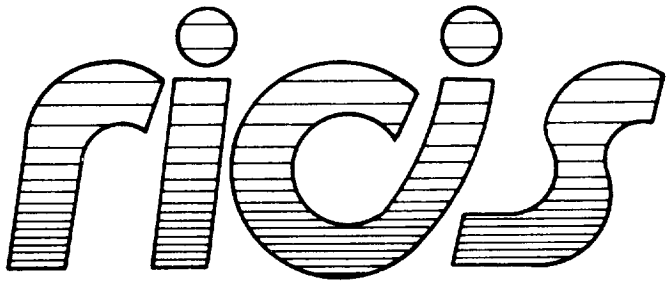
Ada Programming Guidelines for Deterministic Storage Management

David Auty

SofTech, Inc.

January 29, 1988

Cooperative Agreement NCC 9-16
Research Activity No. SE.9



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

Ada Programming Guidelines for Deterministic Storage Management

Preface

This research was conducted under the auspices of the Research Institute for Computing and Information Systems by David Auty of SofTech, Inc. under the technical direction of Charles W. McKay, Director of the Software Engineering Research Center (SERC) at the University of Houston-Clear Lake.

Funding has been provided by the Avionic Systems Division, within the Engineering Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA Technical Monitor for this activity was Terry Humphrey, Data Management Section, Flight Data Systems Branch, NASA/JSC.

Ada Programming Guidelines
for
Deterministic Storage Management
January 29, 1988
(SofTech Document WO-126)

A UHCL/RICIS Report, Contract SE.9

Prepared for

NASA Avionics Systems Division, Research and Engineering
Johnson Space Center

Prepared by

SofTech, Inc.
1300 Hercules Drive, Suite 105
Houston, TX 77058-2747

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

Section 1

OVERVIEW

It has been established through previous reports ([RvsM], [RAL]) that a program can be written in the Ada language such that the program's storage management requirements are determinable prior to its execution. In this report, specific guidelines for ensuring such deterministic usage of Ada dynamic storage requirements will be described. Because requirements may vary from one application to another, guidelines are presented in a most-restrictive to least-restrictive fashion to allow the reader to match the appropriate restrictions to the particular application area under investigation.

1.1 Structure

Section 2 of this report presents the most restrictive guidelines, in that it enumerates programming restrictions sufficient to allow static storage management for most Ada language implementation strategies (as described in the report "Requirements of the Language Versus Manifestations of Current Implementations" ([RvsM])). Subsequent sections describe increasingly more permissive programming guidelines: each section waives one or more restrictions present in preceding sections to provide the programmer greater access to the full power and flexibility of Ada while introducing more significant storage management requirements.

Each section:

1. Explains the scope of enforced and waived restrictions pertaining to the guidelines in question;

2. Provides an analysis of the consequences of waiving particular restrictions in terms of loss of determinism and the additional analysis methods that must be employed in light of the corresponding waivers;

3. Provides an enumeration of any possible simplifications that are implementation or application-specific;

4. Includes one or more examples or case studies showing the use of the guidelines.

1.2 Guideline Impact

The use of these guidelines in the development of Ada software may have an impact on the software engineering process, particularly in terms of portability, guideline implementation, and guideline side effects.

1.2.1 Portability

While consulting the case studies and applying the guidelines detailed in this report, it is important to give adequate consideration to portability issues. As a consequence of the implementation-specific characteristics of storage management, the portability of programs written using these guidelines cannot be guaranteed. Specifically:

1. A program that has sufficient storage under one implementation may run out of storage under another implementation. Testing for adequate available storage must be repeated for each target implementation.

2. The restrictions proposed in this report apply to any reasonable implementation of storage management, such as those described in [RvsM], but it is easy to envision unreasonable but legal implementations for which it is impossible to obtain any assurances about adequacy of storage. Consider, for example, an implementation that does not release storage for a subprogram's activation record upon return from that subprogram. A program compiled under such an implementation would be legal Ada, but could be severely limited in complexity.

1.2.2 Guideline Implementation

The guidelines described in this report can be used in two ways: the guidelines can be imposed on the developers from the outset, or the software can be developed without restrictions and then "reverse-engineered" to comply with the guidelines. Both approaches have specific advantages.

The restricted development approach ensures that system-wide impacts of the guidelines are taken into consideration from very early in the development process. This ensures that design issues of certain restrictions can be dealt with, and helps the testing process by allowing the definition of specific test plans.

The reverse-engineering approach permits the natural, unrestricted development of the system based on the full power and expressiveness of the Ada language, resulting in an openly designed system. This system then serves as a baseline that can be refined for use on a variety of hardware configurations that may require varying levels of constraints to be placed on storage management for the system. For each such system, an analysis takes place that focuses on applying the guidelines of this report by removing specific features from the software. For example, the analysis might result in the conversion of recursive routines to iterative equivalents, or the placing of constraints on unconstrained objects.

The advantages of the reverse-engineering approach include the possibility of acquiring a highly generalized baseline that is unconstrained by current hardware considerations. With the continual refinement of storage devices, it is conceivable that storage management restrictions for a current configuration will not apply to a future configuration. This approach will help to ensure that the software will not become obsolete, and can be upgraded and extended more easily.

1.2.3 Guideline Side Effects

Another important consideration in the use of the guidelines of this report is the likelihood of side effects. Introducing restrictions will often result in a modified "black-box" view of the system or system component. For example, certain guidelines in this report suggest adding constraints to data items or applying additional checks on boundary values. Such modifications may not affect the external specification of a program unit, or modify its functionality, but may result in a modified behavior. The unit's execution speed may differ, it's total storage usage may be changed, different or additional exceptions may be raised implicitly or explicitly, or other differences may exist. Such differences should be identified and clearly documented, particularly when the development approach is oriented to reverse-engineering as discussed previously.

Section 2

GUIDELINES TO GUARANTEE STATIC STORAGE REQUIREMENTS

It has been established in [RvsM] that Ada can be used in such a fashion that the dynamic storage management requirements normally associated with Ada can be constrained or eliminated. The storage management requirements resulting from such constraints are similar to those of more familiar general-purpose languages. In essence, the use of Ada can be restricted to confine the programmer to FORTRAN or HAL-like programming that will allow static determination of storage use. This section will examine such restrictions in detail.

[RvsM] identifies the aspects of the Ada language that result in some form of dynamic storage management requirement. Appendix B summarizes these aspects with examples in the form of figures and code fragments. In particular, these include:

1. Multiple simultaneous subprogram invocations - the number of simultaneously active invocations of a subprogram may not be known until run-time; the subprogram may invoke itself (known as "direct recursion"), two subprograms may invoke each other (known as "mutual recursion"), or a single subprogram may be invoked simultaneously by multiple tasks.

2. Objects with non-static bounds - the size of an object, such as a variable object of an unconstrained array type or discriminated record with unconstrained or variant elements may not be known until run-time.

3. Designated variables - the number of designated variables that a program will attempt to create may not be known until run-time.

4. Task objects - the number of tasks that will exist, and their relative temporal characteristics (i.e., their relationships in time with each other, how many and which ones will exist simultaneously), may not be known until run-time.

Based on this knowledge, an Ada program can be constructed restricting the introduction of these characteristics altogether, thus ensuring fixed storage requirements. Taken together, this list of restrictions perhaps appears more

constraining than is actually the case. In truth, sensible and planned programming techniques will effectively limit many such characteristics inherently. For example, although a directly recursive program may be written such that the depth of the recursion is unknown until run-time, need not be the case; one can frequently bound the depth of recursion without difficulty. The same can be said for the determination of storage requirements for the other characteristics as well. The case studies presented in the sections that follow will elaborate specifically on such programming techniques.

2.1 Restrictions

1. Direct and mutual recursion are prohibited
2. Use of composite objects with non-static bounds is prohibited
3. Use of designated variables is prohibited
4. Tasking (except the environment task executing the main program) is prohibited

2.2 Analysis

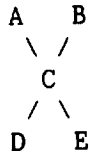
Based on the restrictions presented in this section, the only non-static storage requirements of a program following these guidelines would be those related to subprogram parameters and local data (typically based on a Last-In-First-Out (LIFO) stack arrangement) for iterative subprograms. As such, total storage requirements for such a program can be determined by analyzing and testing the code path(s) resulting in the deepest subprogram call-nesting and largest collection of local data elements.

To perform such an analysis, construct a frame graph to represent the program. It will be a rooted directed acyclic graph, where the root corresponds to the main program. Each path from the root to a leaf of the graph represents a possible set of subprogram or declare-block frames active at the same time. Construct a set of test cases exercising each feasible combination of simultaneously active frames. If sufficient storage exists to run these test cases, the program will never run out of storage.

Notes:

1. Testing all paths in the frame graph is not the same as testing all paths in a flow chart (code paths). A frame graph describes subprogram activation and declare-block execution relationships. In contrast, a code path is generally defined as any segment of code having one entry point and one exit point. Thus, a single subprogram might consist of one or many code paths. For the purposes of determining dynamic storage requirements under the guidelines of this section, only the active frame path is relevant.

2. Some paths from root to leaf are not feasible, i.e., the corresponding combination of calls will never arise because of the logic of the subprograms. Consider the following example: A calls C with parameter value 1 and B calls C with parameter value 2. When called with parameter value 1, C calls D and when called with parameter value 2, C calls E. Then the frame graph is:



   The paths A-C-D and B-C-E are feasible, but the paths A-C-E and B-C-D are not.

2.3 Implementation-dependent Simplifications

2.3.1 Static Storage Allocation

An implementation may provide a pragma (e.g., Pragma STATIC [AUTY]) that indicates to the compiler that static data allocation is to be used where possible. In a sense, the use of such a pragma would have a similar effect to declaring all objects in library packages, in that the objects exist for the duration of program execution. For example, local and parameter data for a subprogram is defined in a static memory area rather than in a subprogram stack. The use of such a pragma would in effect disable recursion and multiple task execution for a subprogram. An obvious benefit of the use of this approach is an improvement in the determinism of storage requirements while limiting the artificial constraints on the programmer that might otherwise be required. Other benefits include the possibility of improved data access efficiency and faster subprogram invocation sequences.

A likely negative result of the use of a static pragma would be an overall (perhaps dramatic) increase in net storage requirements for the program. One of the fundamental benefits of dynamic storage mechanisms is reduced net memory requirements based on the ability to reuse storage. By contrast, the storage used for statically allocated objects is reserved whether or not the objects are actually used.

If such a capability is used, testing and analysis requirements will be similar to those of other languages that use a static allocation scheme.

2.3.2 Composite Objects Passing Mechanism

If the implementation passes composite objects by reference, then composite formal subprogram parameters may be unconstrained, allowing the passing of static objects of arbitrary size. The Ada language definition [LRM] permits specific implementations to determine whether composite object parameters should be passed by copy or by reference. Many implementations choose to pass such objects by reference due to the obvious advantages in efficiency, but such an implementation is not guaranteed. Further, an implementation may use both mechanisms: different calls to the same subprogram may result in the use of either mechanism.

NOTE

> The LRM states that a program is erroneous if its effect depends on the mechanism used for passing parameters. However, dependency of a program on a particular mechanism in order to comply with storage management requirement restrictions does not constitute a dependency of the behavior of the program. Thus the LRM does not rule out such dependencies. The possible side-effect of such a dependency is reduced portability of storage usage tests for the program and consequently reduced portability for the program itself.

Of course, the passing of composite parameters is still governed by the guideline restricting the use of non-static composite objects.

2.3.3 Local Static Storage Allocation

If the implementation allocates storage for all subprogram local objects (including those defined in declare-blocks within the subprogram body) in the subprogram's activation record, then the frame graph can be reduced to a subprogram calling graph. This will reduce the number of required test cases by reducing the total number of graph paths.

If the compiler gives the storage cost of a subprogram call (or if this can be determined by a tool), the paths in the calling graph can be explored analytically instead of by testing. It would be possible, for example, to use a tool that determines calling graphs in conjunction with compiler-or tool-generated storage cost values to provide automated determination of storage requirements for a given program.

2.4 Application-specific Simplifications

None.

2.5 Case Studies - Guideline Examples

The following paragraphs present examples of programming within each of the restricted areas of this section (recursion, designated variables, non-static composite objects, and tasks).

2.5.1 Prohibit Direct or Mutual Recursion

While it is true that recursion is often a very useful programming approach in terms of clear algorithm presentation, it is also true that recursion is never necessary and is actually seldom used. [HOROWITZ] and others have shown that all recursive programs can be written iteratively, and that the iterative version is often more efficient than the recursive version.

The impact of implementing an algorithm iteratively rather than recursively for the most part is a trade-off between clarity and efficiency. A recursive implementation is generally more succinct, particularly where the algorithm in question is recursively defined (e.g., factorial). An iterative implementation, on the other hand, will often enjoy improved efficiency because the overhead of parameter passing and subprogram entry and exit is avoided. Of course, the magnitude of the difference is highly dependent on the algorithm and compiler in question.

As [HOROWITZ] shows, there is a functional equivalence between programs written in either manner. A recursive algorithm can always be converted to an iterative algorithm by following a series of steps described by [HOROWITZ] that essentially simulates the recursive calls of a subprogram by instituting a local stack onto which "parameter" and local data is pushed. Unfortunately, the resulting code may present the same dynamic storage requirements as a recursive program (e.g., if the stack is implemented with access objects.) If the stack is implemented with the use of static objects, a limitation on the number of "recursive" loops is implied, however the same effect can be achieved using true recursive programs (see Section 6 for examples). This approach is most useful when the software is to be written in languages that do not support recursive programming.

Many recursive programs that do not involve a great deal of local or parameter data can be converted to iterative equivalents with a streamlined approach that amounts to replacing recursive calls with while-loops. Such streamlining may change a function's storage requirements from being dynamic and dependent on its parameters to being entirely static and easily determinable. For example:

```
function FACTORIAL (N: positive) return positive is
begin
     if n = 1 then
          return 1;
     else
          return n * FACTORIAL (n-1);
     end if;
end FACTORIAL;
```

can be written as an iterative algorithm by replacing the IF and RETURN statements with WHILE-loops and assignments as shown below. Note that an additional piece of local data is required to track the factorial value that was passed along the stack as a return parameter in the recursive version.

```
function FACTORIAL (N: positive) return positive is
   fact : positive := 1;
begin
   for I in 2 .. n loop
      fact := fact * n;
   end loop;
   return fact;
end FACTORIAL;
```

The result of this transformation is better determinism of storage requirements. In the case of the recursive FACTORIAL, we know that a POSITIVE parameter N will be stored on the subprogram stack with each call to FACTORIAL, but we do not know the depth of that stack because the number of calls to FACTORIAL is directly proportional to the value of N. In the case of the iterative FACTORIAL, there will be exactly one call to FACTORIAL for a given calculation.

NOTE

To be precise, the maximum storage requirements for the recursive FACTORIAL are also deterministic. We know that for each call to FACTORIAL, there will be N-1 additional calls to FACTORIAL. Hence the total number of calls for a given N will be N. The maximum value of N is known by the parameter type to be POSITIVE'last; the maximum depth of recursion is thus also POSITIVE'last. Therefore, a test to determine whether available storage is adequate in the worst case would include the call:

```
x := FACTORIAL (POSITIVE'last);
```

Although one would expect that such a test is not practical, Section 6 will exploit the fact that the maximum depth of recursion of some subprograms is determinable based on the range its parameter values, thus lifting the recursion restriction.

Of course, cases where a transformation of a program from recursive to iterative is simple are not always evident, and the transformation may not be intuitive. In these cases, the approach described by [HOROWITZ] may be preferred.

2.5.2 Prohibit Composite Objects with Non-static Bounds - Declarations

The use of non-static bounds for composite objects (arrays and records) is a convenient and useful feature of Ada. The dynamic storage risks of using non-static bounds can be minimized or eliminated as described in Section 5 of this report. The restrictions on the use of non-static bounds affect two areas: non-static composite data declarations and non-static composite parameters.

In the case of data declarations, consider the subprogram SORT_INPUT_DATA which reads an arbitrary list of integers from the default input device, sorts it through a call to some subprogram SORT, then writes the sorted list to the default output device. In this example, we assume that the appropriate I/O and sort routines have been made available.

```
procedure sort_input_data (n : in  integer) is

     data_list : LIST (1..n);

begin

     for i in 1..n loop
          get (data_list(i));
     end loop;

     sort (data_list);

     for i in 1..n loop
          put (data_list(i));
     end loop;

end sort_input_data;
```

Note that the length of array DATA_LIST is not determinable prior to run-time within the given context. This subprogram is therefore not permitted under the restrictions of this section. Similar restrictions inherent in a FORTRAN- or HAL-like implementation could be overcome as follows:

```
procedure sort_input_data (n : in  integer) is

    max_size  : integer constant := 25;      -- or some other value
    data_list : LIST (1..max_size);

    OUT_OF_RANGE : exception;

begin

    if n >= max_size then
        raise OUT_OF_RANGE;
    end if;

    for i in 1..n loop
        get (data_list(i));
    end loop;

    sort (data_list);

    for i in 1..n loop
        put (data_list(i));
    end loop;

exception

    when OUT_OF_RANGE =>
        put_line ("Value out of range.");
    when others =>
        null;

end sort_input_data;
```

One side effect of this solution is that the significant length of the sort list is unknown. Solutions to this side effect are presented later in this section.

As Section 5 describes, the same deterministic effect can be accomplished more elegantly with the use of an appropriate subtype for the input value N. Note also that the exception OUT_OF_RANGE need not have been defined, nor is the check of the value of N needed. If these are removed, CONSTRAINT_ERROR will be raised within the first FOR-loop, which can then be handled by an exception handler either within the subprogram or externally. The implementation shown above, however, serves to avoid entering the loop in the first place, and also precisely identifies the nature of the error.

The significance of the above alternative is that, although the specification of the subprogram is the same as with the original implementation (that is, the subprogram receives the same parameters and produces the same output when the value is within range), the run-time storage requirements of the alternative implementation are readily determinable: the worst case requirements are directly related to the definition of the MAX_SIZE constant (in this case, 25).

2.5.3 Prohibit Composite Objects with Non-static Bounds - Parameters

The other area of impact for the "no non-static composite objects" restriction is that of subprogram parameters. For example, the iterative SORT procedure described below would not be permitted under this restriction because the parameter LIST is defined as an unconstrained array of integers:

```
-- for this subprogram, type LIST is array (1..<>) of INTEGER;

procedure SORT (a : in out LIST) is

     j : POSITIVE;
     t : INTEGER;

begin
     for i in a'range loop
          j := i;
          for k in j+1 .. a'last loop
               if a(k) < a(j) then
                    j := k;
               end if;
          end loop;
          t := a(i);
          a(i) := a(j);
          a(j) := t;
     end loop;
end SORT;
```

As a result, the size of the passed array at any given invocation is not determinable prior to run-time. One alternative implementation similar to that used above would be as follows, where LIST is redefined as a constrained array of integers:

```
-- for this subprogram, type LIST is array (1..25) of INTEGER;

procedure SORT (a : in out LIST) is

     j : POSITIVE;
     t : INTEGER;

begin
     for i in a'range loop
          exit when a(i) = END_OF_LIST;
          j := i;
          for k in j+1 .. a'last loop
               exit when a(k) = END_OF_LIST;
               .
               .
               .
end SORT;
```

Here, the unconstrained LIST parameter is replaced with a constrained array of length 25. Although the maximum storage requirements are now known, further bookkeeping must be maintained to ensure that only the significant values in the list are sorted. Above, the last significant value in the array is followed by a constant called END_OF_LIST. Based on this, the SORT routine is able to detect the end of the list of values to be sorted.

```
a := (4,3,67,5,12,3,4,66,1234,-4,18,END_OF_LIST, others => 0);
```

Alternatively, the length of the list might be passed as an additional parameter to the sort routine:

```
procedure SORT (a : in out LIST; length : in integer) is ...
```

As the implementation-dependent simplifications described above indicate, this restriction need not extend to unconstrained parameters if the implementation passes composite objects by reference rather than by copy. The Ada language definition allows either approach or even a mixture of both, at the discretion of the implementor. If the implementation takes the "by reference" approach in all cases, then the passing of non-static arrays can be permitted without danger. If the implementation does not use this approach in all cases, then the equivalent could be accomplished by the application by passing composites with the use of access objects. However, the restriction against the use of designated variables has not yet been waived (below).

2.5.4 Prohibit the Use of Designated Variables

The restrictions of this section explicitly prohibit the use of designated variables, and, by definition, access types and objects of access types. This may be overly restrictive, as designated variables can be used in a deterministic fashion (see Section 5), however it is possible to program within these restrictions if necessary. For example, consider the following procedure which uses a linked-list structure to implement a First-In-First-Out (FIFO) queue:

```
package body DISPATCHER is

     type DISPATCH_PACKET;
     type DISPATCH_LINK is access DISPATCH_PACKET;
     type DISPATCH_PACKET is
          record
               TSC_ID    : tsc_id_type;
               START_TIME: time;
               NEXT      : DISPATCH_LINK;
          end record;

     type DISPATCH_QUEUE_TYPE is
          record
               COUNT: integer       := 0;
               FIRST: DISPATCH_LINK := null;
               LAST : DISPATCH_LINK := null;
          end record;

     DISPATCH_QUEUE : DISPATCH_QUEUE_TYPE;

     procedure INITIALIZE ....
     procedure REMOVE ....

     procedure INSERT (tsc_id: in tsc_id_type) is     -- This procedure
                                                      -- assumes the queue
                                                      -- has been
          packet : dispatch_link;                     -- initialized.
     begin
          packet := new dispatch_packet'(tsc_id     => tsc_id,
                                         start_time => CLOCK,
                                         next       => null);

          dispatch_queue.last.next := packet;
          dispatch_queue.last := packet;
          dispatch_queue.count := dispatch_queue.count + 1;
     end INSERT;
end DISPATCHER;
```

Here, the queue is implemented as a linked list of records of type DISPATCH_PACKET. Each packet contains a link to the next packet. Further, there are two links defined as part of the queue itself which keep track of the start and end packets in the queue. This data structure, which amounts to a dynamically-sized stack, can be bounded and implemented as follows:

```
package body DISPATCHER is

   null_link    : integer constant :=  0;
   max_dispatch : integer constant := 50;

   type DISPATCH_LINK : integer range null_link .. max_dispatch;

   type DISPATCH_PACKET is
       record
           TSC_ID     : tsc_id_type;
           START_TIME : time;
       end record;

   type DISPATCH_QUEUE_TYPE is
       record
           COUNT       : integer := 0;
           QUEUE_ENTRY : array (1 .. max_dispatch) of DISPATCH_PACKET;
           FIRST       : DISPATCH_LINK := 1;
           LAST        : DISPATCH_LINK := 1;
       end record;

   DISPATCH_QUEUE : DISPATCH_QUEUE_TYPE;

   procedure DELETE ....

   procedure INSERT (tsc_id: in tsc_id_type) is

   begin

       dispatch_queue.last := dispatch_queue.last + 1;

       dispatch_queue.queue_entry(dispatch_queue.last) :=
                                          (tsc_id     => tsc_id,
                                           start_time => CLOCK);

       dispatch_queue.count := dispatch_queue.count + 1;

   end INSERT;

end DISPATCHER;
```

Note that the NEXT field of the DISPATCH_PACKET record is no longer needed because each packet can assume that the next element in the queue is the array element that follows it sequentially. Similarly, an initialization procedure is no longer needed to allocate the first packet and set the initial pointers to it. In any case, the total size of the queue is determinable prior to runtime. Adequate analysis and testing must ensure that the MAX_DISPATCH limit is adequate.

Similarly, more complex dynamic data structures such as doubly-linked queues where inserts and deletions can occur at any point within the queue can be modeled by extension of the approach used above, although with some difficulty. Additional data structures, such as a "free list" array that tracks free packets, must be maintained to provide the desired effect.

2.5.5 Prohibit Tasking

All programs that use a concurrent model of design can be implemented sequentially, though possibly with significant loss in clarity. Appendix A presents an example of two programs written to the same specification. Although they are functionally identical, one program is purely sequential while the other makes use of Ada tasking.

Section 3

INTRODUCING A FIXED NUMBER OF TASKS

It is not necessary to eliminate all tasks from a program to provide reasonable determinism of storage management requirements. There are four areas of risk regarding storage management when using tasks:

1. Multiple Simultaneous Invocations of a Subprogram
2. Variable Arrays of Task Objects
3. Task Objects Declared in a Variable Loop
4. Task Object Declared in Recursive Subprogram

The latter three situations can be categorized as the potential invocation of an "Unknown Number of Tasks" and are dealt with separately in Section 7 of this report. The first situation is a concern even where the number of tasks to be executed is known prior to runtime. By the restrictions of this section, we know that there are a known number of tasks and that they begin execution during initial program elaboration. What we do not know, however, are the temporal characteristics of the tasks and the subprograms that are called by those tasks. We do not know how many of the tasks will be executing simultaneously or for how long their executions will overlap, since such execution patterns are highly dependent on implementation, application, and transient factors such as data input.

Despite this, it is possible through careful analysis to demonstrate through "worst-case" scenarios that available storage will be adequate to meet the needs of any fixed-task situation.

3.1 Restriction Waived

The use of a fixed number of tasks is permitted. Each task is declared either as a single task or as a task object that is not part of an array. All tasks are declared in library packages or within the main subprogram unit.

3.2 Analysis

The frame graph for each task is constructed. Each graph is a rooted directed acyclic graph. The root for the main task corresponds to the main program and the root for each other task is the corresponding task body. A dummy task is added with one entry called Freeze_Caller and a null task body. A call on this entry has the effect of permanently blocking a task at the point of an entry call. During testing, calls on Freeze_Caller should be inserted at different points in the bottom level subprograms of each task frame graph so that all feasible active-subprogram combinations of each task are attained simultaneously with all feasible active-subprogram combinations of all other tasks.

A set of test cases based on this approach will demonstrate that the storage capacity of the system is adequate under all conditions.

3.3 Implementation-dependent Simplifications

If the compiler enforces limits imposed by STORAGE_SIZE representation specifications for task types, then a limit can be imposed for each task object and tasks can be tested individually. This greatly reduces the number of combinations that must be tested and eliminates the need for the dummy task. There should also be an integrated test to validate that sufficient storage exists for all of the task stacks of the sizes specified.

If the implementation can be instrumented to determine the amount of storage in use by a task at a given point, each task is tested individually to determine the active-subprogram combination at which the task's storage usage peaks. Then all tasks are tested together with each task at its point of maximum storage usage. This minimizes the number of required combinations while ensuring that the worst-case combination is tested.

3.4 Application-specific Simplifications

If two tasks interact in such a way that one task is at its deepest depth of subprogram calling while the other is at its shallowest, and if the implementation allows the same storage to be used for more than one task stack, the approaches above may be overly pessimistic. However, the analysis required to establish that a program is safe because of this task interaction (assuming that safety could not be established in the absence of this interaction) is quite complex, and it is not pursued further in this report. For example, such an analysis would include determining that fragmentation does not overly constrain the ability of one task to make use of storage released by the other task.

3.5 Case Study

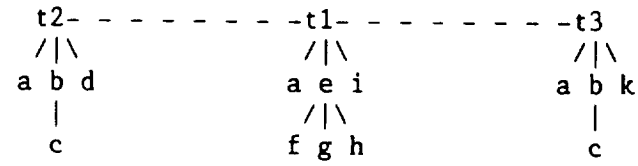
The daily routine of a household consists of a mixture of tasks (such as cleaning the laundry, cooking, shopping, and so forth) that are performed by individuals within the household. In a household with more than one person, it would not be equitable or efficient to perform those duties in a sequential manner. More likely, the daily routine is divided up among the members of the household who perform their individual duties independently and concurrently. The program skeleton below crudely depicts a few such chores:

```
package body daily_procedures is
   procedure select_clothing   is ...
   procedure buy_items         is ...
   procedure punch_timeclock   is ...
   procedure sleep             is ...

   procedure catch_bus is
      procedure pay_exact_fare is ...
   begin
      pay_exact_fare;
   end catch_bus;

   procedure do_household_chores is
      procedure do_laundry     is ...
      procedure do_cooking     is ...
      procedure do_cleaning    is ...
   begin
      if laundry_dirty then
         do_laundry;
      end if;
      do_cooking;
      if house_dirty then
         do_cleaning;
      end if;
   end do_household_chores;
end daily_procedures;

with Daily_Procedures;
procedure daily_routine is
   task go_shopping;
   task go_to_work;
   task body go_shopping is separate;
   task body go_to_work  is separate;
begin
   select_clothing;
   do_household_chores;
   sleep;
end daily_routine;

separate (daily_routine)
task body go_shopping is
begin
   select_clothing;
   catch_bus;
   buy_items;
end go_shopping;

separate (daily_routine)
task body go_to_work is
begin
   select_clothing;
   catch_bus;
   punch_timeclock;
end go_to_work;
```

Here, the daily routine is depicted by three tasks: the main task (procedure daily_routine) and two single task objects (tasks go_shopping and go_to_work). Since the tasks are single, this program conforms to the restrictions of this section. For an analysis of the storage requirements of this program, we will first present the calling graphs for each task. For simplicity, a short identifier will be assigned to each subprogram or task unit as follows:

UNIT	ID	UNIT	ID
daily_routine	t1	do_household_chores	e
go_shopping	t2	do_laundry	f
go_to_work	t3	do_cooking	g
select_clothing	a	do_cleaning	h
catch_bus	b	sleep	i
pay_exact_fare	c	punch_timeclock	k
buy_items	d		

A complete calling graph for this program would appear as follows:



Removal of the dashed lines provides three separate calling graphs, one for each task. Based on these graphs, the analyses described in Sections 3.2, 3.3, and 3.4 then can be applied to each calling graph individually to determine storage management requirements for program execution.

Section 4

INTRODUCING DESIGNATED VARIABLES

The use of designated variables is not risky in and of itself in terms of storage management requirements, however usage such that the number of allocations cannot be determined prior to run-time and the potential for storage fragmentation adds a measure of risk to the use of designated variables. This section permits the controlled use of designated variables.

4.1 Restriction Waived

The use of allocators to allocate objects of any constrained subtype other than a task type or a subtype containing a task-type subcomponent is permitted. The programmer is restricted to a maximum number of allocations if the total number of allocations is not otherwise analytically determinable.

4.2 Analysis

To ensure that sufficient storage is available for all Ada implementations, the worst case scenario of storage usage must be tested. The worst case is defined as the case in which no deallocation occurs by garbage collection or the use of UNCHECKED_DEALLOCATION, since these features are not required by the language. Because all allocated storage will be from "new" storage rather than "reclaimed" storage, fragmentation is not an issue if this approach is taken.

For each elaboration of an access-type declaration, the maximum number of times that an allocator corresponding to that access type will be evaluated must be established. For an access type declared in a library package, this maximum is defined as the number of times during the life of the program that such allocators will be evaluated. For an access type declared in a subprogram, it is defined as the maximum number of times such allocators will

be evaluated during any one activation of the subprogram (excluding the time that a recursive invocation of the same subprogram is active, as the non-recursion restriction has not yet been waived). Because this is a worst-case analysis, it is the total number of allocator evaluations that is relevant, not the net number of variables allocated when deallocated variables are subtracted.

Establishment of this maximum may be based on analysis of the algorithm, although a maximum might be imposed a priori on the algorithm writer when necessary and hard-coded into the program. Examples of such analysis and limits are provided in Section 4.5.

Once the maximum is established, a test program must be created by modifying the program. Upon entry to the scope in which each access type is declared, the test program allocates the maximum number of designated variables as specified. The test program may then be tested as described in earlier sections. If the test does not raise STORAGE_ERROR, execution of the original program should not.

It is also important to consider implicit storage allocation, such as that arising on the return of an object through an unconstrained function return value, or a discriminated record returned from a procedure. Not all storage allocations are immediately obvious.

4.3 Implementation-dependent Simplifications

4.3.1 Analysis Based On Overhead Information

In considering the amount of storage consumed by an allocation, it is important to take into account overhead required by the given implementation, including:

1. storage used for control information

2. storage rendered unusable because the implementation only performs allocation in fixed-sized blocks and some part of the block will be left unused.

If the compiler or its documentation provides information about object overhead, or about the net sizes of objects in unconstrained array and record types, allocations of objects in such types can also be permitted. Storage requirements can be tested by determining the maximum amount of storage that will be required rather than the number of allocations. The worst-case method of establishing this maximum is to take the maximum number of allocations and multiply it by the size of the largest object that will be allocated. It may be possible to establish a maximum based on varying sizes of allocated objects of a given type rather than the largest object if such sizes can be determined analytically or by a priori requirements on the program as with constrained types. For example:

```
type list is array (integer range <>) of integer;
type access_list is access list;
new_list_1 : access_list;
new_list_2 : access_list;
new_list_3 : access_list;
...
new_list_1 := new list (1..18);       -- 18 elements
new_list_2 := new list (2..5);        --  4 elements
new_list_3 := new list (6..47);       -- 42 elements ...
```

Within this context, the largest object of type LIST that is ever allocated contains 42 elements, with three total allocations. Thus a pessimistic maximum of (42 * 3 = 126) can be established. Alternatively, the simplicity of this example permits a more precise maximum of (18 + 4 + 42 = 64) to be established.

Once the maximum storage requirement is determined, the test method described above can be used to allocate the storage.

4.3.2 Analysis Without Overhead Information

If overhead information is not available, the worst case (i.e., the most rigorous test) can usually be determined by allocating the smallest possible object of the type a number of times equal to the total number of elements to be allocated. In the example above, this would manifest as 64 allocations of an object of type LIST with a length of 1:

```
new_list : access_list;
...
for i in 1 .. 64 loop
   new_list := new list(1..1);
end loop;
```

Note that this example assumes that no garbage collection takes place to reclaim allocated storage that is no longer designated by an access value (with each iteration of the loop, the NEW_LIST access value designates a different LIST object, destroying any access to the previous object. As such, a garbage collection scheme may deallocate the related storage). For implementations that provide garbage collection, this situation can be averted by retaining all access values, as with the array in the following example:

```
new_list : array (1..64) of access_list;
...
for i in new_list'range loop
   new_list(i) := new list(1..1);
end loop;
```

4.3.3 Analysis Based on Specification of Storage Requirements

If the compiler accepts STORAGE_SIZE representation clauses for access types, and storage is reserved when the type is delcared, they can be included in the program based on the analysis of the maximum amount of storage that will be needed. There is then no need to transform the program by adding allocations: exercising each path in the calling graph will ensure that the implementation is able to reserve collection regions of the required capacity.

4.4 Application-Specific Simplifications

If the algorithm performs unchecked deallocation for a constrained designated type, and if the implementation ensures that freed storage in such a collection region is always available for reallocation, storage requirements can be tested by having the test program allocate only the maximum net amount of storage that will be allocated at any one time. Additionally, this net maximum can be used for determining the value in STORAGE_SIZE representation clauses.

4.5 Case Study

An example of a program where the maximum number of allocations is determinable analytically is the DISPATCHER example from Section 2.5 which is repeated below.

```
package body DISPATCHER is

    type DISPATCH_PACKET;
    type DISPATCH_LINK is access DISPATCH_PACKET;
    type DISPATCH_PACKET is
        record
            TSC_ID    : tsc_id_type;
            START_TIME: time;
            NEXT      : DISPATCH_LINK;
        end record;
    type DISPATCH_QUEUE_TYPE is
        record
            COUNT: integer       := 0;
            FIRST: DISPATCH_LINK := null;
            LAST : DISPATCH_LINK := null;
        end record;
    DISPATCH_QUEUE : DISPATCH_QUEUE_TYPE;

    procedure INITIALIZE ....
    procedure REMOVE ....
```

```
    procedure INSERT (tsc_id: in tsc_id_type) is    -- This procedure assumes
                                                    --    the queue has been
        packet : dispatch_link;                     --    initialized.

    begin

        packet := new dispatch_packet'(tsc_id     => tsc_id,
                                       start_time => CLOCK,
                                       next       => null);

        dispatch_queue.last.next := packet;
        dispatch_queue.last := packet;
        dispatch_queue.count := dispatch_queue.count + 1;

    end INSERT;

end DISPATCHER;
```

Procedure INSERT allocates exactly one object of type DISPATCH_PACKET. Because DISPATCH_LINK is defined in a library package, the total number of allocations that will be created during the life of the program must be determined. That total can be determined analytically in many cases, such as in the main program below:

```
with dispatcher;
with transient_data; use transient_data;
with task_info;

procedure main is
begin
   case transient_data.system_status is

      when CRITICAL  =>
            dispatcher.insert (alert);
            dispatcher.insert (report);
            dispatcher.insert (watch);
      when NORMAL    =>
            dispatcher.insert (report);
            dispatcher.insert (watch);
      when HIBERNATE =>
            dispatcher.insert (sleep);
      when GLITCH    =>
            dispatcher.insert (alert);
            dispatcher.insert (report);
            dispatcher.insert (sleep);

   end case;

end main;
```

In this example, no more than three designated objects will ever be allocated. A valid test program would initiate a SYSTEM_STATUS value of either CRITICAL or GLITCH to ensure that adequate storage will be available for these cases.

If the test of SYSTEM_STATUS occurs on a continual basis rather than once (e.g., if the case statement above is placed within a continuous loop), a limit may be imposed on the writer of the program to ensure that only a specific number of allocations occur. For some applications it may be possible to place a limit on input values driving such a loop. In other cases, an allocation counter can be maintained for the specific access type. For example, a counter (TOTAL_ALLOC) might be delegated for the DISPATCH_LINK type as shown below:

```
...

MAX_ALLOCATIONS : integer constant := 50;
type DISPATCH_QUEUE_TYPE is
    record
        TOTAL_ALLOC : integer range 0 .. MAX_ALLOCATIONS := 0;
        COUNT       : integer                    := 0;
        FIRST, LAST : DISPATCH_LINK := null;
    end record;
DISPATCH_QUEUE : DISPATCH_QUEUE_TYPE;


procedure INSERT (tsc_id: in tsc_id_type) is

    packet : dispatch_link;

begin

    if dispatch_queue.total_alloc >= MAX_ALLOCATIONS then

       take_some_action;

    else

       dispatch_queue.TOTAL_ALLOC := dispatch_queue.TOTAL_ALLOC + 1;

       packet := new dispatch_packet'(tsc_id     => tsc_id,
                                      start_time => CLOCK,
                                      next       => null);
       ...
    end if;
end INSERT;
```

As noted previously, this approach is worst-case oriented in that it does not take into account storage that is reclaimed by garbage collection or calls to UNCHECKED_DEALLOCATION. This is clearly not practical for applications where a potentially infinite number of allocations will take place during the life of the program. For such cases, the net amount of storage (total allocations less deallocations) must be used to provide practical testing. If the implementation supports STORAGE_SIZE representation specifications, determinism can be maintained assuming:

1. The total storage set aside for the given type is greater than the net maximum that will ever be required, and

2. The implementation ensures that storage freed by calls to UNCHECKED_DEALLOCATION is always available for reallocation.

In the context of the DISPATCHER example, a DELETE procedure might be made available as a complement to the INSERT procedure. The DELETE procedure would remove a given packet from the DISPATCH_QUEUE, then deallocate the related storage via a call to UNCHECKED_DEALLOCATION. If a determinable number of calls to INSERT are followed by a complementary number of calls to DELETE, the amount of storage required is determinable by multiplying the maximum number of inserts that do not have corresponding deletes at any given time by the amount of storage required for a single allocation. This amount can then be used as the value for a STORAGE_SIZE representation specification.

Section 5

INTRODUCING NON-STATIC ARRAYS

Thus far, only array objects with static bounds have been permitted. This section relieves that restriction by exploiting the fact that the maximal storage requirements of a given array object can be determined from the bounds of the array type (or subtype) even when those requirements cannot be determined directly from the bounds of the object itself.

5.1 Restriction Waived

Arrays with non-static bounds are allowed, provided that the components of the arrays are not task objects or objects with task-type subcomponents. The restriction is waived for both declared objects and parameters.

5.2 Analysis

The maximum size for each array object is determined by analysis of the bounds definition of each array type or subtype. At most, the maximum size of an array object is a function of the the range of the index type of the array type or subtype. For example, for the following definition of type int_array,

```
type int_array is array (INTEGER range <>);
```

the maximum size of any object of this type can be defined as:

```
(INTEGER'last - INTEGER'first + 1)
```

Further, the bounds of an array object of this type will provide further bounds information, such as

```
A : int_array (1..10);
```

which obviously indicates a size of 10, or

```
A : int_array (START .. FINISH);
```

which indicates a size of (FINISH - START + 1). If START and FINISH are variables, the range of their respective type or subtype will provide the specific values needed for this calculation. For example, examine the following context:

```
subtype LOWER_BOUND is INTEGER range 1 .. 15;
subtype UPPER_BOUND is INTEGER range 80 .. 132;
procedure make_array (START : LOWER_BOUND; FINISH : UPPER_BOUND) is
   A : int_array (START .. FINISH);
...
```

The maximum size of array A can be determined by supplying the range data of the boundaries into the formula provided above:

```
maximum_size = UPPER_BOUND'last - LOWER_BOUND'first + 1
```

which in this case is (132 - 1 + 1) or 132.

Based on this information, testing the program for adequate storage is a simple matter of providing test cases that exercise the greatest range of bounds that the array can be expected to accommodate.

5.3 Implementation-dependent Simplifications

None.

5.4 Application-specific Simplifications

If all non-static arrays are bounded by variables whose types or subtypes have ranges that reflect true needs (as shown in the examples above), testing can be both rigorous and realistic. This observation is more a matter of

appropriate Ada programming style than restrictive guidelines. Further, violations of such bounds will often be detected at compile-time (e.g., passing a static INTEGER value as a parameter that is out-of-range of a formal integer subtype that will be used as an array bounds in the called subprogram) resulting in higher reliability than many equivalent non-Ada programs.

5.5 Case Study

Although it is possible to write programs with arrays of arbitrary length, proper Ada programming style will prevent non-determinism of maximum storage requirements. For example, assume some function PAD_1 pads a given character string with a given pad character. A call to PAD_1 with the parameters ("12.34", 10, '-') will result in a return value of "-----12.34":

```
function PAD_1  (STRING_OBJECT : STRING;
                 width         : NATURAL;
                 pad_char      : CHARACTER := ' ') return STRING is

   s_out : STRING(1..width);

begin

   s_out (1 .. width - STRING_OBJECT'length) := (OTHERS => pad_char);

   s_out (width - STRING_OBJECT'length + 1 .. width) := STRING_OBJECT;

   return s_out;

end PAD_1;
```

In the above implementation of PAD, a local array S_OUT is declared with a non-static upper bound of subtype NATURAL. For a given Ada implementation, the maximum storage requirements for a call to this subprogram are determinable analytically or verified through testing based on the value of NATURAL'last, as with the parameters: ("this is a test", NATURAL'last, '*').

It is reasonable to assume that few applications would require the use of the full range of values of NATURAL for the width parameter. For instance, an application might use a PAD routine to pad characters for display on an 80-column CRT, eliminating the need to accommodate pad widths of greater than

80 characters. The PAD function should then be written with appropriate formal parameters:

```
function PAD_2  (STRING_OBJECT : STRING;
                 width         : CRT_LINE_LENGTH;
                 pad_char      : CHARACTER := ' ') return STRING;
```

Here, CRT_LINE_LENGTH might be defined as a subtype of INTEGER in the range of 1 .. 80. Thus, the test for maximum storage requirements would be based on CRT_LINE_LENGTH'last, which is certain to be far smaller than NATURAL'last. Any attempt to pass a value outside the range of CRT_LINE_LENGTH will be rejected by the compiler (for a static value) or at run-time by raising a CONSTRAINT_ERROR.

The CRT_LINE_LENGTH restriction could also be carried out from the calling side. The first PAD function (PAD_1) could be called as follows:

```
subtype CRT_LINE_LENGTH is integer range 1..80;
width : CRT_LINE_LENGTH;
...
get (width);

declare
   pad_string : string (1..width);
begin
   pad_string := pad_1 ("test", width, '#');
end;
```

Although the PAD function can accommodate strings of arbitrary length, the context above ensures that the requested padding width will be within the range of CRT_LINE_LENGTH (between 1 and 80). An input value outside that range will raise a constraint error at the point of the GET call.

The advantage to this approach is that the first PAD function can accommodate a wider variety of input values than the second PAD function, which is limited to strings of range CRT_LINE_LENGTH'range. The disadvantage is that the burden of proof of adequate storage is now placed on the user of the routine rather than the writer of the routine. As a result, testing must be more rigorous.

In summary, the maximum storage requirements of a given array are always deterministic analytically as a function of:

```
(array_type'length * array_type'size) + overhead_storage
```

where array_type is the type or subtype for a given array and overhead_storage is any additional storage requirements that an implementation might have in connection with storing and manipulating arrays. Knowledge of array_type'length is all that is needed to test for storage capacity adequacy, using the frame-graph testing approach that has been used thus far. If the test program applies data that will result in the creation of arrays of the maximum size (length = array_type'length), the program will be shown to have adequate storage.

Section 6

INTRODUCING RECURSION

The storage management requirements of a recursive program are only determinable if the depth of recursion of the program is determinable. It is possible to envision a program with an indeterminable maximum depth of recursion, such as a program whose recursive properties are dependent on transient input data of arbitrary size or duration. In practice, however, most recursive programs will have an identifiable maximum depth.

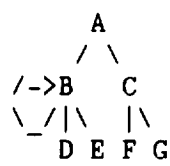
6.1 Restriction Waived

Directly recursive and mutually recursive subprograms are permitted where the depth of recursion is deterministic.

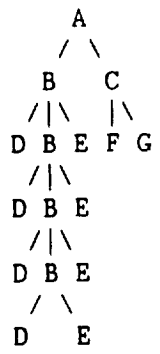
6.2 Analysis

The nature of the recursive properties of the program in question must be analyzed in order to determine the depth of recursion for the program. The parameters, input data, or conditions that impact recursive depth must then be bound. Once bounds have been established, testing can occur as with iterative programs based on an acyclic calling graph.

The calling graph for recursive programs is not acyclic. However, because a maximum depth of recursion has been established, an acyclic equivalent can be built and analysis can proceed as before. For example, if the calling graph is:



and a maximum depth of 4 is determined for B's recursion, the reconstructed calling graph is:



The amount of storage needed at each level is a function of the parameters passed at each level. It might be that different amounts of storage are required at each level, hence the necessity for testing based on the transformed graph. In other words, because recursion can occur through multiple paths, it is not adequate to assume that the storage requirements of each level are identical.

6.3 Implementation-dependent Simplifications

None.

6.4 Application-specific Simplifications

None.

6.5 Case Study

In a simple case, depth of recursion is directly related to a parameter value in a linear fashion. An example of such a program is the FACTORIAL function described in Section 2.5 and repeated below:

```
function FACTORIAL (N: positive) return positive is
begin
    if n = 1 then
       return 1;
    else
       return n * FACTORIAL (n-1);
    end if;
end FACTORIAL;
```

The depth of recursion for FACTORIAL(X) is X; a call of FACTORIAL(5) will exhibit five levels of recursion. Thus linearly recursive programs can be bound easily by applying appropriate constraints to their formal parameters. For FACTORIAL, the formal parameter N is of type POSITIVE which provides an implementation-dependent bound that is probably quite large. A particular application might expect values of n to be less than 25, in which case an integer subtype in the range (1 .. 25) should be defined and used for the parameter N. Alternatively, the bound could be placed on the actual parameter, although this approach presents a higher risk because checks for out-of-range values must be performed at each call to the subprogram rather than once as part the subprogram itself.

A case where depth of recursion is not directly related to a parameter value is in the procedure TRAVERSE which performs an "inorder" traversal of a binary tree:

```
procedure TRAVERSE (T:POINTER) is
begin
    if T /= null then
       TRAVERSE (T.LEFT);
       process (T);
       TRAVERSE (T.RIGHT);
    end if;
end TRAVERSE;
```

The depth of recursion of TRAVERSE is dependent on the size (depth) of the tree T. Because T is implemented as a linked list using access objects, the maximum size of T is arbitrary; the depth of recursion is thus unknown and unlimitable within the given context. A depth could be imposed indirectly by imposing a limit on the size of T during it's construction, using the analysis methods described in Section 4. This approach will ensure deterministic storage requirements, but only where adequate cross-testing is performed to

guarantee the limitations that are assumed by the designers of TRAVERSE. Such cross-testing requires a coordinated systems development and verification effort.

Additional safety can be imposed within the direct context of TRAVERSE that will guarantee a known level of recursion. A counter is maintained that tracks the current recursion level. This method is analogous to the counting method used to track allocations of dispatch packets in Section 4.5.):

```
package body TRAVERSE_PACKAGE is

   MAX_RECURSIONS  : integer constant := 25;    -- defines depth of tree
   recursion_level : integer range 0 .. MAX_RECURSIONS := 0;

   procedure TRAVERSE (T:POINTER) is
   begin
      if recursion_level >= MAX_RECURSIONS then
         do_some_processing;
      else
         recursion_level := recursion_level + 1;
         if T /= null then
            TRAVERSE (T.LEFT);
            PROCESS  (T);
            TRAVERSE (T.RIGHT);
         end if;
         recursion_level := recursion_level - 1;
      end if;
   end TRAVERSE;

end TRAVERSE_PACKAGE;
```

Section 7

INTRODUCING UNKNOWN NUMBERS OF TASKS

The initial restrictions outlined in this report prohibited the use of tasking of any kind (with the exception of the environment task). This restriction was softened in Section 3, in which a fixed number of tasks is permitted, thus continuing to prohibit the use of non-static arrays of tasks, tasks created by allocator evaluation, and tasks defined in iterative or recursive subprograms. Subsequent sections lifted restrictions on the use of non-static arrays, allocators, and recursion for non-task types. This section lifts those restrictions for task types as well.

7.1 Restriction Waived

Tasks that are components or subcomponents of arrays whose size cannot be determined statically are permitted. Task objects or objects with task-type subcomponents that are created by the evaluation of allocators are permitted. Tasks defined in iterative or recursive subprograms are permitted.

7.2 Analysis

The maximum number of tasks that will be created (not the number that will be simultaneously active) is determined. This maximum is determined analytically or by the imposition of programmer limitations as described for non-task types and objects in the preceding sections of this report.

Once the maximum is established, the program is transformed into a test program that creates that number of tasks and manipulates that program so that all feasible active-subprogram combinations of each task are attained simultaneously with all feasible active-subprogram combinations of all other tasks.

The number of combinations that results from this approach will generally be intractable, so safe use of an unpredictable or difficult-to-predict number of tasks will generally require the ability to apply one of the implementation-dependent simplifications described below.

7.3 Implementation-dependent Simplifications

Analysis can be simplified based on a knowledge of task-stack and task-control-block recycling for a particular implementation. For example, some implementations will not recycle task storage at all. Others will recycle storage only after exiting the frame in which the task type is declared. Still others will recycle some or all task storage when direct visibility to a task object is lost, even while the task type is still within scope.

If the implementation's task storage recycle strategy is known, consider the maximum number of tasks whose storage will be simultaneously allocated rather than the number of tasks that will be created during the life of the program. This is analagous to the analysis of the maximum net number of allocations of a given access type as discussed in Section 4.5. The difficulty of such an analysis is the potential for the existence of unknown or difficult-to-predict transient conditions which may effect the temporal characteristics of the tasks in the program. For example, a given task may have multiple paths within it that may be taken resulting in a longer or shorter duration for the task. Further, a task could be delayed indeterminately while awaiting input data from an input/output device. For determinable transient conditions, an appropriate analysis must be worst-case oriented. For indeterminate transient conditions, it would be advisable to apply hard-coded limitations (such as timed entry calls) or to allow for conditions that are orders-of-magnitude worse than conditions that are actually anticipated. In any case, such situations should be isolated to non-critical programs to minimize risk, and must be thoroughly documented.

If the compiler enforces limits imposed by STORAGE_SIZE representation clauses for task types, then a limit can be imposed for each task type and one task in each type can be tested individually. This greatly reduces the risks discussed above and eases testing requirements. In addition to the individual tests, there should be an integrated test to validate that sufficient storage exists for the established maximum number of task stacks of the sizes specified.

If the implementation can be instrumented to determine the amount of storage in use by a task at a given point, one task in each task type can be tested individually to determine the active-subprogram combination at which the task's storage usage peaks. This value can then be used in the determination of STORAGE_SIZE representation clauses if they are supported by the implementation. Based on these values, the established maximum number of tasks can be created and tested together with each task at its point of maximum storage usage.

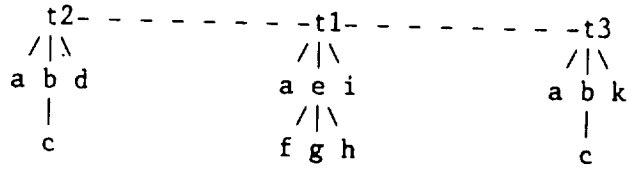
7.4 Application-specific Simplifications

None.

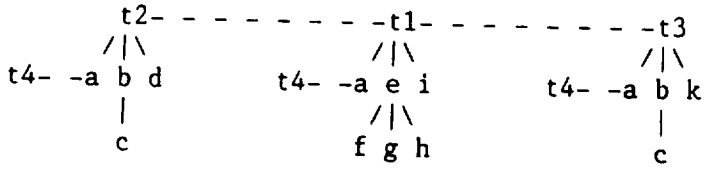
7.5 Case Study

The maximum total number of tasks that will exist during the life of a program is determinable within the guidelines presented in this report. The discussions of the preceding sections present the analysis and testing required to quantify iterative subprogram calls, non-static composite objects, designated object allocation, and recursive subprogram calls. Because these are the mechanisms that can be used for introducing an unknown number of tasks, applying the same analysis and testing to tasks created through subprogram calls, non-static arrays, designated object allocations, and recursive programs should, when combined with the fixed-task guidelines of Section 3, provide the ability to determine the maximum quantity of tasks for a given program. This knowledge is sufficient if the program is small and the worst-case depth can be analyzed.

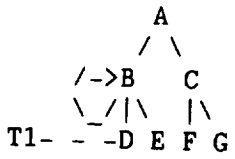
For example, Section 3.5 presented a program skeleton combining three tasks defined at the library unit level resulting in the following calling graph:

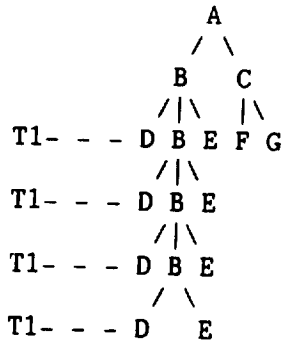


This program models the daily routine of some household and includes three tasks objects labeled T1 (task DAILY_ROUTINE), T2 (task GO_SHOPPING) and T3 (task GO_TO_WORK). We now add a task T4 (LISTEN_TO_RADIO). This task is not created at the library level, but rather is created any time that a call is made to subprogram A (procedure SELECT_CLOTHING). The new calling graph is:



Further, tasks created within a recursive subprogram can be quantified by bounding the recursion and transforming the calling graph from a cyclic graph to an acyclic graph, as described in Section 6; the following graphs are equivalent if the maximum depth of recursion is bound at 4:





Similarly, the guidelines presented for the determination of storage management requirements for designated objects and non-static composite objects in Sections 4 and 5 apply when the type in question is a task type. For example, the maximum number of task objects that will be created by a call to the following procedure is determinable:

```
procedure unknown (n : in some_number) is
   task type t1;
   task_array : array (1..n) of t1;
begin
   ...
```

The maximum of task objects of type t1 is a function of the length of array task_array, and so is known to be no more than the highest possible value of parameter n, or some_number'last. This principle also applies to the following example of task objects created as designated objects:

```
procedure unknown (n : in some_number) is
   task type t1;
   type access_t1 is access t1;
   new_t1 : access_t1;
begin
   for i in 1..n loop
       new_t1 := new t1;
   end loop;
end unknown;
```

As in the preceding example, the maximum number of tasks that will be created by a call to this procedure is some_number'last.

Although these analysis techniques (and others from the preceding sections) are adequate for determining the maximum number of tasks that will be created during the life of a program, they are insufficient for a complete analysis of storage management requirements unless it is assumed that storage that is allocated is never reclaimed and that the execution of the program is finite. Unfortunately, many practical applications cannot make these assumptions: the duration of program execution may be infinite (or as long as power is applied to the system) or may contain too many total tasks than can be accommodated by available storage in the absence of storage reclamation.

Therefore, practical analysis and testing of such applications will generally require foreknowledge of the task storage reclamation process employed by the run-time system or the availability of STORAGE_SIZE representation clauses for tasks types.

With this knowledge, the analysis and testing methods described previously should be sufficient to demonstrate adequate storage capacity.

Appendix A

TASKING EXAMPLE

This appendix presents an example program called REFORMAT in two versions. The first version is written using Ada tasking, while the second version is a purely sequential Ada program. Both versions are written to the identical specification, provided below. A high-level analysis of storage management requirements for both versions is also provided.

A.1 Specification

Program REFORMAT will read an input file in one format and write an output file in another format, as follows:

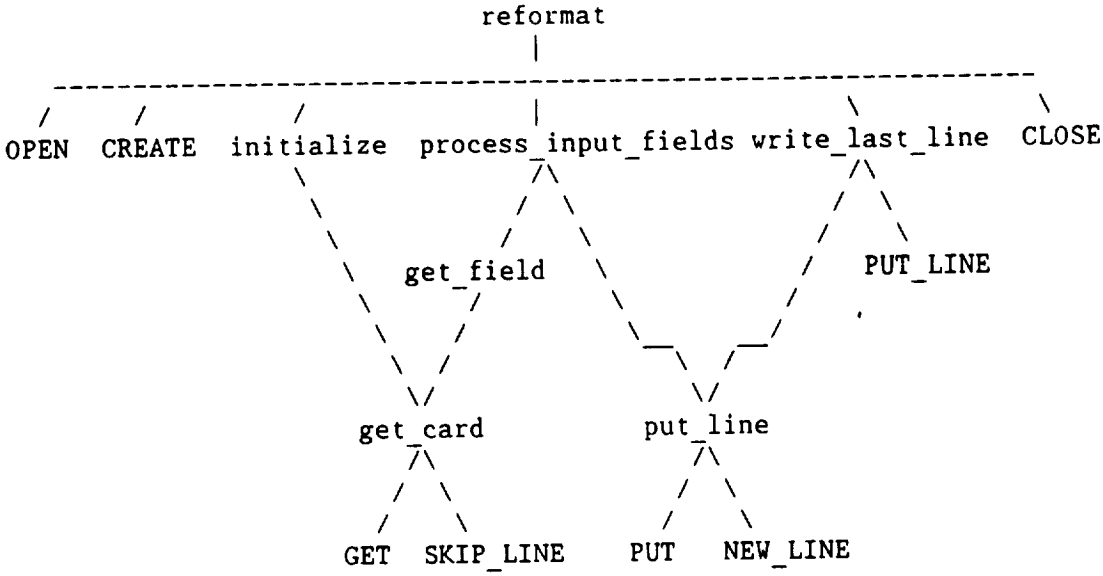
Columns 1-72 of the input file contain twelve six-character fields. Columns 73-80 of each line are empty. The last card contains XXXXXX in the last full field and spaces afterward. The sequence of fields in the input file is to be copied to the output file, but with consecutive occurrences of the same six-character field replaced by a single occurrence. In the output file, fields are to be printed 15 per line, with fields on the same line separated by two spaces.

The name of the input file is DATA.OLD, and the name of the output file is DATA.NEW.

A.2 Analysis of Storage Management Requirements

The storage requirements for both versions of REFORMAT are determinant. The non-tasking version adheres to the guidelines of Section 2: there are no instances of recursion, composite objects with non-static bounds, designated variables, or tasks. Therefore, the program can be tested for adequate storage by constructing the calling graph (below) and exercising each path

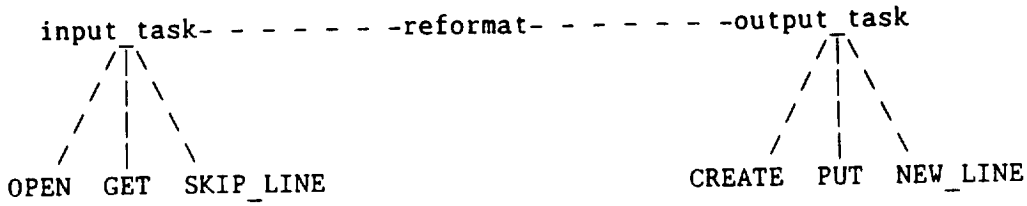
within it. (In this calling graph, subprograms from the pre-defined package TEXT_IO are highlighted with capitalized identifiers.)




```
TEST PATHS:
   reformat--> OPEN
   reformat--> CREATE
   reformat--> initialize--> get_card--> GET
   reformat--> initialize--> get_card--> SKIP_LINE
   reformat--> process_input_fields
   reformat--> process_input_fields--> get_field--> get_card--> GET
   reformat--> process_input_fields--> get_field--> get_card--> SKIP_LINE
   reformat--> process_input_fields--> put_line--> PUT
   reformat--> process_input_fields--> put_line--> NEW_LINE
   reformat--> write_last_line--> put_line--> PUT
   reformat--> write_last_line--> put_line--> NEW_LINE
   reformat--> write_last_line--> PUT_LINE
   reformat--> CLOSE
```

In practice, a single test case can be constructed to test all of these paths.

The tasking version of REFORMAT makes use of no subprograms at all except those of the pre-defined package TEXT_IO, which are again highlighted in the calling graph as capitalized identifiers:



Based on the analysis described in Section 3, all feasible active-subprogram combinations of each task should be tested simultaneously with all active-subprogram combinations of each other task. In this case, such a test would require modification to the TEXT_IO package routines to allow them to lock into the dummy task FREEZE_CALLER. Because modification of a predefined package for test purposes may not be possible, an alternative method must be chosen (this would not be a typical problem with most mission-critical program because embedded systems with critical storage restrictions will normally supply their own customized input/output packages.) The preferred approach is the use of STORAGE_SIZE representation clauses, which allow the independent testing of each individual task.

It is interesting to note the nature of the differences between the two REFORMAT versions in terms of dynamic storage management requirements. The non-tasking version will require very simple storage management: a single subprogram stack is all that is needed. Alternatively, with no recursion or multi-tasking, all storage could be allocated pre-runtime. The tasking version requires somewhat more sophisticated storage management, but still is deterministic. Again, there is no recursion, so each of the three tasks are bounded in their storage requirements. A fixed size stack for each task will cover the dynamic storage management requirements of the proram.

In terms of the total storage needs, it is difficult to compare the two programs without knowledge of the underlying implementation. The non-tasking version requires additional variable and type declarations, while the tasking version requires additional support for the task declarations. To choose the version that is the least storage-intensive, an analysis must be conducted based on the implementation-dependent storage requirements for subprograms, objects, type definitions, and tasks. Depending on the imlementation in quesstion, either the tasking or the non-tasking version may be found to be more storage-efficient.

A.3 Tasking Version of Reformat

```
procedure Reformat is

   subtype Field_Subtype is string (1 .. 6);

   Previous_Field : Field_Subtype;
   This_Field     : Field_Subtype;
   Final_Field    : constant Field_Subtype := "XXXXXX";

   task Input_Task is
      entry Get_Field (Field: out Field_Subtype);
   end Input_Task;

   task Output_Task is
      entry Put_Field (Field: in Field_Subtype);
   end Output_Task;

   task body Input_Task  is separate;
   task body Output_Task is separate;

begin

   Input_Task.Get_Field (Previous_Field);

   loop
      loop
         Input_Task.Get_Field (This_Field);
         exit when This_Field /= Previous_Field;
      end loop;
      Output_Task.Put_Field (Previous_Field);
      Previous_Field := This_Field;
      exit when Previous_Field = Final_Field;
   end loop;

   Output_Task.Put_Field (Final_Field);

end Reformat;
```

```
with Text_Io; use Text_Io;
separate (Reformat)
task body Input_Task is

   Input_File : File_Type;
   Next_Field : Field_Subtype;

begin
   Open (Input_File, In_File, "DATA.OLD");

   Main_Loop:
      loop
         for I in 1 .. 12 loop
            Get (Input_File, Next_Field);
            accept Get_Field (Field: out Field_Subtype) do
               Field := Next_Field;
            end Get_Field;
            exit Main_Loop when Next_Field = Final_Field;
         end loop;
         Skip_Line (Input_FIle);
      end loop Main_Loop;
   Close (Input_File);
end Input_Task;

with Text_Io; use Text_Io;
separate (Reformat)
task body Output_Task is

   Output_File : File_Type;
   Next_Field  : Field_Subtype;

begin

   Create (Output_File, Out_File, "DATA.OLD");
   Main_Loop:
      loop
         for I in 1 .. 15 loop
            accept Put_Field (Field: in Field_Subtype) do
               Next_Field := Field;
            end Put_Field;
            Put (Output_File, Next_Field);
            if I < 15 then
               put ("  ");
            end if;
            exit Main_Loop when Next_Field = Final_Field;
         end loop;
         New_Line (Output_File);
      end loop Main_Loop;
   Close (Output_File);
end Output_Task;
```

A.4 Non-Tasking Version of Reformat

```
with Text_Io; use Text_Io;

procedure Reformat is

   subtype Field    is string(1 .. 6);

   type Card_Image  is array(1 .. 12) of Field;
   type Line_Image  is array(1 .. 15) of Field;

   Input_File         : File_Type;
   Output_File        : File_Type;
   Input_Image        : Card_Image;
   Output_Image       : Line_Image;
   Previous_Field     : Field;
   This_Field         : Field;
   Final_Field        : constant Field := "XXXXXX";
   Input_Field_Number : integer range 1 .. 12 := 2;
   Output_Field_Number: integer range 1 .. 15 := 1;

   procedure Get_Card (File: in File_Type; Image: out Card_Image)
             is separate;

   procedure Put_Line (File: in File_Type; Image: in Line_Image)
             is separate;

   procedure Initialize             is separate;
   procedure Get_Field              is separate;
   procedure Process_Input_Fields   is separate;
   procedure Write_Last_Line        is separate;

begin

   Open   (Input_File,  In_File,  "DATA.OLD");
   Create (Output_File, Out_File, "DATA.NEW");

   Initialize;
   Process_Input_Fields;
   Write_Last_Line;

   Close (Input_File);
   Close (Output_File);

end Reformat;
```

```
separate (Reformat)
procedure Get_Card (File: in File_Type; Image: out Card_Image) is
begin
   for I in 1 .. 12 loop
      Get (File, Image(I));
   end loop;
   Skip_Line(File);
exception
   when others => null;
end Get_Card;


separate (Reformat)
procedure Put_Line (File: in File_Type; Image: in Line_Image) is
begin
   for I in 1 .. 15 loop
      Put (File, Image(I));
      Put (File, "  ");
   end loop;
   New_Line (File);
end Put_Line;


separate (Reformat)
procedure Initialize is
begin
   Get_Card (Input_File, Input_Image);
   Previous_Field := Input_Image(1);
end Initialize;


separate (Reformat)
procedure Get_Field is
begin
   loop
      This_Field := Input_Image (Input_Field_Number);

      if Input_Field_Number /= 12 then
         Input_Field_Number := Input_Field_Number + 1;
      else
         Input_Field_Number := 1;
         Get_Card (Input_File, Input_Image);
      end if;
      exit when This_Field /= Previous_Field;
   end loop;
end Get_Field;
```

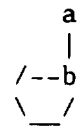
```
separate (Reformat)
procedure Process_Input_Fields is
begin

  loop
      Get_Field;

      Output_Image (Output_Field_Number) := Previous_Field;

      if Output_Field_Number /= 15 then
         Output_Field_Number := Output_Field_Number + 1;
      else
         Put_Line (Output_File, Output_Image);
         Output_Field_Number := 1;
      end if;

      Previous_Field := This_Field;

      exit when Previous_Field = Final_Field;

  end loop;

end Process_Input_Fields;


separate (Reformat)
procedure Write_Last_Line is
begin

   Output_Image (Output_Field_Number) := Previous_Field;

   for I in Output_Field_Number + 1 .. 15 loop
      Output_Image(I) := "      ";
   end loop;

   Put_Line (Output_File, Output_Image);

end Write_Last_Line;
```

Appendix B

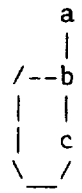
STORAGE MANAGEMENT RISKS FOR Ada

This appendix summarizes the various aspects of Ada usage resulting in dynamic storage management requirements. Calling graph figures and code fragments are used to depict each aspect. It is intended as a highly simplified reference source to be used as a companion to this report.

1. Direct Recursion



2. Mutual Recursion



3. Objects with Non-Static Bounds

```
procedure unknown (n : in integer) is
   an_array : array (1..n) of integer;
begin
   ...


procedure unknown (n : in positive) is
   type stack (n: integer) is
      record
         s: array (1..n) of integer;
         top: integer := 0;
      end record;
begin
   ...
```

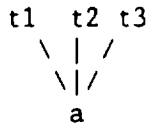
4. Parameters with Non-Static Bounds

```
procedure main is

   type arbitrary is array (positive range <>) of integer;

   procedure test (arb: arbitrary) is
   ...
begin
...
end main;


function concat (a, b: string) return string is
begin
   return a & b;
end concat;
```

5. Unknown Number of Designated Objects

```
procedure alloc (n: integer) is

   type big_rec is
      record
         a: integer;
         b: string (1..5);
      end record;
   type big_rec_access is access big_rec;
   new_big_rec: big_rec_access;

begin
   for i in 1 .. n loop
      new_big_rec := new big_rec;
   end loop;
end alloc;


procedure alloc (n: integer) is

   type big_rec is
      record
         a: integer;
         b: string (1..5);
      end record;
   type big_rec_access is access big_rec;

   new_big_rec: array (1..n) of big_rec_access;

begin
   for i in 1 .. n loop
      new_big_rec(i) := new big_rec;
   end loop;
end alloc;
```

6. Multiple Simultaneous Invocations of a Subprogram



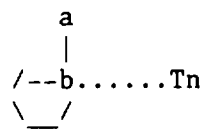
7. Variable Array of Task Objects

```
procedure unknown (n : in integer) is
   task type t1;
   task_array : array (1..n) of t1;
begin
   ...
```

8. Task Object Declared in Variable Loop

```
procedure unknown (n : in integer) is
   task type t1;
   type access_t1 is access t1;
   new_t1 : access_t1;
begin
   for i in 1..n loop
      new_t1 := new t1;
   end loop;
end unknown;
```

9. Task Object Declared in Recursive Subprogram



Appendix C

REFERENCES

1. [AUTY] "Report on Pragma Static", D. Auty, SofTech KIT 1141, SofTech, Alexandria, VA, October 1984.

2. [HOROWITZ] Fundamentals of Data Structures, E. Horowitz and S. Sahni, Computer Science Press, Rockville, MD, 1982.

3. [LRM] "Ada Programming Language", ANSI/MIL-STD-1815A, January 1983.

4. [RAL] "Dynamic Storage Management in Ada: Relevant Aspects of the Language", Volume II, September 1987.

5. [RvsM] "Dynamic Storage Management in Ada: Requirements of the Language Versus Manifestations of Current Implementations", Volume III, September 1987.