A FULLY ABSTRACT AND COMPOSABLE INTER-TASK COMMUNICATION-CONSTRUCT†

by

Nissim Francez* and Shaula Yemini**

Technical Reports #299

November 1983

† Work done while the first author was a World Trade visiting scientist at IBM-Watson Research Center, Yorktown Heights, N.Y. USA.

* Computer Science Department, the Technion-IIT, Israel.

** IBM-T.J. Watson Research Center, Yorktown Heights, N.Y. USA.
ABSTRACT

In this paper we critically examine the suitability of Ada Multiasking constructs for expressing parallel algorithms, and suggest that some changes are required, the main of which is allowing entry calls to serve as guards in a select statement, similarly to the role accept plays. Thereby, entry calls can have other entry calls as alternatives, and can have also entry acceptances as alternatives. The arguments are two folded: 1. from the programming point of view, we present several situations in which the current design may force an unnatural programming style; 2. from a semantics point of view, we show that several semantic principles are violated by the current design. A new principle, called "wide horizons", is introduced. Finally, we suggest a modified general inter-task communication construct, which eliminates all the drawbacks discussed, while preserving the merits of the current design.

C.R. Categories and subject descriptors:
[D.1.3] Programming techniques: concurrent programming
[D.3.3] Language constructs: concurrent programming constructs
[D.4.1] Process management: concurrency, synchronization
[D.4.4] Communication management
[F.3.3] Studies in Program construct.

General terms: concurrent programming languages design

Additional keywords: Ada, rendezvous, semantics, guards.
1. INTRODUCTION

One of the more noted attributes of the Ada inter-task communication construct [Ada1] is its basic asymmetry between its two parts: calling (an entry) and accepting (an entry call). Actually, this asymmetry has two aspects, one of which is less discussed in the literature. In this paper, we discuss some of the consequences of the less often discussed asymmetry and suggest a modification of the inter-task communication construct which avoids the disadvantages mentioned in the sequel, while preserving the advantageous attributes of the rendezvous and making its semantics uniform.

The more frequently mentioned aspect of the asymmetry concerns the naming convention of the two parties (tasks) engaged in a rendezvous (see, e.g. [WJ], [WS], [G] for a more detailed discussion and comparison with related languages (CSP [H] and DP [BH]) and systems (Thoth [CMMS])). Basically, the calling task must explicitly name the identity of the callee task (whose entry is being called); the callee task, however, does not have (and even cannot) explicitly refer to identities of potential callers to its entries. The reason for this design decision is the natural programmability of service processes, residing in public libraries, which do not have to "know" the identities of their potential users. Thereby, no entry call can be "refused" based on the identity of a caller. However, this design decision raises several problems concerning access-control to which we relate in the next section. Another problem raised by this kind of asymmetry is the "return address" problem [RCEM]; as noted in the Ada Rationale [Ada2], it is not easily possible to program a task T "calling back" another task S, which previously called an entry of T. By using certain programming techniques, where arrays of tasks are defined, and each array
element task starts by accepting an entry call, in which an identity codes is passed to it by its parent task, some of the problems may be overcome. However, since tasks in Ada form a limited private type, for which it is forbidden to define equality, no compiler enforced security is possible (see [WL] for a discussion of secure access). It seems that a general purpose, dynamic, access-control mechanism would be preferable.

Here, however, we are interested in a different aspect of the asymmetry of the Ada rendezvous; namely, the ability to serve as a guard. The basic non-deterministic construct in Ada is the select construct, through which a task may simultaneously wait for several of its entries to be called, choosing among them according to availability (of callers). No such facility is provided for entry calls, which cannot serve as guards in a select construct. At most, an entry call can have an alternative local action and can be limited by a single time-out expression (several time-out expressions can be specified when waiting for entries to be called). Thus, whereas stand-alone occurrences of call and accept have some similarities and symmetry in their semantics, e.g. both can delay the issuing task (either for a limited period or indefinitely), this symmetry disappears once they are embedded in a select, rather than becoming the seed of a fully symmetric select. This yields a non-composable and non fully-abstract semantics. We shall return to the general structure of the select construct in Sections III and IV.

In this paper we argue, that the nondeterministic construct should be unified to allow calls as guards, thereby allowing a task to choose between several other tasks to call, depending on availability, and to choose between calling and accepting, also by availability.
Furthermore, we would like to state the following principle, to which we believe the design of languages for concurrent programming should adhere:

The "wide horizon" principle: whenever the semantics of a construct $C$ (in a language for concurrent programming) implies the delay of a process (task) containing $C$, $C$ should be able to have other alternatives [and all such constructs should be able to serve as alternatives to each other].

For example, the semaphore operation $p(s)$ is capable of delaying indefinitely without alternatives, as do monitor calls or conditional critical sections, all of which violate the "wide horizon" principle. We shall not pursue here alternative semantics for these constructs, and rather concentrate on Ada.

Note, however, that this principle does not force every application of such a primitive to have alternatives, since selections with a single alternative are always still possible. The idea is, that alternatives should not be necessarily excluded.

The above mentioned asymmetries have been contrasted with the full symmetry of CSP (e.g. [WL], [WS]) where the send and receive both explicitly name their partners and both may serve as guards. However, the implications of this asymmetry and alternative proposals were not discussed. The restricted naming convention can be somewhat relaxed, as shown in [F], still keeping it symmetric.

The rest of the paper is structured as follows: in Section II we discuss several programming situations, all of which motivate the inclusion of call as guard. In Section III we argue for the same from a more abstract point of view, showing that the current rendezvous construct does not have a fully-abstract and composable semantics.
In Section IV we suggest a unified select statement and show its application to programming natural solutions to the problems mentioned in Section II. It concludes with a discussion of several related issues, and of the question of implementation.

II. ENTRY CALLS AS GUARDS: A PROGRAMMING ORIENTED ARGUMENT

In this section we argue, that for the natural expressibility of certain programming problems, alternative call's, serving as guards, and of call's and accept's as guards, are required.

A. Conjunctive Transmission (broadcast)

Suppose a task T wishes to transfer the same item of information x to several other tasks, say A, B and C. However, the order in which the tasks A, B and C might be ready to receive the item x is unknown to T (possibly not even being predeterminable). Since task T is the generator of the information item x, and "knows" when it is available, and since tasks A, B and C control the time in which x is needed to them, it would be natural for T to initiate the transfer of x in an active way, while tasks A, B and C should be passively ready to accept x when ready for it.

Thus, one would expect that in a corresponding program the structure would be such that the tasks A, B and C would each contain an entry (with an in formal parameter), and will reach an accept statement for that entry when x is needed. In counterpart, task T will call the corresponding entries when x is available.

Unfortunately, since entry calls cannot serve as guards, the only possibility is that T commits itself in advance to some fixed
order of transmission, and attempt to call A, B and C's entries in that order. This may cause high delays and slow the overall performance in case this order is not consistent with the actual order of readiness (to accept x) of tasks A, B and C.

Furthermore, a deadlock may be caused in case the readiness of A, B and C to accept x is not independent of their internal communication. For instance, assume that T's ordering was A + B + C, while A is ready to accept x from T only after it has communicated with B, whereas B is waiting to communicate with T and get x, and only afterwards will B pay attention to A.

As a consequence, one is led to consider a different program structure, which we call inverted communication, where the roles of callers and callees are inverted. According to this inverted structure, task T will play a passive role by having an entry (with a formal output parameter), whose body assigns x to the parameter. Upon generating x, T will enter a loop of accepting three times a call to that entry. Each of the tasks A, B and C, on their part, will attempt to call T's entry when they need x, thus playing the active role in the rendezvous. One could say that a "taking" regime is used, rather than the traditional send-receive regime.

This solution suffers from the following drawbacks:
1. It may be unnatural at occasion, by forcing inverted communication on a task representing some purely passive entity, e.g. a peripheral device
2. It is not robust to access control. There is no simple way that T can ensure that exactly one entry call will be accepted from each of the tasks A, B and C, and no call is accepted from any other task. A much higher overhead would be needed to enable a "defensive programming" style, trying to enforce (by T) the required
communication pattern. As mentioned in the introduction, there is no straightforward method to achieve such a style.

As a result, some important synchronization patterns are not easily enforceable in Ada. A different solution is still possible, if one is ready to "double the Universe", and create auxiliary tasks A', B' and C', which we will call tasks 'A', 'B' and 'C', respectively, and then they will interact with T. This addition of extra tasks is a shift of the problem for the price of complicating the task structure of the program.

Yet, another, rather unnatural solution, not involving inverted communication but involving busy-waiting, might be designed along the lines mentioned in [WS], where T's calls to A, B and C's entries are embedded in a loop and each have a null local alternative, taken if the partner is not available. Thus, the number of times the loop is "vacuously" traversed is not bounded.

The whole difficulty can be eliminated by allowing entry calls to serve as guards and have alternatives. Thus, tasks A, B and C could have their entries, while T would enter a loop with a non-deterministic choice of calling (once only!) each of these entries. A corresponding program, using our suggestion for a unified rendezvous, is displayed in Section IV.

To summarize, by means of guards consisting of entry calls, one achieves:

1. some form of access control: send only to intended recipients the exact number of items required. The above supports a limited form of access control. A general mechanism of supporting access control in Ada requires a more radical change in the scope rules for entries, making them dynamic instead of static, and is beyond the scope of this paper.
2. flexible synchronization control: send according to recipient's availability, avoiding unnecessary delay.

Both these features are lost when using inverted communication in Ada.

B. Disjunctive Transmission

Suppose, again, that a task T attempts to transmit an information item x, this time to any of a number of other tasks, say again A, B and C. Again, task T has no information as to the relative readiness of tasks A, B and C to receive x. For example, task T may be an intermediate station in delivering a message x in a network, attempting to send x through any of the possible "next hop" tasks A, B and C to a remote destination, and not caring about which will actually do the job, yet wishing the first available one to do it.

In the disjunctive transmission case, the absence of guarding entry calls is even more problematic than in the case of the conjunctive transmission: in case one of the tasks A, B and C is never ready to accept x, the conjunctive transmission does not have to terminate, while the disjunctive one has to! Thus, a solution in which task T commits itself to some fixed order of probing, localizing its choice, is certainly incorrect, since it may cause T to wait forever when unnecessary. Ada's conditional call, which provides no alternative targets of call, is of no help either if busy waiting is to be avoided.

Considering an inverted communication solution, as presented in the previous example, again, similar drawbacks appear.

1. It may be unnatural solution. One could hardly consider it a good programming style to let network processes "chase" other processes for message deliveries to remote stations. It also disallows the commonly used "routing tables" to reside in their natural residence, at the sender side.
2. Because of the lack access control, another task may "take" \( x \) before A, B or C manage to. Also, A, B and C's calls on T's entry must be made conditional; otherwise, upon the successful completion of a rendezvous by one of them, the other two remain "stuck" with their calls. This is hard to achieve, unless a bound is known on the minimal delay before the first rendezvous succeeds, which usually is not the case.

Again, a natural solution is programmable using the modified communication construct proposed in Section IV, where T has three alternative entry calls on entries in A, B and C respectively, choosing among them based on availability.

C. Pipelining

Assume tasks \( T(1), \ldots, T(n) \) are to form a pipeline, where \( T(i) \) receives items from \( T(i-1) \) and sends them to \( T(i+1) \), \( 1 < i < n \). The task \( T(1) \) is the producer of items, while task \( T(n) \) is the ultimate consumer. Also, some buffering capability is required localized within the given tasks. In a natural solution, as long as the local buffer is not in a boundary state (either empty or full), each \( T(i) \) is simultaneously ready either to accept an item from \( T(i-1) \) or to send an item to \( T(i+1) \), again willing to choose by availability of its neighbors. One would also like each \( T(i) \) to have the same code, except for its dependence on \( i \).

By an analysis similar to that in the previous cases, one is led to the conclusion that no natural coding exists, unless one allows a mixture of call guards and accept guards, allowing each \( T(i) \) to pay attention to both neighbors, and having a local buffering capability. Thus, this mixture appears to be not less important than having calls as alternatives to other calls.
D. Transmission Stopped by Signal

Another situation where mixture of calls and accepts is beneficial is the following. Consider a task S (sender) repeatedly transmitting a value x to another task R (receiver), stopping upon the reception of another signal.

A possible occurrence of such a situation is in the context of disk i/o. Suppose a system consists of a family of disks, controlled by some control unit which is connected to a channel. The control unit includes a poller, whose job is to interrogate the subfamily of currently disconnected disks, and to sense whenever a head is in the neighborhood of a required sector. Such polling is performed rather frequently in order not to miss the short period during which the head is in the right neighborhood. On the other hand, if a signal is accepted from the channel notifying a new connected disk the polling is stopped and the new request is handled.

The inverted communication solution again displays the same problematics as before. A natural solution would involve S looping, wherein each loop traversal it will choose between an entry-call to R, thereby retransmitting x, and an accepting of an entry-call, thereby receiving the signal and terminating the loop.

Before passing to different arguments in the next section, we would like to present yet another programming argument which further generalizes our requirements for an inter-task communication construct and motivates our proposal in Section IV.

Indexed Families of Entries and Tasks

Two of the more convenient constructs of Ada are the indexed family of entries within a task and task arrays. The construct we propose in
Section IV supports taking advantage of these parameterizations for calling any one of, all of, or a subset of, identical entries of a task array, or calling any one of, all of, or a subset of, the entries in one family, or waiting for any one of, all of, or a subset of, entries in a family to be called (where subsets are specified by a compile time determinable range). In order to take full advantage of these constructs, again we would like to have the following capabilities:

1. Call (exactly once) in an unspecified order (depending on availability) all entries in an indexed family,
2. Call an arbitrary entry in an indexed family, depending on availability.
3. Specify a subfamily (possibly compile time determinable), and apply 1. and 2. above to that subfamily.

These capabilities would again call for calls as guards. As an example, consider a task V representing a vending machine simultaneously serving several users. This task may have a family of entries corresponding to various currency dominations, say 1, 5 and 10. A user task should be able to choose between calling a 10-entry, or calling a 5-entry twice, depending on availability.

Unfortunately, indexed families of tasks do not exist in Ada. Still, one can define arrays of tasks, and one would like to have a convenient means of choosing to call in a parameterized way entries belonging to members of the array determined by the parameters. As shown in [Y], [B], no parallel entry calls are possible. The least one would require is not to commit oneself to a predetermined ordering of calls.

In conclusion, in this section several programming situations
were presented, where it would be more natural to use a rendezvous construct which allows both entry calls and entry acceptance as guards, to form alternatives to each other, thereby adhering to the "wide horizon" principle stated in the introduction.

III. ENTRY CALLS AS GUARDS AND UNIFORM SELECTS: A SEMANTICS ORIENTED ARGUMENT

In this section, we argue for the need of a more uniform select construct, and in particular, one including entry calls as guards. The argument is based on general semantic principles, that we believe should guide the design of programming languages in general, and ones containing concurrency, communications and non-determinism in particular. These general arguments reenhance the practical, ad hoc, arguments presented in the previous section.

Unfortunately, until today no formal denotational semantics exists for the full concurrency in Ada, though one exists for the sequential part of the language [DG]. Thus, we adapt the general principles in question, which are usually applied to mathematically-defined abstract semantics, to the more informal one as derived intuitively from the most recent version of the reference manual [Adal].

A. Closure under Concurrent Composition (binding)

Whenever a programming language admits constructs that allow some kind of a composition, it is most favorable to design such a composition so that it becomes semantically transparent. In other words, it should be the case that the composed construct behaves as a "black box", and thus does not reveal to external observers its internal composition.
This feature is especially important when the composition operator is completely orthogonal to its arguments, as is concurrency vs. sequenciality. Thus, one would like the following principle to hold:

**Closure under binding:** for every concurrent (multi-tasked) program, there exist a sequential (unitask), possibly non-deterministic, program with an equivalent externally observable semantics.

Thus, multi-tasking should be transparent to an external observer, who should not be allowed to deduce that a module (e.g. a package) is composed of concurrently executing tasks. Observes are assumed to be sequential themselves and, hence, incapable of observing simultaneously simultaneous entry calls!

This principle is violated by the current design of the tasking constructs in Ada. Consider the following simple program fragment:

For brevity, we do not separate task specifications from their bodies.

```ada
declare
task t1 is s1.e1() end t1;
task t2 is s2.e2() end t2;
begin
dec)
```

The externally observable behavior of this program section consists of calling two entries in two different tasks in any order, not depending even on the availability of tasks \( s_1 \) and \( s_2 \). No unitask Ada program section can display an equivalent behavior, unless entry calls are allowed as guards, which would create the required effect.

We note that hiding concurrent outputs was a major consideration for the introduction of output guards in CSP, whose design had the above mentioned closure property as a major design goal (Hoare - private communication).
By pursuing this argument further, one can reach the conclusion that the language should provide also local nondeterminism, by which sequential programs can nondeterministically choose among different (guarded) continuations, independently of any intertask communication. Such programs are easily generated as a result of hiding multitask programs. Consider the following simple example (again for brevity we do not separate task specifications from their bodies).

```plaintext
declare task t1 is s.e(x,z) end t1;  // x,y,z are global variables,

task t2 is s.e(y,z) end t2;

task s is entry e(v: in integer; u: out integer),
    select accept e(v: in integer; u: out integer)
        do u:=v end e;
    end-select;

begin end
```

Clearly, the effect of this section, seen as a black box, is a nondeterministic choice between z:=x and z:=y. Thus, the original guarded commands [D], which had nothing to do with communication, are back!

This view of local nondeterminism is contrary to others, e.g. [G], and recently [R], which would like to attempt to restrict nondeterminism to communications, abstracting away from processors speed differences. Obviously, such a point of view is in conflict with the closure principle stated above.

While passing we would like to make the following observation: local nondetermining is already present, though in a disguised form, in the current design of a select construct. Consider two delay alternatives with an identical time out. The choice among them when the
specified time out has elapsed is an instance of a local nondeter-
ministic choice. We prefer an explicit construct for its conveyance.

B. Full Abstractness

A fully abstract semantics [HP] is an important property for any
language construct, enabling it to have a clear meaning, independent
of any context in which it may be imbedded. We claim that in its
current design, the Ada's intertask communication construct does not
provide for a fully-abstract semantics of its components; the unified
construct presented in the next section provides for it, and is
therefore preferable.

Consider, for example, a single accept for a given entry. In a
context where it is in a stand-alone position, sequentially composed
with some other actions, it has the meaning of an indefinite delay
until the corresponding entry is successfully called by some other
task and a rendezvous engagement takes place. However, when an accept
appears as a component of a select construct with an else clause (which
may appear pages apart...) it has the meaning of a conditional readi-
ness to accept an entry call, provided one is immediately possible,
implying no delay at all! Another possibility arises if a delay clause
exists as an alternative in a select, and another meaning that of timed
delay, is assigned to the same construct. Finally, yet another inter-
pretation is implied in case a terminate alternative is present.
Thus, one cannot understand the meaning of accept in a context-
independent way.

The entry call has a similar problematics, having the possibility
of becoming conditional or timed depending on context.
We would like to have an alternative approach, in which the
entry calls and entry acceptances have a unique, context independent,
fully abstract semantics. This should be compared with ordinary

\[
\text{if } \ldots \text{ then } \ldots \text{ end if} \quad \text{vs.} \quad \text{if } \ldots \text{ then } \ldots \text{ else } \ldots \text{ end if},
\]

where the existence of an else clause does not affect the meaning
of the then clause.

C! Compositionality

As a consequence of what has been said in B. above, one gets
several kinds of alternative-selection constructs. This, by itself,
is of no harm. The trouble is, that one cannot determine with which
kind of selection construct one is dealing, until the whole body of
the selection construct is scanned, thereby also violating the principle
of information hiding. For example, in order to determine whether or
not the selection may time-out, delay alternatives must be looked for.
We prefer this information to be concentrated in just one place, in
the heading of the construct, next to the key word select itself,
immediately hinting at which construct is at hand. An indication of
a zero-delay would be given in the heading, instead of turning pages
looking for a "hidden" else clause. For a general discussion of the
role of compositionality of the semantics of programming constructs,
see [JvEB].

In the next section, we present our suggestions for a unified
rendezvous, not suffering from all the problematics mentioned.
IV. A UNIFIED RENDEZVOUS CONSTRUCT

In this section we introduce a revised intertask communication construct, which we claim overcomes the difficulties and problematics stated in the two preceding sections, while preserving the merits of the original design. It has a more uniform character and adheres to the general semantic design principles mentioned.

The properties of the revised rendezvous are as follows:

1. A rendezvous occurs between two matching select statements.

Thus, we omit the stand-alone occurrences of entry calls and entry acceptances; they are both transformed into single-alternative select statements, and the (single) alternative is trivially open. Of course, one may still want to write down stand-alone constructs, but this will be considered "syntactic-sugar." A similar transformation has been applied in [EF] to standalone I/O commands in CSP. We defer the definition of matching until after the select-body is discussed.

2. Each select statement has a header, a body and two escape clauses.

The header of a select specifies a time limit, while the body specifies the alternatives among which the selection should take place. One escape clause describes the action to be taken in case there are no open alternatives, while the other describes the actions to be taken in case the time limit specified in the header is exceeded.

3. The time limit of a select can be specified as some simple-expression or as a special value '*'.

The value of the expression is an upperbound on the amount of time the select is ready to wait for a rendezvous. The special value '*' means unbounded time. A zero value requires an immediate rendezvous or a local alternative.
We shall modify somewhat the current Ada semantics regarding the "distributed termination convention" (DTC) related to the 'terminate' construct. In the current design, a 'terminate' is optional whenever no time limit is specified. We regard the 'terminate' as (implicitly) being present whenever no time-limit is specified, i.e. '*' is used. The effect of this modification is that fewer programs will deadlock, terminating instead. In the current design, a select without time limit, which did not specify a terminate option, will wait for ever if all its "brother tasks" have either terminated, or are in a similar situation. We chose to enforce termination in such a case, freeing the rest of the program that otherwise would be hang up. The task-dependencies rule for the DTC remain intact.

4. The body of a select consists of a number of alternatives, among which a choice occurs.

5. An alternative consists of a guarding part and a guarded part.

The guarded part is any statement. The guarding part itself has two components: a boolean condition and a communication component, which is either an accept to one of the task's own entries or a call to another task's entry or null. An alternative is open if its boolean part was true at the time it was evaluated (to be discussed). Special care must be taken when using a boolean part referring to explicitly shared variables, which might be changed by another task after the guard was evaluated, and with implicitly shared variables (attributes) who might be automatically changed after guard evaluation.

One should consider the implications of different select management strategies for boundary cases such as:
- what alternative is chosen when there are both local (empty communication component) alternatives and communication alternatives (non-empty communication component) available within the limit time; and

- what is done when there are open local alternatives, but the time limit has not yet expired - should there be a wait for a communication alternative to become available, or should a local alternative be selected as soon as established.

We feel that these issues are independent of the current suggestions, and any way of resolving them is consistent with the indicated design goals.

6. Two select statements in two tasks, t1 and t2, match if one of them contains an entry call to an entry belonging to the other one, while the other one contains an accept to that entry and both the call and the accept appear in open alternatives.

7. Following is the evaluation scenario involved in determining whether a select statement can be engaged in a rendezvous.

a. determine waiting timeout from the header.

b. evaluate (in any order) the set of all open alternatives.

At the end of this step, start the delay countdown.

c. Find a matching select in another task.

This is the step with the major deviation from the current evaluation scenario, which is local to the task containing the select, while here a more global scenario takes place.

d. After two matching selects having been found, evaluate the actual parameters of the entry call and proceed into the rendezvous, and the rest of the guarded part following it (exactly as in the current construct).
- If no matching select was found within the time limit, complete
the select by performing the time-out escape clause. We feel
that the actual design of a language construct for structured
handling of time-outs is an important separate issue that is
beyond the scope of the current discussion.

We shall not bother here to give an exact BNF syntax description,
and we assume one can be readily obtained. Rather, we do code the
examples of Section II using the revised construct and some "free"
syntax.

Yet, we would like to make the following remarks about the syntax
of alternatives, which applies to its current form, in the Ada select.
We definitely believe that the syntactic location of the communication
part of an alternative (when not null) is before the separator of the
guarding and guarded parts (the "arrow"). This separator should have
the meaning of an irreversible barrier, once passed. We consider the
current syntax, where the accept of an alternative follows the arrow,
to be a source of semantic confusion. We assume this to be corrected
in any concrete syntax for the currently suggested construct.

Also, we assume the syntax provides some syntactic means to express
parameterized guard selection, e.g. or (i:1...h)b_i: c_i -> S_i, where the
guards may depend on the bound variable i. We shall not enter here into
a discussion of permissible ranges and of the question of their time
of determination. Obviously, compile time determined ranges are
easier to implement.

Following are program sections presenting the usage of the revised
intertask communication mechanism for expressing the programming situa-
tions mentioned in Section II. As noted, we deliberately deviate
from the strict Ada syntax, and use a rather free one.
A. **Conjunctive Transmission**

Here we take \( x \) to be some global variable, to avoid its explicit computation.

```
declare task T is
begin
  begin send_A, send_B, send_C: BOOLEAN:=TRUE;
    while send_A or send_B or send_C loop
      select (*)
      when send_A; A.receive(x) \rightarrow send_A:=FALSE
      or
      when send_B; B.receive(x) \rightarrow send_B:=FALSE
      or
      when send_C; C.receive(x) \rightarrow send_C:=FALSE
    end-select
  end_loop
end;

end;

task A is
  entry receive(y: in type_of_x);
  u: type_of_x;
begin
  select (*)
  accept receive(y: in type_of_x) \rightarrow do u:=y;
  end_receive {end rendezvous "use u as x"...}
  end_select
end_A

task B is {similar to task A};
task C is {similar to task A};
begin end.
```
B. Disjunctive Transmission

This time, no loop is needed.

```c
    task T is
    begin
        select (*)
            when A.receive(x) => skip,
            or
            when B.receive(x) => skip,
            or
            when C.receive(x) => skip
        end-select;
    end T;
```

As for tasks A, B, C, at least one of them should contain a `select` accepting a receive-entry to eventually match T's call.

C. Pipelining

For simplicity, we assume LIFO pipelining.

```c
    task T(i) is
        b: array of item_type; size: 0...N:=0
        entry buf(x: item_type);
        select (*)
            when size < N; accept buf(x: item_type) => do size:=size+1;
            b(size):=end_buf
            or
            when size x>0; T(size+1).buf(b(size)) => do size:=size-1
        end-select
```

Using call T(size+1).buf allows to ensure that the available item be delivered to the required task and to no other task.

Finally, we would like to address the issue of implementation of the revised Ada intertask communication primitives suggested. Obviously, the implementation becomes more expensive in general since the resolving of nondeterministic choices cannot be done any more within
a single task; rather, "handshaking" algorithms have to be employed in order to find matching pairs of selects. Several distributed handshaking algorithms are known in the literature (see [BS] for a comparative discussion and further references) of which the probabilistic one [FR] is highly efficient. Thus, this is another case of a tradeoff between an order of magnitude complication of the implementation versus benefits of expressibility, some access control; and, most importantly, a clean and uniform language construct.

Furthermore, some rather simple compiler-optimizations are possible which will identify selects which do not use the extra power introduced here, and for which the current, more efficient, implementation techniques are valid. Thus, no extra express are necessarily imposed on whoever is satisfied with the current design of Ada's inter-task communication.

ACKNOWLEDGEMENT

We would like to thank A. Sielbeshatz, R.E. Strom and S. Cohen for useful discussions and critical reading of the paper.
REFERENCES


REFERENCES (cont'd)


