following functions for any array type T:

inx_T: W x R → W and $inx_T(<y,r>) = y$.

$eval_T$: W x R → R and $eval_T(<y,r>) = r$.

1) Let t be a subset of W x S

If t′ has cardinality n and t' is a binary mapping [i.e., $\forall y,z((y,z \varepsilon t') \wedge ((y \neq z) \supset (inx_T(y) \neq inx_T(z))))]$, then t' belongs to T.

2) These are all the members of T.

3) Let t be a variable of type T and $y \varepsilon W$, and noting that inx_T is uniquely invertible; then an indexed array reference has a value defined by:

$$t(y) = eval_T(inx_T^{-1}(y)).$$

String types

The axioms define the <u>string</u> type as an <u>array:</u> of characters, and then define the special substring operator.

1) <u>type string</u>(n) = array(1..n) <u>o</u>f char.

2) For any variable, t, of type <u>string</u>(n) and $i,j \varepsilon \{1..n\}$, $t(i,j)$ denotes the substring of t, defined to be:

$$t(i,j) = \{y | (y \varepsilon t) \wedge (i \leq inx(y) \leq i+j-1)\}.$$

5. <u>Denotational Semantics for Data Type Coercions and Operator Evaluation</u>

<u>Coercions on Data Types</u>

This section describes the coercions which are permitted between the various data types of the language. First, some notation and the definition of the function used for coercion are supplied, then the various types of coercions are given. The rules for type coercions and operators are new and differ substantially from previous specifications.

0) Notation

The coercion function, C, is a domain map of the type:

C: ((value, type), type) $\rightarrow$ (value, type)

The form $C(v_1, t_1), t_2) = (v_2, t_2)$ is used to define the coercion function. The type of v_2 is assumed to be t_2. For economy of notation only the value which results from the function will be written. The meaning of $C((v_1,t_1),t_2) = v_2$ is that the value v_1 of type t_1 is coercable to type t_2, giving the value v_2.

A second coercion function C_a, used for assignment is defined as an extension of C. Only the extension not specified by C is explicitly given.

C and C_a are partial functions; when they are undefined on particular values, this means that the coercion of those values is illegal in the language.

Let T, T', T_1, T_2 be types and v, v_1, v_2 be values, then

$$T \leqslant T' \iff \forall x[x \varepsilon T \supset x \varepsilon T']$$

1) Definition of the coercion function

i) Let $v_1, v_2 \varepsilon T$; if $C((V_1,T),T')$ and $C((v_2.T),T')$ are both defined then $(v_1 < v_2) = (C((v_1,T),T') < C((v_2,T),T'))$.

This rule specifies that coercions preserve the order of values within types.

ii) If T, T' are any types, then: $T \leqslant T' \supset C((v,T),T') = v$.

This rule specifies that if v is a member of a type T and all members of type T are members of type T', then v can be coerced to type T' without a change in value. This rule is clear, since any value of v must be a member of T'.

2) Coercion between a char value and a string(1).

i) $C((v_1, \underline{char}), \underline{string}(1)) = v_1$.

ii) $C((v_1, \underline{string}(1)), \underline{char}) = v_1$.

3) Coercions with the real domain, R, and the integer domain, Z, are used to define the operators in the next section.

i) $C((v,Z), \underline{integer}) = \begin{cases} \text{if minint} < v < \text{maxint then } v \\ \text{else undefined} \end{cases}$

ii) The coercion rule for a value in the domain R to the type real states that the resultant value must belong to the set R_0 (which is the set of values of type real), and that the value should be the closest value to v in the set, unless the value of v is outside the bounds of R_0.

$$C((v,R)\underline{real}) = \begin{cases} \text{if } v \varepsilon R_0 \text{ then } v \\ \text{else if minreal} < v < \text{maxreal then} \\ v' \mid (v' \varepsilon R_0) \wedge \forall x \varepsilon R_0 [(v-v')^2 \leq (v-x)^2] \end{cases}$$

4) The extended coercion function C_a - used for assignment to scalar types.

Let T ε {logical, char, integer};

Let T' be any scalar type, such that all members of T' belong to T, then

$$C_a((v,T),T') = \begin{cases} \text{if } v \varepsilon T' \text{ then} \\ \text{else undefined} \end{cases}$$

Operators on Data Types

0) Notation

Functions are used to define the various operators in TOMAL. Let T be the set of all types, V the set of all values, and Op the set of all operators, partitioned into two subsets Op_2 and Op_1 for the binary and unary operators respectively. The evaluation functions, R_1 and R_2, are defined:

R_1: $op_1 \times (V,T) \rightarrow (V,T)$.

R_2: $Op_2 \times (V,T) \times (v,T) \rightarrow (V,T)$.

The application of R_2 is given by $R_2(Op_2,(v_1,t_1),(v_2,t_2)) = (v_3,t_3)$. It has the meaning that the result of applying Op_2, which is a binary operator, to the operands v_1 and v_2, of types t_1 and t_2, respectively, is the value v_3 of type t_3. For unary operators the function R_1, which takes as operands a unary operator and a single value-type pair, is used. The function R_1 also results in a value-type pair.

The operator evaluation function R applies to a small set of (value,type) pairs. The extension to all pairs of arguments to which an operator may be applied is obtained by using the coercion functions. The result of an operator on a set of pairs is obtained by first coercing the pairs, using the fewest possible coercions, to a set of operand pairs for which the operator evaluation function applies, and then applying the evaluation function to the coerced pairs. If the pairs cannot be coerced to a set of pairs for which the operator evaluation function is defined, then the operator is not defined for the pairs.

We make use of the following functions:

$$\max(m_1,\ldots,m_n) = m_i \quad \forall j(i\leqslant j\leqslant n) \supset m_i \geqslant mj))$$

$$\min(m_1,\ldots,m_n) = m_i \ . \ \forall j((1\leqslant j\leqslant n) \ 3 \ (m_i \leqslant m_j)).$$

The operators $*$,t,$-$ are defined as the binary operators over the domains R and Z. The operator $-$ is also negation, when it applies to a single operand, in both domains. / is division in R, <u>div</u> is integer division in Z (i.e., discard the remainder), and mod is defined in Z by:

$$x \text{ nod } y = z \text{ s.t. } (z + (y*(x \text{ div } y)) = x).$$

and, or, not are defined by the operators of and, or, not, in the following table.

A	B	A and B	A or B	not A
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	true

set complementation is defined as:

$(\neg\ v, \underline{set}\ \underline{of}\ \{a..b\}) = \{x \mid x \in a..b\} \wedge \neg(x \in v)\}$

1) Arithmetic Operators

The following rules define the arithmetic operators, Note that computation is always done in either the domain $\mathcal{Z}$ or $\mathcal{R}$. The result is then coerced to the resultant type, which depends on the types on the operands, as well as the operator. This approach easily accommodates problems arising from-an overflow or an underflow, since the result of the coercion to be applied will be undefined. Note that ($v_1\ Op_2\ v_2), X)$ where X is R or Z is used to specify a binary computation in these domains. We assume that all computations used in R and Z are defined.

Let $O_e \varepsilon\{+,-,*\}; O_i \varepsilon\{div, mod\}$, $O_b \varepsilon(O_e \cup O_i)$; $O_d \varepsilon(O_e \cup \{/\})$ and let $O_u \varepsilon$ +,-} (i.e., unary + and -).

i) $R(O_b, (v_1, \underline{integer}), (v_2, \underline{integer})) = C((v_1\ Ob\ v_2, \mathcal{Z}), \underline{integer})$

ii) $R(O_d, (v_1, real), (v_2, \underline{real})) = C((O_d\ v_2, \mathcal{R}, \underline{real})$

iii) $R(O_u, (v_1, \underline{integer})) = (O_u\ v_1, \underline{integer}$

iv) $R(O_u, (v_1, \underline{real})) = (O_u\ v_1, \underline{real})$

2) boolean Operators

Let $O_L \varepsilon\{\underline{and}, \underline{or}\}$.

i) $R(O_L, (v_1, \underline{boolean}), (v_2, \underline{boolean})) = (v_1\ O_L\ v_2\ \underline{boolean})$

ii) $R(\underline{not}, (v_1, \underline{boolean})) = (\neg v_1, \underline{boolean})$

3) Set Operators

The set operators define the resultant type based on the scalar types of the operands. Let $c_1..c_n$ be a scalar type.

a) $R(\&, (v_1, \underline{set}\ \underline{of}\ \{c_i..c_j\}), (v_2, \underline{set}\ \underline{of}\ \{c_{i'}..c_{j'}\})) =$

$(v_1 \cap v_2, \underline{set}\ \underline{of}\ \{\max(c_i, c_{i'})..\min(c_j, c_{j'})\})$

b) $R(+, (v_1, \underline{set}\ \underline{of}\ \{c_i..c_j\}), (v_2, \underline{set}\ \underline{of}\ \{c_{i'}..c_{j'})\}) =$

$(v_1 \cup v_2, \underline{set}\ \underline{of}\ \min(c_i, c_{i'})..\max(c_j, c_{j'})\})$

c) $R(-, (v_1, \underline{set}\ \underline{of}\ \{c_i..c_j\}), (v_2, \underline{set}\ \underline{of}\ \{c_{i'}..c_{j'}\})) =$

$(v_1 \cap (\neg v_2), \underline{set}\ \underline{of}\ \{c_i..c_j\})$

4) String Operator - concatenation

a) $R(!!, (v_1, \underline{string}(m)), (v_2, \underline{string}(n)) = (\{<i, t_i> | 1 \leq i < m+n \land (\text{if } i \leq m$ then $<i, t_i> \varepsilon v_1$ else $<j, t_j> \varepsilon V_2$, where $j = i+1-n)\}, \underline{string}(m+n))$

5) Comparison Operators

Let $O_c \varepsilon \{<>, <, >, =, >=, <=\}$, and let a..b be a subrange of any predefined scalar type. Let T_1, T_2 be scalar types.

a) $R(\underline{in}, (v_1, T_1), (v_2, \text{set of } T_2)) = (v_1 \varepsilon v_2, \underline{boolean})$

b) let t be any scalar type, real, or any set type.

$R(O_c, (v_1, t), (v_2, t)) = (v_1 O_c v_2, \underline{boolean})$.

c) $R(=, (v_1, \underline{string}(m)), (v_2, \underline{string}(n))) = (m=n \land \forall i(\ (1<i<m) \land$

$(<i, v_i> \varepsilon v_1)) \supset (<i, v_j> \varepsilon v_2)), \underline{boolean})$.

d) $R(<>,(v_1,\underline{string}(m)),(v_2,\underline{string}(n))) = (\neg R(=, (v_1,\underline{string}(m)),$

$(V_2,\underline{string}(n))), \underline{boolean})$.

e) $R(>,(v_1,\underline{string}(m)),(v_2,\underline{string}(n))) =$

$(\underline{if}\ n{=}0 \vee (v_1(1,1) > v_2,(1,1))\ \underline{then}$ true $\underline{else}$ [if $m>0 \wedge . v_1(1,1) =$

$v_2(1,1)$ $\underline{then}$ $R(>, V_1(2,m-1),\underline{string}(m-1)), (v_2(2,n-1), string(m-1)))]$

$\underline{else}$ false, $\underline{boolean})$.

This is a recursive rule which compares the first two characters (obtained by the substring operator) and if they are equal reduces the length of the strings by one and recompares. Eventually, either one string has a larger character, or one string is shorter and its characters exhausted, and the result of comparison is determined.

f) $R(>=,(v_1,\underline{string}(m)),(v_2,\underline{string}(n))) = (R(>,(v_1,string(m)), (v_2, string(n)),$

$\underline{boolean}) \vee R(=, (v_1,\underline{string}(m)),(v_2,\underline{string}(n))), \underline{boolean})$.

g) $R(<,(v_1,\underline{string}(m)),(v_2,\underline{string}(n))) = (\neg R(>=,(v_1,\underline{string}(m)),$

$(v_2,\underline{string}(n))),\underline{boolean})$.

h) $R(<=,(v_1,\underline{string}(m)),(v_2,\underline{string}(n))) = (\neg R(>,(v_1\underline{string}(m)),$

$(v_2,\underline{string}(n))),\underline{boolean})$.

6) Denotational Semantics for Scope Rules and Name-Type Bindings

This section defines a set of rules for binding names to types. The rules demonstrate a different approach to name-type binding from previous work. These rules specify scope definition and give semantics to declarations of names and types.

The binding of program identifiers and constants to type is represented by an order of pair: (name, type), where name is an identifier or constant.

Type can be any valid data type or either one of the distinguished constants proc (indicating a procedure) or tsk (indicating a task).

These rules bind names to declared types. Because the language definition does not include the concept of nested blocks, only procedures and tasks form new scopes. The procedure and task proof rules can not introduce a conflict in variable names. The axioms require global and local variable names to be distinguished within a scope. This ability is provided by the scope rules.

A function F, called the binding function, maps a program (or program segment), P, into a new program segment. The effect of F on P is to bind a set of names in P to pairs of the form [name, type]. The result of applying the binding function of a program to that program is a new program different in that every name is bound to a type. Thus all names are replaced by a pair [name, type]. The new program displays the semantic characteristics of the declarations and scope contained in the original program.

F is formed by a composition of functions.

$$F = f_1 * f_2 * \ldots * f_n,$$

where each f_1 is a mapping on a P, which affectsone name in P , mapping it to a single name-type pair. In what follows a lambda-calculus will be used to describe the effect of F on a program segment. Finally, rules for constructing F will be given.

Rules for Applying Variable Bindings

Let P be any program segment whose binding function is F. This function binds names in P to associated types. Suppose the effect of F is specified by the set of bondings:

$$x_1 \rightarrow (x_1, t_1)$$

$$\ldots$$

$$x_n \rightarrow (x_n, t_n).$$

Then F applied to P is given by:

$$F(P) \equiv \lambda x_n \ldots \lambda x_n \; P[(s_1,t_1),\ldots,(x_n,t_n)]$$

It is important that F operates only on names free in P since F is the effect of the declarations of $x_1,\ldots,x_n$ on P. After applying all binding functions (including those for constants and quantified sets, given in a following section), any name left unbound to a type (i.e., free in a bound program) is considered undeclared and therefore in error. The terms local and global are defined, for use elsewhere, with respect to name bindings. If P is a statement block whose binding function is F, then a name x is called <u>global</u> within P if x is free in F(P); x is <u>local</u> to P if x is free in P and bound in F(P).

<u>Rules for Constructing Binding Functions</u>

The binding functions are constructed around program segments which might contain declarations. A <u>program component</u> is any one of the following:

. . 1) a procedure - that is, a segment with the syntax:

procedure <identifier>.. .; <declarations><statement>...<statement> end

or

function <identifier>...; <declarations><statement>...<statement> end

2) a task - that is, it has syntax:

<identifier>:task...; <declarations><statement>...<statement> end

In the following section the construction for the binding function, F, is given. P_1 and P_2 are assumed to be instances of program components.

1) Composition Rule

Let P be any sequence of program components $P_1;P_2$. Let the binding functions for P_1 and P_2 be F_1 and F_2, respectively. Then:

$$F(P) = F_1 * F_2(P_1;P_2)$$

This rule states that the variable bindings to be applied to two program segments are a composition of the two respective bindings.

2) Rules for Constructing a Binding

Let P be a nonempty suffix of a program component, P_1. Let F be the binding function for P. Then the following rules apply.

a) F(end) = ϕ - there is no binding for an end.

b) F(<statement> P) $\equiv$ <statement> F(P) - a statement does not affect the binding.

c) F(var $x_1,\ldots,x_n$:T; P) $\equiv$

$\lambda x_1 \ldots$ Ax, F(var $x_1,\ldots,x_n$: T; P)[$(x_1,T),\ldots(x_n,T)$]

A declaration binds all the names declared to the declared type. The binding is applied to the entire program component.

d) F(type $T_1,\ldots,T_n$ = T; P) $\equiv$ $\lambda T_1 \ldots \lambda T_n$ F(type $T_1,\ldots,$ T;P)

[$(T_1,T),\ldots,(T_n,T)$]

e) If procedure A($a_1,\ldots,a_n$);P - is an instance of a P_1, then:

F(procedure A($a_1,\ldots,a_n$); P) $\equiv$ λA(procedure A($a_1,\ldots,a_n$); P)[A,proc)].

The procedure heading loses all bindings of local variables and includes only the procedure name. The type checking of parameters is done by the axioms for procedure invocation.

f) If function A($a_1,\ldots,a_n$) returns(T); P - is an instance of a P_1, then:

F(function A($A_1,\ldots,a_n$) returns (T); P) =

λA(function A($a_1,\ldots,a_n$) returns (T); P) [A,T)].

g) If A:task...; P - is an instance of a P_1, then:

F(A:task...,P) $\equiv$ λA(A:task...; P) [A,tsk].

h) The module rule: If P is a sequence of program segments with binding function F, and F $\equiv$ k->(x_k,t_k) for k=1,...,n; then F(module M exports($x_i,\ldots,x_j$) P) $\equiv$ $\lambda x_i,\ldots,x_j$ (module M exports

($x_i,\ldots,x_j$) P) [$(x_i,t_i),\ldots,(x_j,t_j)$], where 1 <i,j<n. This

rule states that only explicitly exported names appear outside a module.

Constants and Quantified Sets

Rather than binding constants to type we shall rely on the axioms for data types. This has the clear advantage of simplicity, since the data type axioms already specify the types for constants. From the data type axioms we conclude that every constant is a member of one or more types. The type of smallest cardinality which contains the constant can always be used for the type of the constant. We assume that the types of constants are pre-etermined and bound so that the binding function does not affect the names of constants. An additional binding function is required for quantified sets since they introduce a bound identifier. A quantified set has the form:

{<identifier> in <set expression> | <expression>}

Let f be the binding function for the quantified set. Let the <identifier> be x; let the type of <set expression> be t. Then f has the form:

$f: x \rightarrow (x,t)$.

And f is applied as:

Xx {x in <set expression> | <expression>} [(x,t)].

7. Constraints on the Use of Concurrency

In this section we concern ourselves with the constraints which we must place on concurrent execution to ensure the proof rules. The three subsections are concerned with the actual constraints, their necessity, and methods of ensuring the constraints by syntax.

The sufficiency of the constrains has been demonstrated [Hennessy 77] by proving the consistency of the axiomatic semantics and the interpretative definition under the restrictions on execution sequences which the constraints impose.

Before proceeding we require the concept of a live variable.. Basically a variable is live at a program statement if some executing task is currently using the variable. (For a more detailed definition, see Appendix 2.)

Constraints to Ensure the Applicability of the Axiomatic Definition

Constraint 1 - If a variable is live for a task T at statement A, then no other task may update the variable while T is executing at statement A.

Constraint 2 - If Q is a global procedure in a program P, then no two tasks can execute within the body of Q simultaneously. That is, there can be no pair of tasks sharing a procedure concurrently.

Constraint 3 - Let $T \varepsilon \{T_1,\ldots,T_m\}$, where $T_1,\ldots,T_m$ are all protected by a common semaphore, S, then:

a) Whenever a task protected by S is requested, S must be locked.

b) No task protected by S is ever requested while any task protected by S is active.

c) For every statement of the form:

with...S....do A

Either a single request statement is executed within A, for a task protected by S, or no request for a task protected by S ever occurs.

The Informal Necessity of the Constraints

Constraint 1 - Suppose variables could be updated when they were live; clearly the rule of composition would not hold.

Constraint 2 - If two tasks execute the same procedure concurrently, an update of a live variable occurs if the procedure does any assignment. This constraint is needed because the definition of TOMAL does not require the code of a procedure body to be reentrant.

Constraint 3 - Consider Constraint 3a and suppose S is free at the beginning of the following:

```
request T;
with S do A
```

then the execution of A could begin with T still in execution, and Inv_S would not necessarily be true. This would violate the axiom for with statement.

Consider Constraint 3b and the following program segment (with T,T´ protected by S):

```
    with S do begin
        request (T),
        request (T´)
end
    with S do A
```

The segment could begin execution of A with either T or T'still executing (since either one could free S); as in the case for 3a, Inv_S would not be ensured.

Consider Constraint 3c and the following two tasks executed concurrently, with Task T protected by S:

```
T1:task;
    ...
    with S do begin
        S1;
        if p then request T;
        S2;
    end
    ...
end T1;
T2: task:
    ...
    request T;
    ...
end T2;
```

There are two cases: suppose p = false, then if task T_2 executed its request after the with statement in T_1, then T might execute, violating Inv_S.

If p = true and the request statement in T_2 was executed while T_1 was executing S_1, the variables in Inv_S could be updated unknown to T_1, creating a possible violation of the axioms. Therefore, the request statement in task T_1 must execute before any request for a task bound to S.

Static, Syntactic Conditions that Ensure the Constraints

The above constraints are checkable by flow analysis within the TOMAL language processor. However, to assist the programmer in program construction and provide syntactic constraints we give static constraints which are easily checkable.

Constraint 2 is ensured if every call to a shared procedure appears within a critical region protected by a common semaphore.

Constraint 3 is ensured if every request to task t, where t is protected by S, occurs in the following context:

```
with ...S do begin
  A
  request ;
  ...
end
```

Furthermore, if task t is initially requested, then S is locked initially, and only one task protected by S is initially requested.

Constraint 1 is ensured if every variable v is safe wherever it is updated, and v is either safe wherever referenced or else v is protected by a semaphore S and is referenced only in a segment of the form A above.

8. Concluding Remarks

In this paper we have presented the sequential and concurrent semantics for TOMAL. The major contribution of this work is to demonstrate the application of semantic methods to supply a formal definition, which is primarily axiomatic, for an entire, significant programming language. There are several steps and results upon which the entire definition rests.

The rules for the sequential features utilize the double consequent verification formula to concisely define statements such as: case, break, and return. Although it has not been proven, we believe that the proof rules for the three types of TOMAL procedures are consistent and complete. Although the procedures are nonrecursive and prohibit aliasing by their definition, we do not impose other restrictions, unlike previous axiomatizations [Hoare 73, Donahue 75].

The proof rules for concurrent execution are based on the work of [Hoare 72b, 74, Owicki 75]; let us summarize the new contributions. The synchronization primitive supplied in semaphores is different; the proof rules must must account for this. The concept of a domain assertion is introduced and used for a synchronization proof rule similar to that given by Owicki (critical sections) and in the rules for tasks, where they differ from previous work and are more closely related to monitors. Most importantly, we specify a set of constraints which permit the proof rules to be used, without being overly restrictive. A primary example of this difference is relaxation of the strict requirement of mutual exclusive access to variables which has appeared in previous proof rule systems.

Although it was not our aim to design a language according to its proof rules, the rules for concurrent features proved to be a useful input into the design of the synchronization mechanisms. When we encountered difficulty in selecting appropriate synchronization mechanisms, the concurrency proof rules assisted in selecting the necessary features. The proof rules showed that overly powerful synchronization primitives were both hard to define and possessed no great advantage.

The complementary denotational semantics explicitly associates types with variables and constants, and provides rules for type-correctness both in expressions and in assignment. It is also worthwhile to note that the denotational semantics may be used within the framework of the axiomatic semantics, particularly in the verification of programs containing features outside the domain of the axiomatics. The denotational and axiomatic semantics together accomplish the goal of supplying a definition for all concurrent and sequential features.

An operational semantics extends the axiomatic definition to account for time features. Primarily, the operational definition provides semantics for task scheduling, accounting for both priority and time dependencies. It also completes the language definition whenever the constraints fail to hold for a program.

Two major questions arise: how does one decide when certain segments of a programming language should be defined by different methods, and how can a suitable definition method be chosen? The best answers that we can supply to these questions come from our experience in attempting to provide a formal definition for TOMAL.

Since verification, comprehensibility, and compactness were among our primary goals we strove to utilize the axiomatic method wherever possible. The first obstacles to such an approach were the break and return statements. Because these statements occupy an integral part of the language design, we choose to utilize the extended (i.e., double consequent) form for the axiomatic semantics.

The definition of the concurrent and real-time language features encountered two major difficulties. First, the notion of time dependencies and priority scheduling did not adapt well to the axiomatic method. Several possible schemes for defining these features were investigated. An interpretative method of semantic specification was chosen because it appeared best suited for defining the notions of time-dependent scheduling which are a vital part of this real-time language.

The second difficulty arose because we did not wish to restrict concurrent execution sequences with a structure such as monitors. This was based on the view that such a decision may be dangerous in a real-time environment (a similar view is advocated in Modula). However, we felt a need to extend the axiomatic definition to cover as much of the concurrent language aspects as possible. Hence, we devised a set of compile-time testable conditions which allow a language processor to determine if the axiomatic definition can be utilized. We also constructed a set of more restrictive, syntactic tests for the applicability of the axiomatic definition. These tests can be checked in an ordinary compiler. When a program does not abide by these restrictions, the interpretative definition supplies semantics. As pointed out by Donahue [Donahue 75] the consistency of these two complementary definitions is vital, and provides an additional argument for the correctness of the definitions.

Our major goal of providing semantics for <u>all</u> aspects of the language caused a great deal of concern in regard to the definition of coercion and operator application. The lack of suitable definitions for these areas of a programming language is burdensome to the language user and clearly unnecessary. We found that we could define these features, including concepts such as overflow, in a meaningful manner, which is as simple as the informal definition normally given. Our approach allows the utilization of this method with the axiomatic definition to form a basis for program verification.

The last segment of the language to be defined is that of scope rules and variable definitions. Some efforts to define these concepts have been attempted [Cook 75, Fokkinga 77], utilizing the concept of unique names and environments.

We had several goals in this segment of the definition: define scope for names, define the binding of names to types, and supply the definition in such a way that it can be separately applied from the axiomatic semantics. The last goal reflects the fact that verifying a program would be easier if one could apply scope and binding rules once, as a single separate step. These aims led us to the present definition which we believe is intelligible, and easy to employ.

Thus our effort to define TOMAL proceeded in a series of steps, each one increasing the coverage of the forma definition. Naturally, there is a danger in this approach; the separate definitions may not be compatible in fact, they may be inconsistent. The consistency of the overlapping segments of the definitions (concurrent execution) has been proven [Hennessy 77]. The question of compatibility is one of aesthetics; we feel that this definition provides a good framework for both verification and implementation.

One significant benefit of formal definition is its assistance in the language design. As each language component is defined it forces the designers to think about that feature and come to agreement on its meaning (in some cases the agreed upon meaning and the definition are different). Similarly the designers must choose between implementation independent features and those which are left undefined, for the implementation. Although, the formal definition requires considerable effort, the process is an invaluable component of the larger process of designing a new language and should not be overlooked.

APPENDIX 1

Substitution in Assignment Axioms

In order to define possible coercions executed in an assignment statement and the meaning of assignment to a subscripted variable, an extended definition is supplied for substitution. This form of substitution is utilized in all assignment axioms. The definition relies on data type axioms and coercion rules.

The definition of the substitution P_y^x is defined by the form of the strings involved.

1. If x is not an indexed array or a string, then

   $$P_y^x \text{ means } P^x_{C_a}((y,T_y),T_x)$$

2. If x is an indexed array expression of the form A(i), and A has component type T_0, then

   $$P_Y^{A(i)} \text{ means } P^A_{A-\{<i,val(T_0)>\} \cup \{<i,C_a((y,T_y),T_0)>\}}$$

3. If x is a string or substring and y is coercible to type char, then

   $$P_y^x \text{ means } P^x_{x-\{<1,val(char)>\} \cup \{<1,C_a((y,T_y),char)>\}}$$

4. If T_x is <u>string</u>(m) and T_y is <u>string</u>(n), then

   $$P_y^x \text{ means } P^x_{(x-\{<i,val(char)> \mid 1 \leq i \leq \min(m,n)\} \cup \{<j,y_j> \mid 1 \leq j \leq \min(m,n)\},\ string(m))}$$

5. If x is a substring of the form A(m,n), TA is <u>string</u>(p), and T_y is <u>string</u>(r), then

P^x_y means $P^x_{(x - \{<i, \text{val (char)} \mid m \leqslant i \leqslant \min (m + \min(n,r), P)\}}$

$\cup \{< j, y_j > \mid 1 \leqslant j \leqslant \min (n,r,p-m)\}, \text{string } (p))$

APPENDIX 2

Definition of Update and Variable Liveness

A variable v is updated in a statement A, if:

1) A is an assignment statement and x would be substituted for when applying the axiom of assignment to A.

or

2) A is a call statement and x is an array or string parameter and the formal parameter corresponding to x is updated by any statement in the called procedure.

or

3) A is a for statement and x is the control identifier.

A variable v is live at statement A for task t, if:

1) task t executed a statement A' prior to A, which referenced v,

and

2) task t has not executed a request or do with statement between A' and A.

REFERENCES

[Cook 75)	Cook, S., Axiomatic and Interpretive Semantics for Algol Fragment, Technical Report 79, Dept. of Computer Science, University of Toronto, (February 1975).
[Donahue 75]	Donahue, J. E., Complementary Definitions of Programming Language Semantics, Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto (1975), also Spinger Verlag Lecture Notes in Computer Science #42, New York.
[Fokkinga 77]	Fokkinga, M., Axiomatization of Declarations and the Formal Treatment of the Escape Construct, Proceedings of IFIP Conference on Formal Definition of Programming Concepts, North Holland Publishing Co., August 1977.
[Hennessy 75]	Hennessy, J. L., R. B. Kieburtz , and D. R. Smith, TOMAL: A Task-Oriented Microprocessor Applications Language, IEEE Trans. IECI 22, 3 (August 75), 283-289.
[Hennessy 77]	Hennessy, J. L., A Real-Time Language for Small Processors: Design, Definition and Implementation, Ph.D Thesis, Dept. of Computer Science, SUNY at Stony Brook, also Technical Report #73 (August 77).
[Hoare 69]	Hoare, C. A. R., An Axiomatic Basis for Computer Programming, CACM 12, 10 (October 1969), 576-580.
[Hoare 72a]	Hoare, C. A. R., Notes on Data Structuring, in Structured Programming, Academic Press, London, 83-174
[Hoare 73]	Hoare, C. A. R. and N. Wirth, An Axiomatic Definition of the Programming Language Pascal, Acta Informatica 2, 4 (1973)) 335-355.
[Hoare 74]	Hoare, C. A. R. and P. Lauer, Consistent and Complementary Definitions of Formal Semantics of Programming Languages, Acta Informatica 3, 2 (1974), 135-154.
[Hoare 72b]	Hoare, C. A. R., Towards a Theory of Parallel Programming, in Operating System Techniques, Academic Press, London (1972).
[Kieburtz 76]	Kieburtz, R. B., and J. C. Cherniavsky, Axioms for Structural Induction on Programs Containing Block Exit, Proc. MRI Sym. on Software Engr., Polytechnic Inst. of New York (1976).
[Lucas 71]	Lucas, P., P. Lauer, and H. Stigleittner, Method and Notation for the Formal Definition of Programming Languages, IBM Technical Report 25.087, IBM Laboratory Vienna (1971).

[Nassi 74] Nassi, I., and E. Akkoyunlu, Verification Techniques for a Hierarchy of Control Structures, Tech. Rpt. #26, Dept. of Computer Science, SUNY at Stony Brook (1974), to appear SIAM Journal on Computing.

[Owicki 75] Owicki, S. S., Axiomatic Proof Techniques for Parallel Programs, Ph.D. Thesis, Dept. of Computer Science, Cornell University (1975).

[Vantilborgh 72] Vantilborgh, H., and A. vanLamsweerde, On an Extension of Dijkstra's Semaphore Primitives, Information Processing Letters 1 (1972), 181-186.

[Wirth 71] Wirth, N., The Programming Language Pascal, Acta Inforrnatica 1, 1 (1971), 35-63.

[Wulf 71] Wulf, W. A., D. B. Russell, and A. N. Habermann, BLISS: A Language for Systems Programming, CACM 14, 12 (Dec. 71), 780-790.