A Comparison of Formal Definitions of Ada Tasking

Xudong He and J.A.N. Lee

TR 89-4

A COMPARISON OF FORMAL DEFINITIONS OF ADA TASKING

March 20, 1989

Xudong He
J.A.N. Lee

Department of Computer Science

Virginia Tech

Blacksburg, VA 24061

A Comparison of Formal Definitions of Ada Tasking

Xudong He and J.A.N. Lee

Department of Computer Science
Virginia Polytechnic Institute & State University
Blacksburg, VA 24061, U.S.A.

Abstract

In this paper, various formalisms (the operational approach, the denotational approach, the axiomatic approach, Petri nets and temporal logic) are briefly introduced through their applications in formally defining Ada tasking, and their coverages and relative merits are then compared. Therefore this paper serves two purposes: (1) to review the current status of formal definition of Ada and (2) to propose future research direction in applying formalisms in language definition and software development.

1. Introduction

There are many experimental formal definitions of Ada[1] subsets, especially the concurrent portion -- tasking, which is also considered to be the hardest of the language to describe. The formalisms used include: the operational approach, the denotational approach, the axiomatic approach, Petri nets, and temporal logic ([9], [11], [14], [17], [20], [26], [28], [38]). The definitions differ greatly in their appearances and abstract levels from very formal and mathematical to semi-formal and graphical, and from very high level and user oriented to very low level and implementation oriented.

Since the definition of a language serves many different purposes (verification, implementation etc.) and different professionals (language designers, language theoreticians, language implementers, system programmers and application programmers), it will be very useful to make a comparison of various definition techniques and to point out their relative merits so that they can be applied appropriately and effectively. There is also a need to view the current status of formal definition of Ada since it is not just another programming language [40] and to see how to bridge the gap between formal specification and verification [33].

In this paper, several formal definitions of Ada tasking are briefly reviewed and their coverages are discussed. The formalisms used in the definitions are also introduced and their relative merits are compared. In Section 2, various tasking related concepts are introduced. In Section 3, various formalisms and their applications to the formal definition of Ada tasking are reveiwed. In Section 4, the formalisms are compared and future research direction in the formal definition of programming languages and in applying formalisms in software development process is proposed.

---

[1] Ada is the trademark of the Department of Defense of USA

2. *Basic Concepts of Ada Tasking*

Task units are the only form of program units of Ada which execute concurrently. A task unit consists of a specification and a body, with the syntax shown below [3]:

```
task_declaration ::= task_specification;

task_specification ::=
    TASK [TYPE] identifier [IS
         {entry_declaration}
         {representation_clause}
    END [task_simple_name] ]

task_body ::=
    TASK BODY task_simple_name IS
         [declarative_part]
    BEGIN
         sequence_of_statements
      [exception
         exception_handler
         {exception_handler}]
    END [task_simple_name];
```

2.1. Task Creation, Execution and Termination

Tasks can be defined by task declarations or by defining task objects in terms of existing task types. They are created by elaboration of the declarations or by evaluation of an allocator respectively. The execution of a task is activated by the elaboration of the declarative part of its body which defines the execution of any task of the corresponding task type. Execution of different tasks proceeds in parallel.

Each task has at least one parent on which it depends. The direct dependency relation is defined by following rules [3]:

- The task designated by a task object which is the object, or a subcomponent of the object, created by the evaluation of an allocator depends on the parent that elaborates the corresponding access type definition; and
- The task designated by any other task object depends on the parent whose execution creates the task object.

Therefore, a non-cyclic tree structure of dependency relations is expected.

A task may *complete* its execution in two situations:

1. when it has finished the execution of its body; or
2. when an exception is raised during its execution.

The *termination* of a task takes place:

1. when it has completed its execution and it does not have any dependent task; or
2. when it has completed its execution and all of its dependent tasks have terminated.

2.2. Synchronization and Communication (Rendezvous)

Tasks can synchronize and communicate each other through entry calling and acceptance, which is known as *rendezvous*. The syntax of entry declaration, entry call statement and accept statement is shown below [3]:

```
entry_declaration ::=
   ENTRY identifier [(discrete_range)] [formal_part];

entry_call_statement ::=
   entry_name [actual_parameter_part];

accept_statement ::=
   ACCEPT entry_simple_name [(entry_index)] [formal_part] [DO
        sequence_of_statements
   END [entry_simple_name] ];

entry_index ::= expression
```

Synchronization takes place in the following situations:

1. if the calling task issues an entry call statement before a corresponding accept statement is reached by the task owning the entry, the execution of the calling task is *suspended* until the calling is accepted; or
2. if a task reaches an accept statement prior to any call of that entry, the execution of the task is *suspended* until such a call is received.

The *communication* takes place when both calling and called tasks has reached the corresponding entry call statement and a accept statement. The accept statement is then executed by the called task. The interaction is known as a *rendezvous*. Entry queues are used when more than one tasks call the same entry before a corresponding accept statement is reached.

2.3. Real-Time Construct (Delay Statements)

The execution of a task can be delayed by executing a delay statement which has the following syntax [3]:

```
delay_statement ::= DELAY simple_expression;
```

The type of simple_expression must be of the predefined fixed point type DURATION and its value is in seconds.

2.4. Select Statement

Both accept statements and entry call statements can be made selective. The syntax of select statement is [3]:

```
select_statement ::= selective_wait | conditional_entry_call |
                     timed_entry_call
```

The alternative accept statements can be guarded, an alternative is said to be *open* if it is not guarded or if the guarded condition becomes true. For an entry call issued by a conditional entry call, if a rendezvous is not immediately possible, the call is then canceled. When an entry call is

issued by a timed entry call, and a rendezvous is not started within a given delay, then the call is canceled.

2.5. Other Task Related Issues

Other task related issues such as *priority*, *abort* statement, *shared variables*, and *exception handling* will not be discussed here. They are either not being in all the following formal definitions or too difficult to be formalized ([16], [25]).

3. Various Formal Definitions and Their Coverages

There are various formal definitions of the Ada tasking concept for different Ada versions ([1], [2], [3]). The formalisms used can be classified as: the operational approach ([9], [26]), the denotational approach ([17], [27]) , the axiomatic approach [20], Petri nets ([14], [28]), and temporal logic [38].

In the following sections, we discuss the coverage of different definitions and the relative merits of different formalisms.

3.1. The Operational Approach to Ada Tasking

The *operational approach* is the earliest formalism developed for specifying programming language semantics. It is based on the theory of computation processes (state transitions), where the meanings of programs are defined in terms of state transitions over an *abstract machine* which interprets the programs. In [9], a multi-processing implementation-oriented formal definition of preliminary Ada was described in SEMANOL.

3.1.1. The SEMANOL System, Definition and Notation

The SEMANOL system consists of:

- a meta language for defining formal operational definitions of the syntax and semantics of programming languages; and
- an interpreter which executes SEMANOL metaprograms.

A SEMANOL metaprogram definition of a programming language L has the following parts:

- the context-free syntax of L (in a BNF-Like notation);
- the static semantics; and
- the dynamic semantics of control defined by both applicative SEMANOL definitions and executable imperative SEMANOL command statements.

The SEMANOL notations are very similar to those in a high level procedural programming language. Its style is procedural and constructive. The following commands are most frequently used:

- ASSIGN_VALUE! x=y has the normal interpretation of assignment statement as in most high level programming languages;
- COMPUTE! x causes (meta) evaluation of the expression x; and
- CO-COMPUTE! causes each of the processes to execute concurrently.

3.1.2. The SEMANOL Definition of Ada

In [9], both sequential and concurrent structures of Ada [1] were defined. The definition was given in two parts labelled by SEMANTIC-DEFINITIONS and CONTROL-COMMANDS respectively. The first part provided the semantics of elaboration and evaluation (DFs and PROC-DFs) and the second part controled the execution of metaprogram. The excerpt from the SEMANOL definition is:

```
CONTROL-COMMANDS:

BEGIN
  ASSIGN-VALUE! Ada-program-text =
                              lexically-transformed(total program)
  ASSIGN-VALUE! program-tree = CONTEXT-FREE-PARSE-TREE
                 (Ada-program-text,[complete-compilation])
  IF [program-tree] is-not-statically-valid THEN STOP!
  COMPUTE! initialize-system-state (semanol-processor-list)
  CO-COMPUTE! control IN-PROCESSES (semanol-processor-list)
END.
```

The first two statements create a syntactically correct abstract tree from an Ada program; then static semantics checking is performed followed by the initialization of system state (tables, semophores etc.); finally, CO-COMPUTE! activates the concurrent execution of meta processes.

Each meta process had the following definition:

```
PROF-DF control
  WHILE TRUE
  BEGIN
    COMPUTE! op-kernel
    WHILE sequential-processing-may-continue
    BEGIN
      COMPUTE! effects-of (current-step)
      COMPUTE! update-to-successor(current-step)
    END
  END
```

A process executes sequentially until one of the following occurences happens: a rendezvous, an abort, an exception or an interrupt; then the process calls the kernel by executing COMPUTE! op-kernel.

The definition acted like a multi-pass compiler and the op-kernel served as the kernel of an operating system. The concurrect execution of tasks was modeled by a multi-programming system running on a fixed number of processes with a shared memory. Rendezvous between tasks was simulated by P and V operations for synchronization, and ordinary parameter passing for communication. A piece from the definition is:

```
PROC-DF entry-call-effect-of (step)
BEGIN
  ASSIGN-VALUE! entry-name = dynamic-entry-name (name (step))
  ASSIGN-VALUE! owner = owner-task (entry-name)
  COMPUTE! locked-enque ([:current-task,param-part(step):],
           "on" entry-q (entry-name))
  COMPUTE! update-to-successor (current-step)
  ASSIGN-VALUE! non-sequential-condition = 'entry-call'
END.
```

```
PART-DF op-kernel
  =: complete-entry-call IF [non-sequential-condition]
                            is-entry-call

PROC-DF complete-entry-call
  P! semaphore (delay-list)
  P! semaphore (entry-gates(owner))
  IF [entry-name] is-open THEN
    BEGIN
      COMPUTE! enque-ready-task(owner)
      COMPUTE! close-entries(owner)
      V! semaphore (entry-gates(owner))
      COMPUTE! locked-decrement(running-tasks)
      COMPUTE! scheduler
      RETURN-WITH-VALUE! NIL
    END
  V! semaphore (delay-list)
  COMPUTE! scheduler.
```

The execution of entry-call-effect-of is invoked from the COMPUTE! effects-of in the PROC-DF of the process controling the calling task, and it puts the name of the calling task in the entry queue; sets the condition to be 'entry-call'; and leaves the sequential execution (disconnects the process from the calling task). Then the op-kernel starts the following actions: prevents the multiple acceptances by using two semaphores in case of a select statement has an open delay statement and/or open accept statements; puts the called task in the ready queue when the entry is open; calls the scheduler (COMPUTE! scheduler) for reassigning the process to some task (possiblly to the called task).

The definition was at a very low level and had many implementation related decisions such as multi-stack simulation of environment, P and V operations for synchronizations and a fixed number of multi-processes with a shared memory model for execution. These are the typical drawbacks of the operational approach which complicates the understanding and limits the application of the formal definition. Though exception raising was briefly treated in [9], no verification issues were addressed so that the correctness of the definition and the programs could not be assured. The definition was only based on the very simple case of concurrency.

3.2.The Denotational Approach to Ada Tasking

The *denotational approach* is based on Lambda-Calculus and Scott's Domain Theory ([20], [41]). In the denotational approach, entities in programming languages are mapped into mathematical objects (trees, lists, sets, functions etc.). In such a definition, three parts are identified: *syntactic domains, semantic domains and semantic functions* which map objects in syntactic domains to those in semantic domains. One important principle of the denotational approach is the compositionality in which the denotation of a composite construct is composed from the denotations of its constituents.

In [27], a formal model of the Ada [1] tasking was developed in VDM which was highly pragmatic. A more formal treatment was given in [17].

3.2.1. The VDM META-IV

The VDM approach uses a special meta-language called META-IV ([10], [11]) which is essentially syntactically sugared Lambda-Calculus combined with an imperative style and an exit mechanism. It is not purely denotational and actually is a hybrid of the denotational and operational approaches, but by convention belongs to denotational approach [10].

A VDM definition of a programming language L consists of:

- an abstract syntax (syntactic domains) which is an abstraction and refinement of the concrete BNF-Like syntax of L;
- semantic domains that are mathematically well-defined objects with types of trees, lists, sets and mappings;
- auxiliary functions (is-well-formed functions) which are predicates for checking the legality of the programs and define the static semantics of the L; and
- interpretation functions which give the meanings of the execution of programs and define the dynamic semantics of L.

Auxiliary functions and interpretation functions are called semantic functions and are mappings from abstract syntax to semantic domains.

The VDM META-IV notations are very close to the notations in high level programming languages such as Pascal and Ada. For a tree structure domain definition, the symbol :: is used; otherwise, = is used as the definition symbol. The ***mk-*** operator shows the components of a composition structure and the symbol = = > is used as a convention in the ***type*** clause of a semantic function to represent the state changing. The ***def*** x: f; e(x) combines the effects of first evaluating f and then e(x). The ***let*** command is often used in semantic functions as an imperative assignment statement. McCarthy style case statement is most often employed in the semantic functions. The ***exit*** mechanism acts like a structured goto statement [10] and serves as a substitution for the traditional *continuation* concept.

3.2.2. The VDM Definition of Ada Tasking

In [27], the META-IV was extended with a process concept and a very simple communication and synchronization concept both based on the ideas of CSP [23]. The process concept had a fixed number of process types (called processors) upon which a dynamically varying number of process instances were based. The communication concept employed input and output construct based on the value passing hand-shaking idea of CSP.

The definition had a meta-process model which consisted of the fixed processes: SYSTEM, STORAGE, TIMER, CLOCK and MONITOR. The SYSTEM process was responsible for the initialization and termination of whole system. The STORAGE process simulated the shared memory. The TIMER and CLOCK facilitated the real-time tasking. The MONITOR was the key unit of the model which performed all task creation, termination and interaction. All dynamic processes were of the TASK type which described the sequential interpretation of an Ada task.

Some simplified semantic functions for rendezvous is shown below to provide some flavor of the VDM definition, for a complete definition please refer to [28] and [11].

```
int-entry-call(tv,ev,parmassoc,ecmode) env =
  LET req = MK-EntryCall(tv,ev,parassco,ecmode) IN
 (DEF MK-EndCall(callresult):
      monitor-call(req);
      RETURN (callresult)    )
TYPE: EntryCall --> ( ENV ==> (CALLED|CLOSED|EXPIRED)
```

The interpretation function for entry call has four parameters: tv (task value containing task identifier), ev (entry identifier), parmassoc (parameters of entry call statement), and ecmode (mode indicating whether the call is conditioned or timed); and is interpreted in the environment env which associates variable names with various attributes. Upon the receiving of an entry call, the monitor is invoked by monitor-call(req) statement. The monitor process will put the above parameters in appropriate queues until a rendezvous occurs, then it will return the call result to the process simulating the calling task.

```
int-accept(MK-accept(ename,stmtl)) env =
  (DEF MK-(ev, ): eval-EntryName(ename) env;
   LET req = MK-ACCEPT(ev) IN
   DEF MK-START-MEETING( ,parmassoc) : monitor-call(req) ;
   int-acceptbody(stmtl,parmassoc)                        )
TYPE: Accept --> (ENV ==> )
```

This interpretation function will call the monitor process once an accept statement is encountered with the name of the task containing the statement. The elaboration of monitor-call and int-acceptbody is noninterruptable marked by ;. The name is also put in an appropriate queue by the monitor until a corresponding calling happens; then the body of the accept statement is executed by: int-acceptbody(stmtl,parmassoc) with the actual parameters from the calling task passed by the monitor.

This definition was at a low level and dependent on design details. The model utilized a MONITOR to create tasks and control the communications between tasks which made it only applicable to a central controlled environment. The definition was comparatively lengthy and difficult to understand. Though the definition covered almost all aspects of the tasking concept except real time issues, no verification issues were discussed. However the definition should be instructive to implementors.

3.2.3. The Concepts of Process and the Semantics of a CSP-Like Language

In [17], a more formal treatment of Ada [2] tasking in denotational semantics was given, based on the concept of *process* domains. A *uniform process* is a tree-like construct which abstracts the structure of the sequences of elementary actions generated during the execution of a program. An example of finite processes is:

$\{ \langle a, \{ \langle b,p \rangle, \langle c,q \rangle \} \rangle \}$

where a, b, c are elements of an alphabet A and p, q are uniform processes defined upon A. One distinguishable uniform process is the ***nil*** process denoted by p_0. Operations on finite processes are recursively defined as following:

- ***Composition*** *"•"* is defined by:

  $p \bullet p_0 = p,$
  $p \bullet X = \{p \bullet x \mid x \in X\},$
  $p \bullet \langle a,q \rangle = \langle a, p \bullet q \rangle;$

- ***Union*** *"U"* is defined by:

  $p \cup p_0 = p_0 \cup p = p,$
  for $p,q <> p_0$, p Uq is the set theoretic union of the sets p and q;

- ***Merge*** *"||"* is defined by:

  $p \parallel p_0 = p_0 \parallel p = p,$
  $X \parallel Y = (X \parallel_L Y) \cup (X \parallel_R Y),$
  $X \parallel_L Y = \{x \parallel Y \mid x \in X\},$
  $X \parallel_R Y = \{X \parallel y \mid y \in Y\},$
  $\langle a,p \rangle \parallel X = \langle a, p \parallel Y \rangle$, and
  $X \parallel \langle b,q \rangle = \langle b, X \parallel q \rangle.$

Processes with additional structures (synchronization, function and communication) and corresponding operations were defined individually in [17]. In the following sections, a CSP-Like language L defined in process concepts [17] is presented.

The syntax of L is:

S ::= x:=s | ***skip*** | b | S1;S2 | S1 U S2 | S1 || S2 | S* | c?x | c!s |S\c | b=>S
where x is a variable, b is an expression and c is channel name.

The semantics of L is defined in terms of λ notations - the approach originated from Oxford University (OFU) [42], where V is the set of *values* (meanings of variables and expressions), Σ = Var $\rightarrow$ V is the set of *states*, $\alpha \in V$, $\sigma \in \Sigma$, $\sigma\{\alpha/x\}$ is a state which differs from α only at x, V,W are functions which for each s, b and σ determine values $V(s)(\sigma) \in V$ and $W(b)(\sigma)$ in {true, false}:

1. $M(x:=s) = \lambda\sigma.\{ <\sigma\{ V(s)(\sigma)/x\}, p_0>\}$
   $M(\textit{skip}) = \lambda\sigma.\{ <\sigma, p_0> \}$
   $M(b) = \lambda\ \sigma.$ ***if*** $W(b)(\sigma)$ ***then*** $\{<\sigma, p_0>\}$ ***else*** $\emptyset$

2. $M(S1;S2) = M(S2) \bullet M(S1)$
   $M(S1\ U\ S2) = M(S1)\ U\ M(S2)$
   $M(S1 \parallel S2) = M(S1) \parallel M(S2)$
   $M(S^*) = \lim_i p_i$,

with p_0 as always, and $p_{i+1} = (p_i \bullet M(S))\ U\ \lambda\sigma.\{<\sigma, p_0>\}$

3. $M(c?x) = \lambda\sigma.\{<\gamma, \lambda\alpha.<\sigma\{\alpha/x\}, p_0>>\}$
   $M(c!s) = \lambda\sigma.\{<\overline{\gamma}, V(s)(\sigma),\sigma, p_0>\}$

4. $M(S\backslash c) = M(S)\backslash\gamma$
   $M(b=>S) = \lambda\ \sigma.$ ***if*** $W(b)(\sigma)$ ***then*** $M(S)(\sigma)$ ***else*** $\emptyset$

The meaning of a assignment statement is defined by the process $<\sigma\{ V(s)(\sigma)/x\}, p_0>$ which differs from the original process only at x. The meaning of an input statement c?x is defined by waiting for the value of α under port name γ. The meaning of a restriction statement S\c is defined by the process after removing the communication pairs $<\gamma,...>$ and $<\overline{\gamma},...>$.

3.2.4. The Process Based Semantics of Ada Tasking

The process based semantics of Ada tasking given in terms of the CSP-Like language in the last section is taken from [17] and illustrated in this section.

The syntax of a fragment of Ada tasking L_A is:

- Programs $S \in L_A$ are:

  $S ::= T_1 \parallel T_2 \parallel ... \parallel T_m$;

- Tasks T are:

  T ::= x:=s | ***skip*** | ***if*** b ***then*** T_1 ***else*** T_2 |
  while b ***do*** T | e(s,z) | T_1; T_2 |
  accept e(x,y) ***do*** T |
  select $b_1 \rightarrow$ ***accept*** e_1 (x_1,y_1) ***do*** T_1'; T_1''
  ▪ ... ▪
  $b_n \rightarrow$ ***accept*** $e_n(x_n,y_n)$ ***do*** T_1'; T_1''

In the definition, the parameters of entry call statement were only restricted to two with the first parameter having call-by-value mode and the second having call-by-value-result mode. The numbers of tasks and entry names were also fixed.

The semantics is as follows, where the Ada statements are overmarked by ′ for distinction.

1. $(x:=s)' \equiv (x:=s)$
   $\textit{skip}' \equiv \textit{skip}$
   $(T_1; T_2)' \equiv (T_1; T_2)$

2. (***if*** b ***then*** T_1 ***else*** T_2)′ $\equiv$ (b; T_1') U ($\neg$ b; T_2')
   (***while*** b ***do*** T)′ $\equiv$ (b;T′)*; $\neg$ b

3. $e(s,z)' \equiv e!s; e!z; e?z$
   (***accept*** $e(x,y)$ ***do*** $T)' \equiv e?x; e?y; T'; e!y$
   (***select*** ... $)' \equiv \bigcup_{i=1}^{n} (b_i = > e_i ? x_i; e_i ? y_i; (T_i')'; e_i ! y_i ; (T_i'')') \text{ U } (\neg b_1 \wedge ... \wedge \neg b_n); \Delta$
4. $S' \equiv (T_1' \,||\, ... \,||\, T_m') \setminus \{e_1, ..., e_s\}$

The meaning of entry call $e(s,z)'$ is simulated by three communication statements in L which are fairly easy to understand. The meanings of ***accept*** statement and ***select*** statement are also straight forward. The meaning of the concurrent statement: $T_1' \,||\, ... \,||\, T_m'$ is defined by simultaneously removing matched communicating pairs.

In [17], a fragment of Ada [2] tasking was modeled by a CSP-Like language. The semantics was denotational and based on the process concept. Various operations and properties for processes were defined. The treatment was very theoretical; however, no proof rules and verification issues were discussed. This definition might not be useful for implementation purpose.

3.3. The Axiomatic Approach to Ada Tasking

The *axiomatic approach* is based on the principle of Hoare Logic which has a set of axioms and inference rules. The meaning of a construct (expression, statement etc.) is defined by the *precondition* (which must be satisfied initially) and *postcondition* (which must be satisfied after the evaluation of the construct). In [20], a sound and relatively complete axiomatization of the Ada [2] rendezvous was simulated by CSP.

3.3.1. CSP and Its Proof System

CSP was developed by Hoare [23]. It contains the traditional sequential programming constructs such as assignment, branching and looping and with following concurrent constructs:

- a *concurrent* statement of the form: $[P_1 \,||...||\, P_n]$ expressing the concurrent execution of processes $P_1,...,P_n$ where each P_i refers to a statement S_i, (as denoted by $P_i::S_i$) and each pair P_i and P_j ($i \neq j$) have no shared variables;

- *input and output* commands of the forms P_j ? x (in S_i) and P_i ! y (in S_j). P_j ? x expresses a request to P_j to assign a value to the (local) variable x of P_i and P_i ! y expresses a request to P_i to receive a value from P_j. The only communication and synchronization mechanism between processes is via a matched pair input and output commands; and

- *guarded* commands which have the form B → S where B can be a Boolean expression, an input and output command, or any combination of both. A Boolean guard is *passable* if it is true; an I/O command is *passable* when a corresponding I/O command in the process addressed is ready. Both guarded *selection*: [B1 → S1 ▪ B2 → S2] and guarded *iteration*: *[B1 → S1 ▪ B2 → S2] can be built.

The proof system of CSP was developed in [6]. The idea was to use Hoare's Logic for the sequential part of the language and use two axioms for local reasoning about processes in isolation: $\{p\}\ P_i$! a $\{q\}$ and $\{p\}\ P_j$? x $\{q\}$, then apply a *cooperation test* to verify whether these pre- and post-assertions were compatible with the communication. To express this test, a *general invariant*, GI, was used to group the local reasonings for each process globally together. GI was especially used to distinguish among all communication possibilities, the syntactically (statically) matching ones and the semantically (dynamically) matching ones. Auxiliary variables were introduced to express the necessary assertions and invariants.

3.3.2. The CSP Definition and Proof System of Ada Tasking

In [20], a subset called Ada-CS was defined by the following BNF-like grammar:

program ::= ***begin*** task {task} ***end***

task ::= ***task*** task_id decl ***begin*** stats ***end***
decl ::= {entry_decl} {var_decl}
entry_decl ::= ***entry*** entry_id {formal_part}
var_decl ::= var_id_list : ***int*** | var_id_list : ***bool***
var_id_list ::= var_id {,var_id}
formal_part ::= [var_id_list] [# var_id_list]
stats ::= stat {;stat}
stat ::= ***null*** | ass_st | if_st | while_st | call_st | acc_st | sel_st
ass_st ::= var_id := expr
if_st ::= ***if*** bool_expr ***then*** stats ***else*** stats
while_st ::= ***while*** bool_expr ***do*** stats
call_st ::= ***call*** task_id.entry_id (actual_part)
actual_part ::= {expr} [# var_id_list]
acc_st ::= ***accept*** entry_id (formal_part) ***do*** stats
sel_st ::= ***select*** sel_br {v sel_br}
expr ::= "expression"
bool_expr ::= "boolean expression"
id ::= "identifier"

The subset Ada-CS mainly contains task related constructs: entry declaration, entry call statement, accept statement and select statement. The actual parameters of entry call are of two modes: ***in*** before the separator # and ***in out*** after #.

The semantics of Ada-CS was defined by axioms and inference rules in the form of Hoare Logic. A rendezvous was simulated by two CSP communication commands on the right:

	Ada-CS:	CSP
T_1 :	***call*** T_2.entry(e#x)	T_2!(e,x); T_2?x
T_2 :	***accept*** entry(u#v) ***do*** S	T_1?(u,v); S; T_1!v

The following axioms (denoted by A) and proof rules (denoted by R) were adopted:

A1. entry call:

$\{p\}$ ***call*** T.entry($\vec{e}$ # $\vec{x}$) $\{q\}$

R1. accept:

$$\frac{\{p_1\}\ S\ \{q_1\}}{\{p\}\ \textbf{accept}\ \text{entry}(\vec{u}\#\ \vec{v})\ \textbf{do}\ S\ \{q\}}$$

R2. select:

$$\frac{\{p \wedge b_1\}\ S_1\ \{q\},\ldots,\{p \wedge b_n\}\ S_n\ \{q\}}{\{p\}\ \textbf{select}\ b_1\text{: } S_1\ \text{v}\ \ldots\ \text{v}\ b_n\text{: } S_n\ \{q\}}$$

The usual assignment-axiom, the null-axiom and the if-, while-, composition-, consequence- and conjunction-rules were defined as in CSP. The communication rule is shown below:

R3. formation:

$$\frac{\begin{array}{l}\{p_1 \wedge p_2 \wedge GI\}\ S_1';\ S_1''[.]\ \{\bar{p}_1 \wedge \bar{p}_2 \wedge GI\} \\ \{\bar{p}_2\}\ S\ \{\bar{q}_2\} \\ \{\bar{p}_1 \wedge \bar{q}_2[.] \wedge GI\}\ S_2'[.];\ S_2''\ \{q_1 \wedge q_2 \wedge GI\}\end{array}}{\{p_1 \wedge p_2 \wedge GI\}\ S_1';\ \textbf{call}\ T_j\ .a(\vec{e}\ \#\ \vec{x});\ S_1'' \parallel \textbf{accept}\ a(\vec{u}\ \#\ \vec{v})\ \textbf{do}\ S_2';\ >S<;\ S_2''\ \{q_1 \wedge q_2 \wedge GI\}}$$

where the call is contained in task T_i and the accept statement in task T_j; [.] is the abbreviation of parameter substitution $[\vec{e}/\vec{u},\ \vec{x}/\vec{v}]$; $\{p_1\}\ T_i\ \{q_1\}$ and $\{p_2\}\ T_j\ \{q_2\}$. The assertion $\bar{p}_1$ specifies the condition satisfied by the variables in T_i not appearing in the actual result-parameter list; $\bar{p}_2$ [.] and $\bar{q}_2$[.] specify the precondition and postcondition that S must satisfy, $>S<$ represents the fact that S does not contain any other entry call statement or accept statement. The interpretation of the rule is if the three premises are true, then the conclusion (under the line) holds for all matching communication pairs: ***call*** in T_i and ***accept*** in T_j

In [20], a very small subset of Ada [2] was defined by the axiomatic approach. The following restrictions were imposed: there was a fixed number of tasks, no shared variables, no entry-queues, no delay statement, deadlock simulating abortion. The definition might not be useful for implementation, but it should be very useful for verification purpose. The axiomatization was proved to be sound and relatively complete. Only partial correctness was addressed.

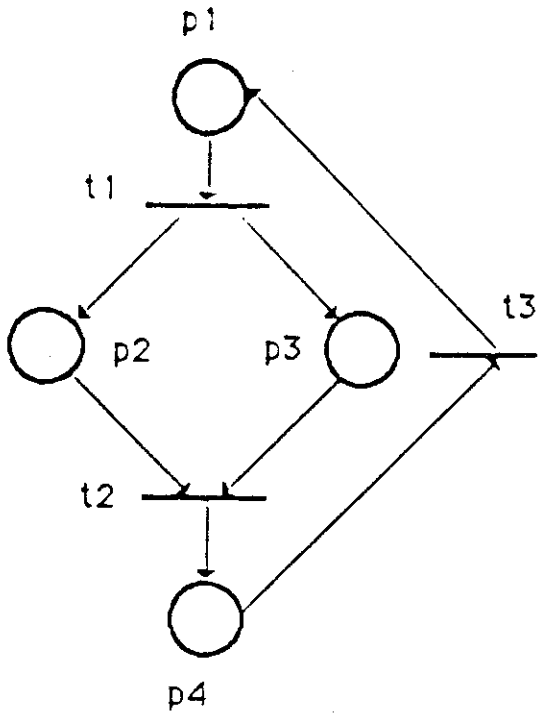
3.4. Petri Nets Applied to Ada Tasking

Petri nets are elegant tools for describing the structure of communication networks which have been widely applied in both software and hardware ([35], [39]). A Petri net can be defined as a triple (P,T,A) where P is a set of *places* which are represented by circles, T is a set of *transitions* which are represented by bars, and A is a set of *arcs* which connect places to transitions and transitions to places (there are no direct connections either between places or between transitions). A Petri net is usually *marked* with dots in various places. The execution of a Marked Petri net is by *firing* a sequence of *enabled* transitions. Time information can be associated with transitions to form *Timed* Petri nets. For each transition in a timed Petri net, a pair of time information (t_{min}, t_{max}) is given where t_{min} indicates the minimal time must elapse (after a transition is enabled) before the transition can fire and t_{max} denotes the maximum time for which an enabled transition cannot fire. If a timed transition does not fire, the timed transition is reset. A standard Petri net is a special case of timed Petri net with $t_{min} = \emptyset$ and $t_{max} = \infty$ [34]. In [39], Petri nets were further divided into *condition-event nets* (CE-nets), *place-transition nets* (PT-nets) and *predicate-relation nets* (PR-nets) with increasing expressive power. In predicates and relations nets, tokens are individualized, places are predicates and simple transitions are replaced by relation expressions. Figure 1. shows some concepts of simple CE-nets. Petri nets can be defined algebraically and be represented graphically. It is their graphical representation that makes them especially attractive.

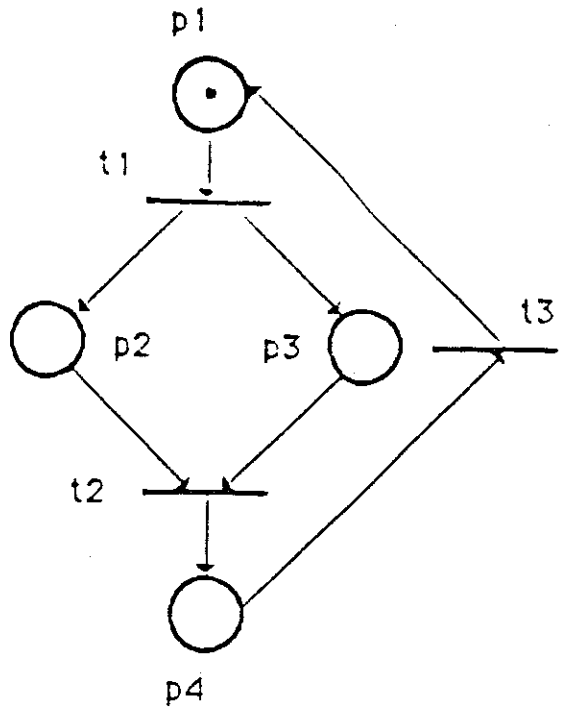
3.4.1. Petri Net Modeling of Ada Tasking

In [28], Ada [3] tasking was modeled by Petri nets. The rendezvous between two tasks T_1 and T_2 can simply be represented by Figure 2. and be interpreted by the firing rules of Petri nets: i.e. T_1 and T_2 must reach the point *start rendezvous* together to make the transition t1 to fire.

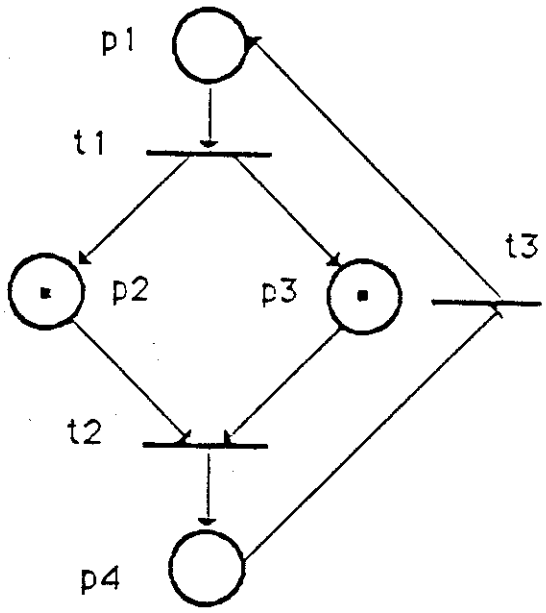
This definition covered almost all aspects of Ada tasking: task activation, task dependency, delay statement, rendezvous, exception, selective wait, and timed entry call. It was the only formal definition so far dealing with real time issues of Ada tasking. The definition was graphical and very easy to understand. The definition was not systematic nor structural and it might not be good for implementation purpose.



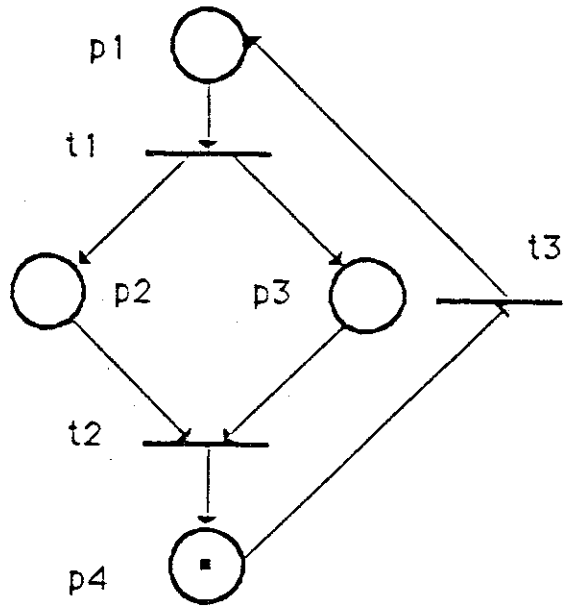
(a) An unmarked Petri net



(b) A marked Petri net: t1 is enabled



(c) After t1 fired, t2 is enabled



(d) After t2 fired, t3 is enabled

Figure 1 - Basic Concepts

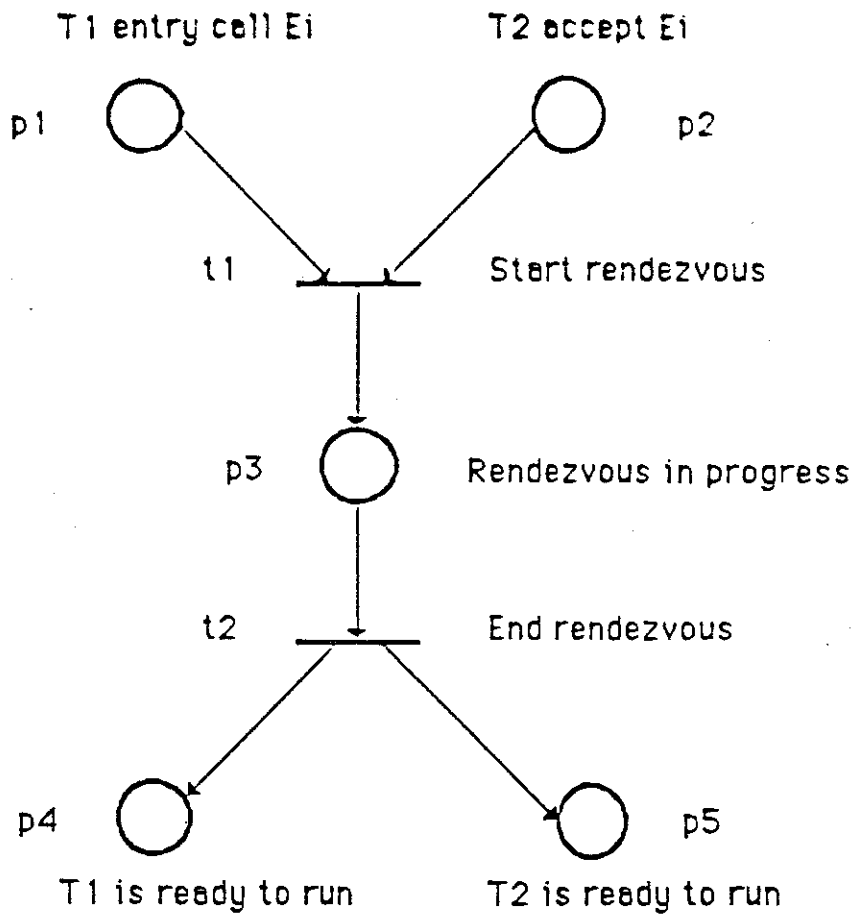


Figure 2 - A Petri net specification of Ada Rendezvous

3.5. Temporal Logic Applied to Ada Tasking

Temporal logic is a kind of *modal logic* [29] which deals with time concepts. It is the first order logic plus some *temporal operators* such as uninary operators: *always* (represented by $\Box$), *sometimes* (represented by $\Diamond$), *next* (represented by $\bigcirc$) and binary operator *until* (represented by *U*).

Temporal terms are *first order logic terms* - constants, individual variables and function applications such as $f(x_1,x_2,...,x_n)$, and $\bigcirc t$ where t is a first order logic term. The meaning of $\bigcirc t$ is the next value of t.

Temporal formulas are constructed from *first order logic formulas* (truth values, propositions, predicates and applications of negation $\neg$, disjunction v, conjunction $\wedge$, implication $\rightarrow$, identity $\equiv$, and quantifiers $\exists$ and $\forall$) and the application of the above temporal operators. For example, $\Box$ $((x > 5) \wedge (y < 10))$ is a temporal formula. A formula without temporal operators is said to be *ordinary* (or *static, classic*).

Temporal logic formulas are interpreted over sequences of program states. A *state* is a mapping which assigns values to all individual variables. Therefore, temporal logic describes the dynamic properties of a process while ordinary first order logic gives the static interpretation of a formula over a single state.

Let σ: s_0 , s_1, s_2 ,... be an infinite state sequence and $\sigma^{(k)}$ be k-shifted sequence: s_k , s_{k+1},.... Let u, v, w are ordinary first order formulas, then the meanings of temporal operators are explained below:

$\Box$ w holds on σ iff w is satisfied in all states in σ

$\Diamond$ w holds on σ iff w is satisfied by at least one state in σ

$\bigcirc$ w holds on σ iff w is satisfied by $\sigma^{(1)}$

u U v holds on σ iff sometimes v holds and until then u holds continuously.

There are three kinds of well-known temporal logics:

1. *linear time temporal logic* ([30], [37]),
2. *branching time temporal logic* [15], and
3. *interval temporal logic* [32].

Their differences are on the structures of the state sequence allowed. In linear time temporal logic, each state in the sequence can have one and only one successor state. In branching time temporal logic, each state in the sequence can have more than one successor state so that the state sequence can have a tree structure. In interval temporal logic, the temporal formula is interpreted over a specified interval of the state sequence.

3.5.1. Temporal Logic Definition of Ada Tasking

In [38], a small fragment of Ada [2] relating to rendezvous was studied under the assumption: no shared variables, no new dynamically created tasks, no procedures and subprograms, no nested blocks, and no delay statement. The definition was given in operational semantics and was based on the preliminary version of Ada. The proof rules for the definition were given in linear temporal logic and only for *just* and *fair* computation sequences. The use of temporal logic in proving correctness was demonstrated by examples. The definition explicitly dealt with entry queues and was at a low level.

In order to give some flavor of temporal logic definition, a simplified example is provided by this author. Traditionally, a state in temporal logic contains two kinds of variables *control* variables which indicate the locations during the execution of programs (usually represented by ***at***, ***after*** etc.) and normal *program* variables. Following [38], a state is defined as:

$$s = <(T_1 \textit{ at } S_1) \wedge \ldots \wedge (T_m \textit{ at } S_m); \vec{\eta}>$$

where $T_1,\ldots,T_m$ are m tasks, S_i is a sequence of statements, $\vec{\eta}$ contains the current values of program variables. Let

$$s_k = <\ldots(T_i \textit{ at } e(\vec{u}; \vec{v});S_i) \wedge \ldots \wedge (T_j \textit{ at accept } e(\vec{f}:\textit{in};\vec{g}:\textit{out});B \textit{ end } e;S_j) \wedge \ldots;\vec{\eta}>,$$

$$s_l = <\ldots(T_i \textit{ at rendezvous } e;\ S_i) \wedge \ldots \wedge (T_j \textit{ at } \vec{f} := \vec{u};\ B;\ \vec{v} := \vec{g};\ \textit{end } e;\ S_j) \wedge \ldots;\vec{\eta}>,$$

$$s_m = <\ldots(T_i \textit{ at rendezvous } e;\ S_i) \wedge \ldots \wedge (T_j \textit{ at end } e;\ S_j) \wedge \ldots;\vec{\eta}>,$$

$$s_n = <\ldots(T_i \textit{ at } S_i) \wedge \ldots \wedge (T_j \textit{ at } S_j) \wedge \ldots;\vec{\eta}>$$

then the simplified rendezvous transition between task T_i and T_j is:

$$s_k \rightarrow s_l$$

and the rendezvous termination transition is:

$s_m \rightarrow s_n$

Let ϕ_1, ϕ_2, ψ_1, and ψ_2 be temporal formulas describing state properties: s_k , s_m, s_l and s_n respectively, then the correctness of rendezvous between T_i and T_j can be expressed as:

1. $\phi_1(s_k) \rightarrow \Diamond\ \psi_1\ (s_l)$

2. $\Box\ (\ \phi_2(s_m) \rightarrow \psi_2(s_n)\ )$

where the first formula means the entry call in T_i will eventually be accepted by an accept statement in T_j and the second formula expresses that the rendezvous between T_i and T_j will always terminate correctly.

4. Discussion and Conclusion

In this paper, various formalisms and their applications in formally defining Ada tasking are briefly introduced. The coverages of various definitions and their relative merits are also discussed.

None of the described definitions cover all aspects of Ada tasking. It is especially difficult to deal with real-time issues and the verification issue has almost always been neglected. The definitions differ greatly in their appearances and abstract levels. Some are semi-formal and graphical, and some are very formal and mathematical. Some are at very high level and user oriented, and some are at very low level and implementation oriented.

The results of the comparisons of various formal definitions in terms of their applicabilities to different usages and their suitabilities for different users are summarized in Table 1 and Table 2 respectively. Values of 1, 2 and 3 have been assigned to each entry in the tables with 1 implying poor and 3 for very good. The assignments are rather subjective, being based on authors' own experiences and viewpoints.

	Reference	Validation	Verification	Implementation
Operational	1	2	1	3
Denotational VDM	2	2	2	3
Denotational OFU	1	1	3	1
Axiomatic	3	2	3	1
Petri Nets	2	3	2	1
Temporal Logic	1	1	3	1

Table 1. Formal Definitions vs. Their Usages

	Language Designer	Language Theoretician	Language Implementer	System Programmer	Application Programmer
Operational	2	1	3	3	1
Denotational VDM	2	2	3	3	2
Denotational OFU	3	3	1	1	1
Axiomatic	3	3	1	1	3
Petri Nets	3	2	2	2	3
Temporal Logic	2	3	1	1	1

Table 2 Formal Definitions vs. Their Users

Using formalisms in programming language definitions and software development has been pioneered and advocated by many well-known computer scientists ([8], [12], [24], [31], [43]). A formal semantics can achieve following properties: preciseness, unambiguity, soundness and possibly completeness and serve following purposes:

- Reference for users (helping user to write correct programs),
- Reference for implementers (helping language implementers to write correct compilers),
- Proofs of programs (making mathematical proofs or even mechanical proofs and verifications possible),
- Proofs of implementations (establishing the necessary conditions for the correctness of implementations),
- Automatic implementations (opening the possibilities of automating or partially automating the processes of constructing compilers and interpreters),
- Improved language design (inspiring language designers to develop better programming languages),
- Standardization of programming languages (increasing the languages' value as mediums of communication and tools for writing portable programs), and
- Models of computation theories (challenging computer science theoreticians to investigate new computation models and theories).

Since the definition of a programming language serves many different purposes (verification, implementation etc.) and different professionals (language designers, language theoreticians, language implementers , system programmers, application programmers), the definition should consist of different abstract levels by employing different formal methods.

There are three basic ways to specify software and in particular, programming languages:

1. By an abstract program that describes its behavior such as using the operational approach or Petri nets. This kind of specification is usually easier to understand but harder to verify (prove);
2. By mathematical functions that abstract its functional such as using the denotational approach or the algebraic approach [19]. This kind of specification is very useful in formally proving the correctness and in making equivalence transformations, but it needs training to understand; and
3. By a collection of properties that it must satisfy such as using the axiomatic approach or temporal logic. This kind of specification has a big advantage that the specification is directly verifiable (provable) in the same kind of framework.

Each different formalism has its own advantage and disadvantage and is appropriate for some specific areas. A programming language (especially the conventional language or von Neumann language such as Ada) has three distinguishable parts: control abstraction, data abstraction, and procedural abstraction which can be best defined by using Petri nets and temporal logic, the algebraic approach and the axiomatic approach, and the denotational approach and the operational approach respectively. Based on our strong belief that using a spectrum of formalisms to develop a complementary formal definition will have advantages of various formalisms, a model for integrating the above formalisms in software development has been developed in [22].

References

[1] Ada79: "Preliminary Ada Reference Manual", ACM SIGPLAN Notices, Vol.14, No.6 (Part A), June 1979.

[2] Ada81: *The Programming Language Ada Reference Manual*, LNCS[2] Vol.106, Springer-Verlag, 1981.

[3] Ada83: *Reference Manal for the Ada Programming Language*, ANSI/MIL-STD-1815A, American National Standards, 1983.

[4] Anderson, E.R., F.C. Belz and E.K. Blum: "Issues in the Formal Specification of Programming Languages", in *Formal Description of Programming Concepts* (ed. E.J. Neuhold), North-Holland, 1978, pp.1-30.

[5] Apt. K.R., N. Francez and W.P. De Roever: "A Proof System for Communicating Sequential Processes", ACM TOPLAS, Vol.2, No.3, July 1980, pp.359-385.

[6] Astesiano, E. et al.: "The Ada Challenge for New Formal Semantics Techniques", Proc. of the Ada - Europe International Conference (ed. P.L. Wallis), Edinburgh, 1986, pp.239-248.

[7] Barringer, H.: "Formal Specification Techniques for Parallel and Distributed Systems - A Short Review", Proc. of the 3rd Joint Ada Europe / AdaTEC Conference (ed. J. Teller), Brussels, 1984, pp.281-294.

[8] Bauer, F.L.: "Where Does Computer Science Come From and Where Is It Going? ", in *Formal Models in Programming* (eds. E.J. Neuhold and G. Chroust), IFIP, North-Holland, 1985, pp.31-44.

[9] Belz, F.C., E.K. Blum and D. Heimbigner: "A Multi-Processing Implementation -Oriented Formal Definition of Ada in SEMANOL", SIGPLAN Notices, Vol.15, No.11, 1980, pp.202-212.

[10] Bjorner, D. and C.B. Jones(eds.): *The Vienna Development Method: The Meta Language*, LNCS, Vol.61, Springer-Verlag, 1978.

[11] Bjorner, D. and O.N. Oest(eds.): *Towards a Formal Definition of Ada*, LNCS, Vol.98, Springer-Verlag, 1980.

[12] Bjorner, D.: "On The Use of Formal Methods in Software Develpment", Proc. of 9th Software Engineering Conference, California, 1987, pp.17-29.

[13] Burns, A.: *Concurrent Programming in Ada*, Cambridge University Press, 1985.

[14] Cherry, G.W.: *Parallel Programming in ANSI Standard Ada*, Reston Publishing, 1984.

[15] Clarke, E.M. and E.A. Emerson: "Design and Synthesis of Synchronization Skeletons Using Branching - Time Temporal Logic", in LNCS, Vol.131, Springer-Verlag, 1981, pp.52-71.

[16] Cohen, N.H.: "Ada Axiomatic Semantics: Problems & Solutions", Proc. of the Ada - Europe International Conference (ed. P.L. Wallis), Edinburgh, 1986.

---

[2] LNCS: Lecture Notes in Computer Science.

[17] De Bakker, J.W. and J.I. Zucker: "Processes And A Fair Semantics for the Ada Rendezvous", LNCS, Vol.154, Springer-Verlag, 1983, pp.52-66.

[18] Dewar, R., P. Kruchten and E. Schonberg: "What Should Be In A Formal Definition of Ada", Proc. of 8th Meeting of the Ada-Europe Formal Semantics Working Group, Bruxelles, 1983.

[19] Ehrig, H. and B. Mahr: *Fundamentals of Algebraic Specification: Equations and Initial Semantics*, Springer-Verlag, 1985.

[20] Gerth, R.: "A Sound and Complete Hoare Axiomatization of Ada Rendezvous", LNCS, Vol.140, Springer-Verlag, 1982, pp.252-264.

[21] Gouge, V.D., G. Kahn and B. Lang: "On the Formal Definition of Ada", in LNCS, Springer-Verlag, Vol.94, 1980.

[22] He, X. and J.A.N. Lee: "A Strategy for Integrating Formalisms in Software Development", Proc. of 6th CIPS Computer Conference, Edmonton, Canada, 1988, pp.33-42.

[23] Hoare, C.A.R.: "Communicating Sequential Processes", CACM, Vol.21, No.8, Aug. 1978.

[24] Hoare, C.A.R. et al: "Laws of Programming", CACM, Vol.30, No.8, Aug. 1987, pp.672-687.

[25] Lamport, L.: "What It Means for a Concurrent Program to Satisfy a Specification: Why No One Has Specified Priority", Conf. Record of the 12th Annual ACM Symposium on POPL, 1985, pp.78-83.

[26] Li, W.: "An Operational Semantics for Ada Mutitasking and Exception Handling", Proc. of AdaTEC Conf., Washington, 1982.

[27] Lovengreen, H.H. and D. Bjorner: "On A Formal Model Of The Tasking Concept In Ada", SIGPLAN Notices, Vol.15, No.11, Nov. 1980.

[28] Mandrioli, D., R. Zicari, C. Ghezzi and F.Tisato: "Modeling the Ada Task System By Petri Nets", Computer Language, Vol.10, No.1, 1985, pp.43-61.

[29] Manna, Z. and A. Pnueli: "Verification of Concurrent Programs: the Temporal Framework", in *The Correctness Problem in Computer Science* (eds. R.S. Boyer and J.S. Moore), Academic Press, 1981, pp.215-274.

[30] Manna, Z. and A. Pnueli: "How to Cook a Temporal Proof System for Your Pet Language", 10th Annual ACM Symposium on the Principles of Programming Languages, Austin, Texas, 1983, pp.141-154.

[31] Milne, R. and C. Strachey: *A Theory of Programming Language Semantics*, John Wiley and Sons, 1976.

[32] Moszkowski, B. and Z. Manna: "Reasoning in Interval Temporal Logic", in LNCS, Vol.164, Springer-Verlag, 1983, pp.360-370.

[33] Nyberg, K.A., A.A. Hook & J.F. Kramer: "The Status of Verification Technology for the Ada Language", IDA Paper, P-1859, July, 1985.

[34] Peterson, J.L.: "Petri Nets", ACM Computing Survey, Vol.9, 1977, pp.223-252.

[35] Peterson, J.L.: *Petri Net Theory and the Modeling of Systems* , Prentice-Hall, 1981.

[36] Plotkin, G.D.: "A Power Domain Construction", SIAM J. on Comp., Vol.5, 1976, pp.452-487.

[37] Pnueli, A.: "The Temporal Semantics of Concurrent Programs", Theoretical Computer Science, Vol.13, 1981, pp.45-60.

[38] Pnueli, A. and W.P. DeRoever: "Rendezvous with Ada - A Proof Theoretical View", Proc. of the AdaTEC Conference, Crystal City, 1982, pp.129-136.

[39] Reisig, W.: *Petri Nets - An Introduction*, Springer-Verlag, 1982.

[40] Sammet, J.E.: "Why Ada Is Not Just Another Programming Language", CACM, Vol.29, No.8, Aug. 1986, pp.722-732.

[41] Scott, D.S.: "Domains for Denotational Semantics", LNCS, Vol.140, Springer-Verlag, 1982, pp.577-613.

[42] Stoy, J.E.: *Denotational Semantics: the Scott - Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[43] Zemanek, H.: "Formal Definition: The Hard Way", in *Formal Models in Programming* (eds. E.J. Neuhold and G. Chroust), IFIP, North-Holland, 1985.