RSRE

MEMORANDUM No. 4244

ROYAL SIGNALS & RADAR ESTABLISHMENT

A MAPPING FOR CONVERTING MASCOT DESIGNS TO ADA PROGRAMS

Author: T Dean

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

CONDITIONS OF RELEASE BR-109214

0032584

..... U

..... Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4244

A Mapping for Converting Mascot designs to Ada Programs

Timothy Dean

19 July 1988

Abstract

The Future Networks section of CC1 division uses Mascot III in designing parallel systems. These designs are then implemented in the Ada programming language. A systematic method for mapping Mascot designs into Ada programs has been developed, and this document describes the main characteristics of the method. In most applications, the mapping in its basic form is entirely adequate, and so a large part of the document is spent in describing this. However, certain situations present problems which can only be solved by making some alterations. Therefore a short section on these problems and their solutions is also included.

The method was chosen since it produces Ada programs with components corresponding directly to the Mascot design. This was considered a big advantage in producing maintainable and clear code. Mascot Templates (both algorithmic and structural) are represented as generic packages, and Mascot components are instantiations of those packages.

Contents

1 Introduction

Mascot is a software engineering methodology which is particularly applicable to real-time systems. It also embraces parts of the software life-cycle (eg testing) which other methodologies do not reach. These two features have contributed to Mascot's popularity and wide use.

For this reason, a study was commissioned by the Ministry of Defence into possible mappings between Mascot III and the Ada programming language [3]. The study suggested nine possible ways in which the Mascot design philosophy could be modelled in Ada. This paper describes a method most closely related to method seven of that study. The method in its basic form falls down when used in certain systems, and so some possible extensions and alterations have also been suggested.

The mapping does not proscribe any of the real-time Ada facilities, but it promotes a more systematic approach to their use. A detailed knowledge of the basics of Mascot III is assumed, together with an understanding of the Ada programming language.

It is hoped that this document will achieve two aims. First, it is intended as a manual for people trying to learn the method for the first time. Therefore, a large number of examples have been included illustrating various points. Secondly, a few comments have been made assessing the strengths and weaknesses of the method.

A variety of experiments using the method were performed by the Future Networks section of CC1 Division. The DEC Ada compiler was used [2] running on DEC's VMS Operating System for all the experiments. Some very simple systems were implemented, eg in demonstrating the famous *Triplex* example described in the Mascot Handbook, (see [1]). The DoD Internet Protocol [4] was also implemented using the method, to demonstrate its feasibility in large systems.

2 The Mapping from Ada to Mascot III

2.1 Overall Strategy

Each Mascot object (IDA, Pool, Channel, Activity, Subsystem) is treated as an Ada package. Mascot templates are generic packages, Mascot instances are instantiations of those packages. Each package is separately compiled and inserted into an Ada library. This promotes software modularization and reusability.

2.2 Mechanism for window implementation

A Mascot access interface consists of a *list of procedures*. Once an access interface has been defined, it acts in the same way as a type declaration in a programming language. A window of a particular access interface type provides the procedures defined by that interface.

Each window is an 'inner' package specification within the generic package specification. Consider the simple channel below:

```
ACCESS INTERFACE GET-INT;
    PROCEDURE GET (I : OUT INTEGER);
    - HOLDS IF THE CHANNEL IS EMPTY
    PROCEDURE CLEAR;
    - EMPTIES THE CHANNEL
END.

ACCESS INTERFACE PUT-INT;
    PROCEDURE PUT (I : IN INTEGER);
    - HOLDS IF THE CHANNEL IS FULL
    PROCEDURE CLEAR;
END.

CHANNEL SIMPLE-CHAN-TEMP;
  CONSTANT MAX-CHAN-SIZE : POSITIVE;
  - THE MAXIMUM NUMBER OF INTEGERS WHICH
  - CAN BE STORED IN THE CHANNEL.
    PROVIDES GW : GET-INT;
    PROVIDES PW : PUT-INT;
END.
```

The channel possess two windows, pw and gw, one of type put_int and the other of type get_int. This is turned into Ada as follows:

```
GENERIC
    max_chan_size : positive;
PACKAGE simple_chan_temp IS
```

```
    PACKAGE gw IS
        PROCEDURE get (i : OUT integer);
    -- holds if the channel is empty
        PROCEDURE clear;
    -- empties the channel
    END;

    PACKAGE pw IS
        PROCEDURE put (i : in integer);
        -- holds if the channel is full
        PROCEDURE clear;
    END;
END simple_chan_temp;
```

A useful naming convention is to use template names as generic package names, and the window names as the internal package names. The procedure headings are copied directly from the Mascot access interface.

2.3 Mechanism for port implementation

A port of a particular access interface type can call any of the procedures defined by that interface. A port and a window can only be connected if they are of the same access interface type. A Mascot template has ports which are not yet connected to windows. It is only when components are created from that template that the connection between ports and windows is established.

It was decided that the generic formal procedures of a package were the best method of mapping ports to Ada. Each access interface procedure becomes a formal parameter of the generic package. The problem of identical procedure names in different ports is avoided by prefixing each procedure name with its port name and an underscore. Consider the following activity:

```
ACTIVITY ACT-TEMP;
    REQUIRES PUTP : PUT-INT;
    REQUIRES GETP : GET-INT;
.
.
END.
```

Such an activity is turned into Ada as follows:

```
GENERIC
   WITH PROCEDURE putp_put (i : IN integer);
```

```
   WITH PROCEDURE putp_clear;
   WITH PROCEDURE getp_get (i : OUT integer);
   WITH PROCEDURE getp_clear;
PACKAGE act_temp IS
PRIVATE
   -- the purpose of this part is purely aesthetic.
   -- It means that ports can be referenced by the
   -- usual Mascot dot notation within the activity body.
   PACKAGE putp IS
       PROCEDURE put (i : IN integer) RENAMES putp_put;
       PROCEDURE clear RENAMES putp_clear;
   END putp;
   PACKAGE getp IS
       PROCEDURE get (i : OUT integer) RENAMES getp_get;
       PROCEDURE clear RENAMES getp_clear;
   END getp;
END act_temp;

PACKAGE BODY act_temp IS

    PROCEDURE act_proc IS
    BEGIN
        -- code for the activity
    END;

    TASK activity_task;
    TASK BODY activity_task IS
    BEGIN
        act_proc;
    END;
END act_temp;
```

The private part of the package can be used to convert the procedure names to their original form for use in the package body.

2.4 Connecting ports to windows

A particular port is connected to a window by passing the window procedures as actual parameters when instantiating the generic package which contains the port. As a simple example, suppose an activity generates a stream of integers to be passed to a second activity. This is achieved by the use of a simple channel as above, and the Mascot textual notation is outlined as follows:

```
ACTIVITY ACT-TEMP1;
```

```
   REQUIRES PP : PUT-INT;

END.

ACTIVITY ACT-TEMP2;
   REQUIRES GP : GET-INT;

END.

SYSTEM TEST;
   USES SIMPLE-CHAN-TEMP, ACT-TEMP1, ACT-TEMP2;
   CHANNEL INTEGER-CHAN1 : SIMPLE-CHAN-TEMP
      (MAX-CHAN-SIZE = 100);
   ACTIVITY A1 : ACT-TEMP1 (PP = INTEGER-CHAN1.PW);
   ACTIVITY A2 : ACT-TEMP2 (GP = INTEGER-CHAN1.GW);

END.
```

The above is turned into Ada as follows:

```
GENERIC
   WITH PROCEDURE pp_put (i : IN integer);
   WITH PROCEDURE pp_clear;
PACKAGE act_temp1 IS
PRIVATE
   PACKAGE pp IS
       PROCEDURE put (i : IN integer) RENAMES pp_put;
       PROCEDURE clear RENAMES pp_clear;
   END pp;
END act_temp1;

PACKAGE BODY act_temp1 IS

    PROCEDURE act_proc IS
    BEGIN
        -- code for the activity
    END;

    TASK activity_task;
    TASK BODY activity_task IS
    BEGIN
        act_proc;
    END;
END act_temp1;
```

```
GENERIC
   WITH PROCEDURE gp_get (i : OUT integer);
   WITH PROCEDURE gp_clear;
PACKAGE act_temp2 IS
PRIVATE
   PACKAGE gp IS
       PROCEDURE get (i : OUT integer) RENAMES gp_get;
       PROCEDURE clear RENAMES gp_clear;
   END gp;
END act_temp2;

PACKAGE BODY act_temp2 IS

    PROCEDURE act_proc IS
    BEGIN
        -- code for the activity
    END;

    TASK activity_task;
    TASK BODY activity_task IS
    BEGIN
        act_proc;
    END;
END act_temp2;


WITH simple_chan_temp, act_temp1, act_temp2;
PROCEDURE test IS

   PACKAGE integer_chan1 IS NEW simple_chan_temp (
       max_chan_size => 100);
   PACKAGE a1 IS NEW act_temp1 (
       pp_put => integer_chan1.pw.put,
       pp_clear => integer_chan1.pw.clear);
   PACKAGE a2 IS NEW act_temp2 (
       gp_get => integer_chan1.gw.get,
       gp_clear => integer_chan1.gw.clear);

BEGIN
    null;
END;
```

2.5 Subsystems

Subsystems are a new and powerful idea introduced in Mascot III. They enable complex systems to be broken down into more manageable parts, while hiding the clutter of small details. The ports of a subsystem are connected to the ports of internal subsystems, or activities. Likewise with windows.

A simple example is the best way to illustrate how to write a subsystem. The subsystem possesses one port and one window. The textual notation is as follows:

```
ACTIVITY DUP-TEMP;
   REQUIRES GP : GET-INT;
   REQUIRES PP : PUT-INT;

END.

SUBSYSTEM SIMPLE-SUB-TEMP;
  REQUIRES GP : GET-INT;
  PROVIDES GW : GET-INT;
  USES SIMPLE-CHAN-TEMP, DUP-TEMP;

  CHANNEL CH : SIMPLE-CHAN-TEMP
     (MAX-CHAN-SIZE = 100);
  ACTIVITY DUP : DUP-TEMP (
      PP = CH.PW,
      GP =GP);
  GW = CH.GW;
END.
```

The Ada for the above is:

```
GENERIC
   -- the port declaration
   WITH PROCEDURE gp_get (i : OUT integer);
   WITH PROCEDURE gp_clear;
PACKAGE simple_sub_temp IS
    -- the window declaration
    PACKAGE gw IS
        PROCEDURE get (i : OUT integer);
        PROCEDURE clear;
    END;
PRIVATE
   -- the port renaming
   PACKAGE gp IS
       PROCEDURE get (i : OUT integer) RENAMES gp_get;
```

```
        PROCEDURE clear RENAMES gp_clear;
    END gp;
END simple_sub_temp;

WITH simple_chan_temp, dup_temp;
PACKAGE BODY simple_sub_temp IS

    PACKAGE ch IS NEW simple_chan_temp (max_chan_size => 100);
    PACKAGE dup IS NEW act_temp1 (
        pp_put => ch.pw.put,
        pp_clear => ch.pw.clear,
        gp_get => gp.get,
        gp_clear => gp.clear);

    PACKAGE BODY gw IS
        PROCEDURE get (i : OUT integer) IS
        BEGIN
           ch.get (i);
        END;

        PROCEDURE clear IS
        BEGIN
            ch.clear;
        END;
    END;
END simple_sub_temp;
```

2.6 Writing the code for Mascot objects

The most difficult part of defining the mapping from Mascot to Ada is the code-structure of primitive objects. There is no prescriptive method of generating the code, but it is left to the judgment of the implementor. The following paragraphs are intended as guidelines.

2.6.1 The code for activities

As a general principle it has been found that activities are best implemented as single tasks. In order that the possibility of deadlock is minimized, activity tasks should have no entries. Rather, all synchronization should take place inside the IDAs, in keeping with the philosophy of Mascot.

For a number of reasons, waiting for events by continuous polling is bad programming practice. Therefore, activities should be designed to avoid pol'ing wherever possible. (The Mascot design can often indicate activities which are in danger of needing to poll: the arrow-heads on paths indicate which way the data is flowing. Therefore, if more than one arrow-head converges on an activity, it may be that the activity will have to poll those paths for data.) There are two possible ways to avoid having to poll. With careful design,

it may be possible to redesign the system such that the two data flows can be converged before reaching the activity. Alternatively, the system can be treated as a Finite-State Machine. In this way the *order* in which the data will arrive down the multiple paths is known in advance. Thus polling is not required.

2.6.2 The code for IDAs

IDAs can be coded in a variety of ways:

- Most Ada implementations allocate a separate stack for each task. When procedures are activated, space for all local variables is allocated on this stack. Thus it is possible for two tasks to execute a common procedure concurrently *provided the procedure does not update a non-dynamic variable*, (i.e. one which is not created at execution-time on the stack). If an IDA has no internal variables which are changed by access procedures, it will need no protection mechanism. This type of IDA is typically used to initialize read-only variables used by more than one activity.
- Most IDAs have internal variables or state information which is updated by the access procedures. In these IDAs, concurrent access must be prevented. This can be achieved in a variety of ways:
  - Each variable may have its own guard. Ada tasks can be written which implement semaphores for this purpose. If more than one process accesses a variable, the VMS lock management services are useful.
  - The IDA as a whole can be protected. Although this is not as efficient as the previous method, the code is much simpler to write and debug. It is suggested that this method be used in preference to the above, except where speed is absolutely essential.
  - The most elegant solution is to encapsulate the whole IDA as a task. Each access procedure becomes an entry to the task. The great strength of this method is that the rich set of Ada rendezvous constructs can be used (eg conditional or timed entry calls). Access interface procedures are made very much more flexible if for example, a time-out parameter can be supplied. This produces very efficient code in which activities are automatically suspended until re-scheduled. An example of this can be found in the channel template, (see below).

2.7 Some examples

2.7.1 Two activities

The first activity generates a stream of characters. It puts these characters to a port.

```
GENERIC
    WITH PROCEDURE pp_put (ch : IN character;
        wait_on_full : boolean := true);
    WITH PROCEDURE pp_clear;
```

```
PACKAGE gen_data_temp IS
PRIVATE
    PACKAGE pp IS
        PROCEDURE put (ch : IN character;
            wait_on_full : boolean := true) RENAMES pp_put;
        PROCEDURE clear RENAMES pp_clear;
    END pp;
END gen_data_temp;

PACKAGE BODY gen_data_temp IS
    PROCEDURE task_code IS
    BEGIN
        LOOP
            FOR i IN 'A'..'Z' LOOP
                pp.put (i);
            END LOOP;
        END LOOP;
    END task_code;

    TASK activity_task;
    TASK BODY activity_task IS
    BEGIN
        task_code;
    END activity_task;
END gen_data_temp;
```

The second activity reads characters from a port and prints them out.

```
GENERIC
    WITH PROCEDURE gp_get (ch : OUT character);
    WITH PROCEDURE gp_clear;
PACKAGE print_data_temp IS
PRIVATE
    PACKAGE gp IS
        PROCEDURE get (ch : OUT character) RENAMES gp_get;
        PROCEDURE clear RENAMES gp_clear;
    END gp;
END print_data_temp;

WITH text_io;
PACKAGE BODY print_data_temp IS

    PROCEDURE task_code IS
        ch : character;
```

```
    BEGIN
        LOOP
            gp.get (ch);
            text_io.put (ch);
            text_io.new_line;
        END LOOP;
    END task_code;

    TASK activity_task;
    TASK BODY activity_task IS
    BEGIN
        task_code;
    END activity_task;
END print_data_temp;
```

2.7.2 A channel

The following channel has two windows. One of type put_int and the other OF type get_int.

```
GENERIC
    type elem_type IS PRIVATE;
    max_chan_size : positive;
PACKAGE gen_chan_temp IS
    PACKAGE gw IS
        PROCEDURE get (e : OUT elem_type);
        -- holds if the channel IS empty
        PROCEDURE clear;
        -- empties the channel
    END;

    PACKAGE pw IS
        PROCEDURE put (
            e : IN elem_type;
            wait_on_full : boolean := true);
            -- holds if the channel is full
        PROCEDURE clear;
    END;
END gen_chan_temp;


PACKAGE BODY gen_chan_temp IS

TASK ida_task IS
    ENTRY get_entry (e : OUT elem_type);
    ENTRY put_entry (e : IN elem_type);
```

```
    ENTRY clear_entry;
END ida_task;

TASK BODY ida_task IS
    buffer : ARRAY (1..max_chan_size) OF elem_type;
    next_read, next_written : integer := 1;
    current_size : integer := 0;

    PROCEDURE get (e : OUT elem_type) IS
    BEGIN
        e := buffer (next_read);
        next_read := (next_read mod max_chan_size) + 1;
        current_size := current_size - 1;
    END;

    PROCEDURE put (e : IN elem_type) IS
    BEGIN
        buffer (next_written) := e;
        next_written := (next_written mod max_chan_size) + 1;
        current_size := current_size + 1;
    END;

    PROCEDURE clear IS
    BEGIN
        current_size := 0;
        next_written := next_read;
    END;

BEGIN
    LOOP
        SELECT
            WHEN current_size /= 0 =>
                ACCEPT get_entry (e : OUT elem_type) DO
                    get (e);
                END;
        OR
            WHEN current_size < integer (max_chan_size) =>
                ACCEPT put_entry (e : IN elem_type) DO
                    put (e);
                END;
        OR
            ACCEPT clear_entry DO
                clear;
            END;
        END SELECT;
```

```
    END LOOP;
END ida_task;


PACKAGE BODY gw IS
    PROCEDURE get (e : OUT elem_type) IS
    BEGIN
        ida_task.get_entry (e);
    END;

    PROCEDURE clear IS
    BEGIN
        ida_task.clear_entry;
    END;
END;

PACKAGE BODY pw IS
    PROCEDURE put (
        e : IN elem_type;
        wait_on_full : boolean := true) IS
    BEGIN
        IF wait_on_full THEN
            ida_task.put_entry (e);
        ELSE
            SELECT
                ida_task.put_entry (e);
            ELSE
                null;
            END SELECT;
        END IF;
    END;

    PROCEDURE clear IS
    BEGIN
        ida_task.clear_entry;
    END;
END;
```

2.7.3 A simple system

The next example demonstrates how all the above can be combined into a real system.

```
WITH gen_chan_temp, print_data_temp, gen_data_temp;
PROCEDURE test IS
```

```
    PACKAGE ch1 IS NEW gen_chan_temp (
        elem_type => character,
        max_chan_size => 17);

    PACKAGE gen_data IS NEW gen_data_temp
         (pp_put => ch1.pw.put,
         pp_clear => ch1.pw.clear);

    PACKAGE print_data IS NEW print_data_temp
         (gp_get => ch1.gw.get,
         gp_clear => ch1.gw.clear);

BEGIN
    null;
END;
```

2.7.4 A simple subsystem

This example is a simple duplex subsystem. The subsystem has one window of type put and two windows of type get. Everything inserted into the put window is duplicated in both get windows:

```
GENERIC
    type elem_type IS PRIVATE;
    max_chan_size : positive;
PACKAGE duplex_temp IS

    PACKAGE gw1 IS
        PROCEDURE get (e : OUT elem_type);
        PROCEDURE clear;
    END;

    PACKAGE gw2 IS
        PROCEDURE get (e : OUT elem_type);
        PROCEDURE clear;
    END;

    PACKAGE pw IS
        PROCEDURE put (
            e : IN elem_type;
            wait_on_full : boolean := true);
        PROCEDURE clear;
    END;
END duplex_temp;
```

```
PACKAGE BODY duplex_temp IS
    PACKAGE ch IS NEW gen_chan_temp(
        elem_type => elem_type,
        max_chan_size => max_chan_size);
    PACKAGE ch1 IS NEW gen_chan_temp(
        elem_type => elem_type,
        max_chan_size => max_chan_size);
    PACKAGE ch2 IS NEW gen_chan_temp(
        elem_type => elem_type,
        max_chan_size => max_chan_size);
    PACKAGE dup_data IS NEW dup_data_temp
         (gp_get => ch.gw.get,
         gp_clear => ch.gw.clear,
         pp1_put =>  ch1.pw.put,
         pp1_clear => ch1.pw.clear,
         pp2_put => ch2.pw.put,
         pp2_clear => ch2.pw.clear);

    PACKAGE BODY gw1 IS
        PROCEDURE get (e : OUT elem_type) IS
        BEGIN
            ch1.gw.get (e);
        END;

        PROCEDURE clear IS
        BEGIN
            ch1.gw.clear;
        END;
    END;

    PACKAGE BODY gw2 IS
        PROCEDURE get (e : OUT elem_type) IS
        BEGIN
            ch2.gw.get (e);
        END;

        PROCEDURE clear IS
        BEGIN
            ch2.gw.clear;
        END;
    END;

    PACKAGE BODY pw IS
        PROCEDURE put (
            e : IN elem_type;
```

```
            wait_on_full : boolean := true) IS
        BEGIN
            ch.pw.put (e, wait_on_full);
        END;

        PROCEDURE clear IS
        BEGIN
            ch.pw.clear;
        END;
    END;
END;
```

3 Possible Enhancements to the Mapping

A number of experiments were conducted in building real systems using the method. The experiments varied in size from simple two-activity systems to a complete implementation of the Internet Protocol. The following weaknesses were found in the basic method:

- Cyclic dependancy problems.
- Calling unelaborated packages.
- Termination difficulties.
- Recompilation overheads.
- Shared data.

3.1 Cyclic dependancy problems

Ports are connected to windows by associating actual procedures with generic formal procedures. This means that windows must be declared before the ports which connect to them. This is impossible to do, however, in a system with cyclic port to window dependancies.

If this problem arises, a slight alteration to the method has to be made. At the beginning of each structural unit (subsystem or system), procedure *stubs* are declared for all window procedures. Following this, the objects making up that unit can be declared in any order. Ports of each object are tied to the stub procedures. After all the objects have been declared, the stub bodies are elaborated. Each consists of a call to the appropriate window procedure. It may seem that this results in an inefficiency, since each access procedure results in two subprogram activations, with all the stack initialisation overheads being duplicated. However, the Ada *inline* pragma can be used to overcome this problem, if speed is critical in the system.

3.2 Calling unelaborated packages

The second problem may come about as result of the solution to the first problem. When packages are elaborated, any tasks contained in the bodies are activated immediately. As a result, an activity may call an access procedure which is in an unelaborated unit. This results in a PROGRAM_ERROR exception being raised in the activity. This problem can be solved by including a special *start* procedure as a formal parameter of all activities and subsystems containing active elements. The first thing which each activity does is to call this procedure. At the system level, the procedure *start* is declared. It consists of a call to a simple semaphore which is released by the activation of the main program:

```
procedure system is

   TASK semaphore IS
      ENTRY release;
```

```
        ENTRY start;
    END;

    TASK BODY semaphore IS
    BEGIN
        ACCEPT release;
        LOOP
            ACCEPT start;
        END LOOP;
    END;

    PROCEDURE start IS
    BEGIN
        semaphore.start;
    END;

    PACKAGE subsys1 IS NEW subsys1_temp (
        start => start,
        -- other ports
        );

    PACKAGE subsys2 IS NEW subsys2_temp (
        start => start,
        -- other ports
        );

BEGIN
    semaphore.release;
END;
```

This causes all activities to 'hang' until all the packages have been elaborated.

NB this corresponds to the two-stage process described in the Mascot III handbook. *Initialisation* consists of the elaboration of each package, and *starting* consists of a trigger from the main program. (See Section 4.6 of the Mascot handbook).

3.3 Termination difficulties

When an exception occurs in an Ada task, it causes the task to terminate, (unless it is handled with an exception handler). All other tasks continue executing normally. This produces a problem in situations when it would be preferable to abort the complete system. In tasks which declare entries, the terminate alternative PROVIDES a means of aborting the task. However, activities should possess no entries if they have been properly mapped into Ada. Therefore another method must be found to cause them to terminate. As yet no satisfactory solution has been found for this problem. One possible method would

be a special *abort* procedure possessed by all objects. The outermost level of each task could then handle all exceptions by calling the abort procedure of all other objects, and then re-raising the exception. This has not been exhaustively investigated at the time of writing.

3.4 Recompilation overheads

The instantiation of generic units is an extremely slow process in most compilers. It is usually achieved by a macro-substitution of the generic entities with their equivalents and compiling the resulting source code. For large subsystems, this can result in a considerable amount of recompilation. This problem has not been looked at seriously since it is regarded as a compiler shortcoming rather than a weakness in the mapping.

3.5 Shared data

The method makes a distinction between two types of package. There are packages representing Mascot objects, and packages which declare common procedures, type definitions, constants etc. which are used by those packages. However, it is possible to declare shareable data in the second type of package, bypassing the whole Mascot method of protecting data. Severe errors may result if data is concurrently written by more than one activity. This type of data may be declared in the body of such packages, and so may not easily be spotted, but can still be concurrently accessed by more than one activity. Some of the predefined packages must be protected from this type of access. The package *text_io* is a *case in point*. It updates the cursor position when procedure *put* is called, and this may result in undesired effects if it is called concurrently.

The solution to this problem is to encapsulate these packages in an IDA. The IDA version contains all the same procedure declarations as the original package. (Note that types and constants need not be redeclared). The body of each procedure consists of setting a semaphore, calling the original procedure, and releasing the semaphore.

4 Conclusions

The method has been used with good success in a wide variety of applications. There have been a number of benefits which have resulted from its use:

- Software is better structured and easier to maintain. ('Raw' Ada tends to be completely ad hoc in its use of concurrency features.)
- The Mascot idea of active and passive design elements makes deadlock situations much easier to detect.
- A large library of reusable templates can be built up, producing much more reliable and maintainable programs.

The main disadvantage of the method is the amount of time taken to manually code object templates. Each access interface procedure has to be duplicated four or five times in order to link a port to a window. For an access interface of ten or more procedures, this

can be an extremely time-consuming process. An automatic method of generating code would be invaluable in developing large systems.

Another disadvantage is the time taken to compile generic packages. The DEC Ada Compiler, for example, recompiles all generics at each instantiation. However, with faster hardware and better compiler techniques, this problem will gradually diminish.

References

[1] THE OFFICIAL HANDBOOK OF MASCOT, VERSION 3.1. G. Bate. ©Crown Copyright 1987.

[2] VAX Ada Language Reference Manual. Digital Equipment Corporation. Copyright ©1985 by Digital Equipment Corporation.

[3] Mascot III with Ada Study (U). Ferranti Computer Systems Ltd. April 1987.

[4] The Internet Protocol : DARPA INTERNET PROGRAM PROTOCOL SEPCIFICATION, September 1981. (RFC791)

DOCUMENT CONTROL SHEET

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	MEMO 4244		Unclassified

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
7784000	ROYAL SIGNALS & RADAR ESTABLISHMENT ST ANDREWS ROAD, GREAT MALVERN WORCESTERSHIRE. WR14 3PS
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

A MAPPING FOR CONVERTING MASCOT DESIGNS TO ADA PROGRAMS

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
DEAN T			1988 . 7	23

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract

The Future Networks section of CC1 Division uses Mascot III in designing parallel systems. These designs are then implemented in the Ada programming language. A systematic method for mapping Mascot designs into Ada programs has been developed, and this document describes the main characteristics of the method. In most applications, the mapping in its basic form is entirely adequate, and so a large part of the document is spent in describing this. However, certain situations present problems which can only be solved by making some alterations. Therefore a short section on these problems and their solutions is also included.

The method was chosen since it produces Ada programs with components corresponding directly to the Mascot design. This was considered a big advantage in producing maintainable and clear code. Mascot Templates (both algorithmic and structural) are represented as generic packages, and Mascot components are instantiations of those packages.

SBO/48