Technical Report on Phase I Research
Report # SCA-149

Ada-Linda Preliminary Report: Motivation,
Informal Description and Examples
Contract # N00014-89-C-0268

Research Supported by
Strategic Defense Initiative/Innovation
Science and Technology

Managed By
The Office of Naval Research *

246 Church Street
Suite 307
New Haven, CT 06510
(203) 777-7442

* The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as
an official Department of the Navy position, policy, or decision, unless so designated by other official documentation.
Contents

1 Introduction 2
2 Brief description of Ada-Linda 3
3 Linda World Examples. 8
4 Ada World Example. 15
5 Multiple tuple spaces 21
6 File systems 21
1 Introduction

SDIO 3405 — "Strategic Defense System Software Policy" — is typical of many Defense Department policy statements in decreeing that all "mission-critical software" be implemented in Ada [SDIO 3405, p.1] The intentions behind the decree are honorable:

A standard programming language and standard notations for specifications and designs are highly desirable to provide a common basis for understanding and to permit the development and wide usage of common tools for maintenance, evaluation and testing [ibid. p.3].

The software developer (duly inspired) wades into the body of the policy statement — only to discover a long section on "waiver procedures". Grounds on which the required use of Ada may be waived include "(1) performance, or (2) appropriateness of the Ada programming model" [p. 10]. Waivers in the second category may be based on the inappropriateness of Ada interprocess communication, initialization or task scheduling, among other things.

If the developer's plans include distributed systems or parallel applications, his enthusiasm may now be waning. Interprocess communication and task scheduling aren't minor issues. They underlie every attempt at parallel or distributed programming. In acknowledging Ada's deficiencies in these areas, SDIO 3405 implicitly admits that, so far as parallel and distributed programs are concerned, Standard Ada will not, indeed cannot meet its stated goal of serving as a "standard programming language."

It might be argued that (to the contrary) no programming language can do everything right, and that Ada represents a reasonable attempt at covering the bases, even if it falls short in some areas. In this report, we attempt to demonstrate that this optimistic view fails to hold water. The waiver conditions in SDIO 3405 don't overstate the case. In fact, Standard Ada is poorly suited to distributed systems and unacceptable for parallel applications.
1. Synchronization in Standard Ada is inappropriate and too expensive, 
2. naming is too inflexible, and 
3. there is no support for safely-sharable data structures. 

Furthermore, Standard Ada does not generalize to provide the support that will be wanted soon (even if it isn't now) for fine-grained asynchronous parallelism and for object-structured file systems. And so far as distributed and parallel programming in Standard Ada is concerned, insufficient power and flexibility are complemented by excessive complexity. The complexity of the tasking and interprocess communication constructs in Standard Ada is a burden to program-writers, program-readers, implementors and tools developers. 

In hindsight, Standard Ada's inadequacies in these areas aren't surprising. They are a natural consequence of the crucial but frequently overlooked fact that the design of Ada preceded the decade in which parallel and distributed programming were first widely practiced. 

In the absence of any substantial experience with parallel and distributed programming, the Ada designers made the best guesses they could. It's no discredit to them to acknowledge that, in retrospect, they often guessed wrong. But it will be to our discredit if we don't correct the mistake, before massive investment in parallel and distributed Ada software yields complex and inefficient codes at high cost — to the extent (see "waivers") that these supposed Ada applications are actually written in Ada at all. 

This report gives a brief and informal description of Ada Linda. It then discusses some "Linda world examples" (parallel applications) and some "Ada world examples" (distributed system routines, of the sort the Ada designers themselves used to illustrate and explain their constructs). 

2 Brief description of Ada-Linda 

Ada-Linda (otherwise known as Ada Lite) is intended for implementation as a precompiler that generates standard Ada code, to be linked to the Ada Linda runtime library. 

The following outline assumes familiarity with standard Ada and with Linda. 

Synopsis: Ada-Linda compared with C-Linda. The Linda components of these languages are basically the same, and processes in the the two languages should be able to communicate via a common tuple space. But there
are some differences in detail. Because C has a weak type system, C-Linda makes relatively weak use of typing; Ada-Linda depends more extensively on type information.

In C-Linda, objects added to tuple space are created dynamically via out or eval statements. A C-Linda tuple as a whole has no explicit type. In Ada-Linda, any single object of any type may be added to tuple space via out; any object may be used as a template for object retrieval via In or rd. The retrieved object will be identical in type to the template object. In Ada Linda, The role of C-Linda tuples (i.e. of heterogeneous aggregates) is played by Ada records (which happen to be heterogeneous aggregates anyway). Where C-Linda adds a 3-tuple to tuple space (TS), Ada-Linda adds a 3-field record.

Formals in C-Linda are replaced by empty fields in Ada-Linda. Where C-Linda uses a formal in a tuple or template, Ada-Linda uses a field that is bound (via ordinary assignment statement) to the distinguished value empty. (Every type in Ada-Linda implicitly includes the empty value.) empty fields in templates behave in the same way as formals in C-Linda templates: after matching, they are no longer empty; they are bound to the corresponding values in the retrieved object.

C-Linda's eval is replaced by Ada-Linda's ticking prefix. Evaluation of a ticking expression yields a task descriptor which detonates (or in other words, starts to evaluate) when it is dumped into tuple space. Thus, both passive and live tuples are generated via out.

Tuple space operations in Ada-Linda. Ada Linda provides the basic operations out, In and rd for operating on tuple spaces, the compound operations outin and Inout, and the new keywords empty and ticking.

1. out adds a copy of any Ada object to tuple space.

Examples:

Given the declarations

type RType is
  record
    name: string;
    index: integer;
    value: ValType;
  end record;

R1, R2: RType;

The statements

R1.name := "annette";
R1.index := 25;
R1.value := V;
out R1;

adds a three-field record to tuple space; this operation is in essence identical to the C-Linda operation

out("annette", 25, V),

except that in Ada-Linda, all objects added to tuple space must be typed, and can be withdrawn only by templates of the same type.

The same effect can be achieved by the statement

out R1("annette", 25, V);

which initializes R1 to the aggregate value ("annette", 25, V) before out'ing it, or by

out R1(name => annette, index => 25, value => V).

(When a tuple space operation cites an aggregate-valued variable A, inclusion of the form

fieldname => fieldvalue

within parentheses after the variable's name is equivalent to executing the statement

A.fieldname := fieldvalue

before the tuple space operation.)

Finally, the same thing can also be accomplished by

out RType("annette", 25, V),

or by

out RType(name => annette, index => 25, value => V).

In these two cases, a new object of type RType is generated dynamically and added to tuple space (on analogy with allocation via the new statement).

The statement
out RType("annette", empty, V)

adds an object whose middle field is empty. (The role of empty fields in matching is discussed below.)

2. in withdraws an object from tuple space. (1) It may use any Ada object as a template. We refer to this object as the template-object and the object withdrawn from TS as the TS-object. (2) The TS-object will have the same type as the template-object. (3) Any or all fields of a template object may have values; any or all may be empty. The same holds for a TS-object. A value in the template-object must be matched either by the same value or by an empty field in the TS-object. A value in the TS-object must be matched either by the same value or by an empty field in the template-object. In other words,

<table>
<thead>
<tr>
<th>template:</th>
<th>TS-object:</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>value</td>
<td>empty</td>
</tr>
<tr>
<td>empty</td>
<td>value</td>
</tr>
<tr>
<td>empty</td>
<td>empty</td>
</tr>
</tbody>
</table>

(4) After the in completes, all empty fields in the template-object have been filled in with the corresponding value from the TS-object. (5) If there are no matching objects in TS, In blocks until one shows up. If there are many, it makes a non-deterministic choice.

Examples. Given

type RType is
record
  name: string;
  index: integer;
  value: ValType;
end record;

R1, R2: RType;

out R1(name => annette, index => 25, value => V);

the TS-object generated by out may be withdrawn by

in R2("annette", empty, empty);

\footnote{When we say that a field has a value, we mean that it is not empty. For present purposes, we don't consider empty to be a value.}
or equivalently by

\[ \text{in R2(name => "annette", index => empty, value => empty)} \]

or by

\[
\begin{align*}
&\text{R2.name := "annette";} \\
&\text{R2.index := empty;} \\
&\text{R2.value := empty;} \\
&\text{in R2;}
\end{align*}
\]

After any one of these In's,

\[ \text{R2.index = 25} \]

and

\[ \text{R2.value = V.} \]

The same TS-object might be withdrawn by

\[ \text{in R2(name => empty, index => empty, value => V);} \]

by

\[ \text{in R2(name => empty, index => empty, value => empty);} \]

and so on.

3. rd words in the same way as In, but without withdrawing the matched
TS-object from tuple space.

4. The keyword ticking is used to prefix an expression. Bindings for all
names in a ticking expression are established when the ticking statement is
evaluated; further evaluation of the expression is postponed until the expression
is added to tuple space, directly or as a top-level field of an aggregate object.

\textit{Example.} The statement

\[
\begin{align*}
&\text{type Worker is..;} \\
&\text{function WorkerTask(\ldots) \ldots;} \\
&\text{out Worker(ticking WorkerTask(i))};
\end{align*}
\]

is in essence equivalent to C-Linda
eval(WorkerTask(i));

It creates a live object in tuple space — i.e., a new process (a new evaluation thread or locus of execution), which evaluates the function call WorkerTask(i) and then turns into a single-field (passive) TS-object containing the result yielded by WorkerTask(i).

Assuming the definitions above,

out R1(name => annette, index => ticking 3**2 + 4**2, value => V);

generates a live object whose middle field evaluates the given expression.

5. Ada Lite is a superset of standard Ada, but these Linda operations supercede standard Ada's tasks, accept statements and entry calls, select statements and guards. Ada Lite programs may use these Standard Ada constructs, but will never require them.

6. Topics not discussed here: multiple tuple spaces and time-bounded In and rd statements are part of Ada Lite. They will be described in the next (more formal and complete) version of this specification.

3 Linda World Examples.

Problem: a maximally-simple master-worker program. A master process creates $n$ workers. It dumps task (or "job") assignments into tuple space; workers repeatedly withdraw a job, do it, and dump the result back into tuple space. When all results have been reported, the master adds an "All Done" task to tuple space, and the workers terminate. The Lite version is in figure 1, Standard Ada in figure 2.

(a) Behavior of the Standard Ada version. The main task creates worker tasks by using the allocator. (Dynamically allocating a new task object causes it to be activated.) The master then enters a select loop within which it accepts new-job requests or results from the workers. When all jobs have been processed, it enters a final loop in which it supplies each worker with an "AllDone" job, causing it to terminate.

(b) What's wrong with the Ada version vs. the Ada Lite Version?

First, consider job assignment. In the Lite version, workers that need jobs withdraw them directly from TS; they add results directly to TS. The Master can run ahead of the workers: it can generate job descriptions, and add them to TS, as soon as this is (logically) possible. When a worker needs a job and TS holds at least one job descriptor, the worker blocks only during the time
procedure MasterWorker(NumWorkers: integer) is

declare

    type Worker is integer;
    type JobType is [ ... ]; -- describes a single sub-task
    type ResultType is [ ... ]; -- describes a result
    i, JobCount: integer := 0;
    JobDescription: JobType;

function WorkerTask return integer is
declare
    Job: JobType;
    Result: ResultType;
begin
    loop
        in Job(Empty); -- withdraw a job
        exit when Job = AllDone;
        Result := WorkOn(Job); -- do it
        out Result; -- dump result
    end loop;
    out Job(AllDone); -- replace "all done" token;
    return 1;
end WorkerTask;

begin

    for i := 1 to NumWorkers loop -- create workers
        out Worker(ticking WorkerTask());
    end loop;

    while (more jobs) loop -- dump job descriptors
        JobDescription := Next job;
        out Job(JobDescription);
        JobsOut := JobsOut + 1;
    end loop;

    while (JobsOut > 0) loop -- collect results
        in Result(Empty); -- collect answers;
        Do something with the Result;
        JobsOut := JobsOut - 1;
    end loop;

    out Job(AllDone);
end;

Figure 1: Ada Lite version
task body MasterWorkers is

task type WorkerTask is

loop
    Master.GetJob(Job);
    exit when Job = AllDone;
    Result := WorkOn(Job);
    Master.Report(Result);
end loop;
end WorkerTask;

task body Master is

declare
    worker: access WorkerTask;
begin
    for i := 1 to NumWorkers loop -- create workers
        worker := new WorkerTask;
    end loop;

    while (more jobs) OR (JobsOut > 0) loop
        select
            when (more jobs) =>
                accept GetJob(J: JobType) do
                    J := Next job;
                end GetJob;
                JobsOut := JobsOut + 1;
            or
                accept Report(R: Result);
                JobsOut := JobsOut - 1;
                do Something with the Result;
        end select;
    end loop;

    for i := 1 to NumWorkers loop
        accept GetJob(J: JobType) do
            J := AllDone
        end GetJob;
    end loop;

end MasterWorkers;

Figure 2: Standard A+ version
necessary to withdraw the descriptor from TS. The master can generate new job assignments or collect results while workers are withdrawing job assignments.

In the Standard Ada version, workers must rendezvous with the master in order to get job assignments. When a worker needs a job, it must block until the Master is scheduled, accepts its request and generates a job descriptor (or withdraws one from a local holding pen). Thus, even when a job-seeking worker is the only process to be seeking work at a given time, a job-assignment is a more complex operation in Standard Ada than in Lite. But suppose that n workers simultaneously require assignments. In Lite, all n reach into TS and withdraw assignments simultaneously; in Standard Ada, they form an n-long queue at the "GetJob" entry. The nth worker is blocked until the previous n - 1 have all gotten assignments. Because the Master task must respond personally to each job request, it is a potential bottleneck.

Objection: If n Lite workers attempt to withdraw objects from TS simultaneously, aren't they also forced to access TS serially, leading to the same kind of queueing delays as in Standard Ada?

Answer: Let's assume that the Lite implementation in fact stores all job descriptors in a single TS segment, with a single lock. If this is the case, the n Lite workers are indeed serialized upon access to TS. But: each worker in line need only acquire a lock, withdraw the head element of a queue of tuples, and release the lock. In Ada, each worker in line needs to execute an entire rendezvous with the master task before the next worker gets its turn. In Lite, tuples can be moved asynchronously out of the master's address space and into some space that is conveniently accessible to the workers (i.e., into TS — into shared memory or system buffers, depending on the host hardware). Job descriptors are "staged", set-up for fast and convenient grabbing by the workers — while the workers are busy (on some previous job). Ada doesn't allow this kind of staging. Job descriptors remain stuck in the master's address space until workers claim them explicitly.

Now, consider the returning of results. In Lite, workers add result objects directly to TS; they block only during the period that is required to add a new object to TS.

In Standard Ada, workers must again rendezvous with the master task in order to return results. Even if the master is otherwise idle when a result is to be returned, the worker must block (again) until the master is scheduled and has accepted its result. But suppose that n workers attempt to return results simultaneously: again, the consequence is an n-long queue of workers awaiting rendezvous. Note that, while workers blocked awaiting rendezvous with the "GetJob" entry at least have a quasi-legitimate reason for waiting (this may
be a costly way to get a task assignment, but they can't proceed until they've
gotten one somehow or other), workers who are blocked awaiting rendezvous
with the "Report" entry are blocked for no logical reason whatsoever. They do
not need a response from the master.

In our programming experience, this situation — processes with data objects
to disseminate, with no logical need to await a response or a result — is ubiq-
uitous. Forcing such processes to block while their data objects are "accepted"
by some other process is perverse. Not only is it costly, it's conceptually inept.
It hides the real nature of the transaction behind the false front of a remote
procedure call. An expressive language allows the programmer to code simple
operations simply, without circumlocutions.

Finally, consider the interaction between job assignment and result report-
ing. In Lite, there is none: one worker can be withdrawing a job assignment
while another is adding a result to TS. In Standard Ada, a process with a result
to report can be forced to wait not only while other processes report results, but
while other processes are assigned jobs — results are accepted and jobs are as-
signed by a single master task within a single accept loop. Similarly, job-seeking
workers may have to wait while other workers report results.

Objection: couldn't Ada have used a series of two loops, the first
to assign jobs and the second to collect results, as the Lite version
does?

Answer: No. As soon as a worker attempted to return a result, it
would block until all jobs had been assigned and the first accept
loop had terminated. This scheme makes it impossible for workers
to return results until all jobs have been assigned; hence workers
must build and maintain local result buffers, and return them upon
completion. Sometimes this is acceptable. Other times it is pure
nuisance-overhead.

A more complex master-worker program

Problem: compute the elements of a matrix each of whose elements is defined in
terms of the elements directly above and to the left of it — i.e., we can compute
all elements along a counter-diagonal simultaneously as soon as we know the
previous counter-diagonal. Strategy: The matrix is divided into sub-blocks. A
job assignment consists of computing all sub-blocks in a given row-band. As
soon as the $i$th sub-block in a band is complete, its bottom row is communicated
to the process working on the band immediately below, so that it can proceed
to compute its own $i$th sub-block. When a worker is finished with the $j$th row
band, it proceeds to the $j + \text{NumWorkers}^\text{th}$ band.
How does each worker communicate its bottom-row segment to the next?

The solution is simple and immediate in Lite:

--- I'm working on row-band i, about to compute the jth column block;
--- I need a bottom row from the block above in order to proceed:
in BottomRow(row => i-1, column => j, Vector => empty);
... compute, referring to BottomRow.Vector(index) ...
... I put my own newly-computed bottom row in NewVec ...
--- Release my newly-computed bottom row segment to next worker:
out BottomRow(row => i, column => j, Vector => NewVec);
... now proceed to my j + 1st column

The equivalent solution — each worker communicates bottom-row information directly to the next worker — is in effect impossible in Ada. Each worker task would need to begin the computation of each block by executing an accept statement, by means of which the worker directly above would pass in a newly-computed bottom row. At the end of each block computation, it would call the corresponding entry in the worker directly beneath:

--- I'm working on row-band i, about to compute block j;
--- I need a bottom row from the block above in order to proceed:
accept BottomRow(Vector: VectorType);
... compute, referring to Vector(i) ...
... I put my own newly-computed bottom row in NewVec ...
--- Release my newly-computed bottom row segment to next worker:
NextWorker.BottomRow(NewVec); -- entry call
... now proceed to my j + 1st column

But this is unacceptable. The concluding entry call prevents this worker from proceeding until a rendezvous with the next worker has been achieved. Even if the next worker is ready and waiting at its own accept statement, the accept-mediated data transfer forces the sender to wait until the receiver has been scheduled and the data has been transferred into the receiver's address space — again, pointless overhead. Once the bottom-row has been transferred out of its own address space, the sender has nothing more to gain by waiting. In the Lite version, the sender proceeds as soon as its data has been added to TS. However, the next worker may not have reached its corresponding "accept" statement. It may be running behind, and may still be busy on an earlier (further-to-the-left) sub-block; or it may not yet even have begun computing this band. Suppose we have 5 workers and 20 row bands. When the worker computing the fifth band from the top attempts to communicate a bottom row to the worker responsible for the sixth band from the top, this next worker may still be at work on the top band (i.e. on band number 1). (Arranging things
in such a way that there are more bands than workers maximizes parallelism
by minimizing sub-block size. Smaller sub-blocks mean shorter initial waiting
periods until all workers can get started, and a shorter final shut-down period
during which workers gradually drop out.)

So we try something else. Workers execute entry calls to the master in order
to deposit or retrieve bottom rows; the master executes

```ada
loop -- first attempt
select
  accept PutBottomRow(i: integer, vector: VectorType) do
    Buffer(i) := vector
  end PutBottomRow
or
  accept GetBottomRow(i: integer, out vector: VectorType) do
    vector := Buffer(i);
  end GetBottomRow
```

This version doesn't work, because a worker might call "GetBottomRow" before
the row it needs has been deposited. Clearly we need a guarded accept. But
the solution above can't be corrected by installing a guard, because the guard-
condition would depend on the parameter to the entry call. That is, we would
need

```ada
when Available(i) =>
  accept GetBottomRow(i ... ) -- need same binding for i
```

which is not allowed. Families of entries are allowed, but there doesn't seem to
be a way to use them here, because we can't create an array of accept statements;
each family member must be specified explicitly. Not to say that there isn't a
solution along these lines. We could use the technique discussed in chapter 11
of the Ada Rationale: if the GetBottomRow entry accepts a rendezvous only
to discover that it can't meet the client's needs, it instructs the client (via pre-
arranged protocol) to go to the end of the line, by calling the same entry again;
the next time its turn comes at the head of the accept queue, it may be luckier.
The ludicrous complexity of this technique makes it clearly unacceptable (as the
Rationale's authors themselves seem to realize).

Which is just as well; this style of solution — the master task buffers each
bottom row — is undesirable in any case, for reasons discussed in the previous
example. Workers incur the overhead of rendezvous with the master; they may
be forced to wait their turn at the head of the line; workers depositing rows may
have to wait for workers withdrawing rows, and vice versa; as before, workers
depositing rows require no response from the master task in any case, and so
the overhead they incur is especially pointless. This sort of master task is a
breeding-swamp for overhead and bottlenecks.
The solution in this case is to create an array of tasks. Each task executes

```ada
accept PutBottomRow(v1: Vectortype) do
    vector := v1;
end PutBottomRow;
accept GetBottomRow(out v2: Vectortype) do
    v2 := vector;
end GetBottomRow;
```

Note that we need $rc - c$ of these tasks, where we have $r$ row and $c$ column bands. That is, we need one task for each bottom row that is to be passed from one worker to another. (Along the final row band at the bottom, we don’t need to communicate bottom-row data.) We can dispense with guards, because a bottom row must always be deposited before it is claimed.

Note that each one of these many buffers tasks is, from an algorithmic point of view, completely unnecessary. None of these tasks does any computing or speeds up the computation in any way. For all its complexity, it’s difficult to regard this as better than a marginally acceptable solution, if that.

The general point. Processes in a parallel application often need to exchange data. Lite provides an easy and convenient way to do this; the data is simply added to $TS$, labelled in some natural way. In Ada, there is no shared object memory; data must be buffered by tasks explicitly provided for the purpose. Designing the appropriate buffer tasks adds to the complexity of the programming job. It may also confront the programmer with an unpleasant choice: either he provides one or some minimal number of tasks, making each one a potential performance chokepoint and in some cases causing complicated synchronisation problems, or he provides lots of tasks, creating a complex (a pointlessly complex, in algorithmic terms) runtime program structure.

Dynamic process creation, result-parallelism and fine-grain parallelism.

[In the full report.]

4 Ada World Example.

Problem. Provide a bounded-buffer service: clients deposit and withdraw data objects; consumers block if the buffer is empty, producers if it is full.

Figure 3 gives a standard Ada solution. Lite’s version of a solution in this style is given in figures 4 and 5. This Lite program is unidiomatic; we discuss a
task body buffering is
    buffer: array(1..n) of elem;
    i,j: integer := 1;
    count: integer := 0;
begin
  loop
    select
    when count > 0 =>
      accept read(x:out elem) do
        x := buffer(j);
      end;
      j := j mod n+1; count := count-1;
    or
    when count < n =>
      accept write(x: in elem) do
        buffer(i) := j
      end;
      i := i mod n+1; count := count+1;
    end select;
  end loop;
end buffering

Figure 3: Standard Ada version

better approach below. Standard Ada's version of this solution is more concise than the Lite solution in figures 4 and 5.

Having noted this fact, the explanation is significant. In Standard Ada, communication in the remote-procedure-call style is hardwired into the language. When an application actually needs this style of communication, Standard Ada accommodates it neatly. When an application needs some other kind of communication, it's likely to be out of luck. We discussed above how complicated it can be to achieve Lite-style asynchronous communication via Standard Ada's entry calls and tasks.

In Lite, the situation is reversed. Remote procedure call is not built in; if you need it, you must build it yourself, by using an out to send a request followed by an In to receive the result (see figure 5). (In the bounded buffer example, consumers who have invoked the buffer service need to block until a result can be returned.) As the Lite solution shows, it's easy to achieve this effect in Lite — far easier than it was to achieve a Lite-style solution in Standard Ada.
type buffReq(req: reqtype) is
  record
    val: Elem;
  end record;

type EmptyBuffAvail is 0;
type FullBuffAvail is 0;

procedure BufferTask;
Req: buffReq := (Empty, Empty);
begin
  for i := 1 to BuffSize loop
    out EmptyBuffAvail;
  end loop;

in Req
  case Req.reqType is
    when PUT =>
      Buffer(j) := Req.val;
      j := j mod N + 1;
      out FullBuffAvail;
    when GET =>
      out Req(Req.val => Buffer(i));
      i := i mod N + 1;
      out EmptyBuffAvail;
  end case;
end loop;
end BufferTask;

Figure 4: Task definition for the Lite version of the Ada-style solution (see next figure also).
procedure read(X: out Elem) is
Req: buffReq := (GET, 0);
-- I assume 0 is a legitimate value for "val"; this allows me
-- to match this TS tuple against a template whose second field
-- is empty
begin
in FullBuffAvail; -- is there a full?
outin Req; -- equivalent to "out Req; in Req"
X := Req.val;
end read;

procedure write(X: in Elem) is
Req: buffReq := (PUT, X)
begin
in EmptyBuffAvail; -- is there an empty?
out Req;
end write;

Figure 5: The procedures used to invoke the Lite task.

The distributed solution

In Lite, though, this whole approach makes little sense. Standard Ada has no
choice but to provide a task whenever shared data must be buffered. Lite allows
such data to be stored directly in tuple space.

Each buffer slot in the bounded buffer is represented by a TS-object. This
object's first field is an index; its second is either the buffered datum itself,
or empty (if this slot is currently unoccupied). Two other TS-objects store
the current indices of the first unoccupied buffer slot and the first full buffer
respectively.

When a process needs to drop something into the buffer, it starts by consult-
ing the index of the first unoccupied slot, and then incrementing this value. (If
the value was 10, it plans to fill the buffer whose index is 10; the next producer
will fill the buffer whose index is 11, and so on.) Accordingly it execute

in NextFree(Empty); -- what's the index of the next free buffer?
NextFree := NextFree + 1;
out NextFree; -- I'll take it; next guy gets the next free buffer

Now it grabs the 10th buffer, fills it, then drops it back into TS; val is the
datum to be deposited in the buffer:

\[ \text{in: } \text{BufferElt(NextFree, val); -- grab free buffer, fill it, dump it back} \]

This compound inout form is shorthand for (is identical to) the sequence

\[ \begin{align*}
\text{in BufferElt(NextFree, val);} \\
\text{out BufferElt(NextFree, val);}
\end{align*} \]

Note that the in statement grabs a TS-object whose first element is the integer NextFree and whose second is empty. After this transaction, the value of the second field of the BufferElt record is val. (This in doesn't actually change the value of any of BufferElt's fields, because none of its fields were empty. It merely block until the empty buffer for which it is waiting becomes available.)

Processes that need to withdraw an element from the buffer follow the same procedure. First, they consult and increment a NextLoaded index. Next, they wait for the loaded buffer to show up, by executing

\[ \begin{align*}
\text{in NewElt(NextLoaded, empty);} \\
\text{out BufferEltType(NextLoaded + NumBuffs, Empty);}
\end{align*} \]

In this case, the second element of the matching TS-object must not be empty (two empty's don't match). When this In statement completes, NewElt's second field contains the datum from the buffer. Finally, a new, empty buffer is dumped at the end of the line. The end of the line will be NumBuffs down from the full buffer I just withdrew:

There may be processes awaiting empty buffers; if so, one of them (the first to put in its request) is waiting for this buffer (i.e., it's blocked at an in statement awaiting an empty buffer whose first element is the index value NextLoaded + NumBuffs). It grabs this buffer, fills it, and the process continues.

This solutions saves the overhead of creating and scheduling an extra task; users of the bounded buffer no longer need to rendezvous with a task in order to use the buffer; the objects in the buffer are stored in tuple space, and are therefore directly accessible to buffer users, instead of being stuck inside a buffer task's address space. So long as some slots are free and others are full, consumers and producers can access the buffer simultaneously (as they cannot in the Standard Ada-style solutions). Note that the Standard Ada-style solution gets worse as the number of buffers increases: more buffers means more tasks, or greater bottleneck-potential for a single task.
package body BUFFER is
declare
  type FreeIndex is integer;
  type LoadedIndex is integer;
  type BufferEltType is
    record
      index: integer;
      content: Elem;
    end;
end;

procedure put(in val: Elem);
declare
  NextFree: FreeIndex;
  BufferElt: BufferEltType;
begin
  in NextFree(Empty); -- what's the index of the next free buffer?
    NextFree := NextFree + 1;
  out NextFree; -- I'll take it; next guy gets the next free buffer
  inout Elt(NextFree, val); -- grab next free buffer and fill it
end put;

procedure get(out val: Elem);
declare
  NextLoaded: Loaded;
  NewElt: BufferEltType;
begin
  in NextLoaded(Empty); -- what's the index of the next loaded buffer?
    NextLoaded := NextLoaded + 1;
  out NextLoaded; -- next guy takes the next loaded buffer
  in NewElt(NextLoaded, Empty); -- grab the loaded buffer
  out BufferEltType(NextLoaded + NumBuffers, Empty); -- empty it, and replace it
  val := NewElt.content;
end get;

begin
  for i := 1 to BuffSize loop
    out BufferElt(i, empty);
  end loop;

out NextFree(1);
out NextLoaded(1);
end BUFFER;

Figure 6: Ada Lite version
A more complex Ada-world example

[In the next version.]

5 Multiple tuple spaces

[In the next version.]

6 File systems

Lite includes a built-in object-structured file system. Programs can create many tuple spaces; a persistent tuple space plays the role of a file. Objects are deposited in the file via `Space.out` (Space is the name of some persistent tuple space), read via `Space.rd` and undated via `Space.in` followed by `Space.out`.

Who needs an object-structured file system? Returning to the bounded buffer example: suppose this buffer represents long-term rather than temporary storage. A consumer process may withdraw some element that was deposited earlier (minutes, days, years earlier) by a producer that has now terminated. We can use such a structure to coordinate (for example) data-producing and data-analysing or data-visualizing programs. While the data is being produced, all available computing resources are devoted to producing it (or receiving it); later, other programs need to inspect this data.

The distributed bounded buffer program still works. Prefixing tuple space operations with the appropriate tuple space name is the only change required. The Standard Ada solution can't be extended in this way. Standard Ada's tasks and accept statements have nothing to do with files and file i/o.

A more complete discussion of the tuple spaces as object-structured files will be included in the next version.