Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

Experimenting Dynamic Linking in Ada

Paola Inverardi, Franco Mazzanti

Nota Interna B4-30
Luglio 1990

Experimenting Dynamic Linking in Ada

Paola Inverardi, Franco Mazzanti
Istituto di Elaborazione dell'Informazione - CNR
Via S. Maria n. 46, I-56100 PISA
inverard@icnucevm.cnuce.cnr.it
mazzanti@icnucevm.cnuce.cnr.it

Abstract

A proposal to achieve dynamic reconfiguration within the framework of the Ada programming language [1] is presented. A dynamic linking kernel facility is illustrated and its implementation using the Verdix VADS 5.5 Ada compiling system on a SUN3-120 running the BSD Unix version 4.3 operating system is discussed. This kernel allows an Ada program to dynamically change its own configuration, linking at run-time new pieces of code. It is shown how this dynamic facility can be consistently integrated at the Ada language level, without introducing severe inconsistencies with respect to the standard semantics.

1 Introduction

Certain kinds of software systems are expected to run for a long time (months or years) without interruptions, for example telecommunication systems or spacecraft control systems. For these systems, code maintenance (i.e. corrections of errors detected after the program has been installed and when it is actively working) or, more in general, reconfiguration (i.e. adding new functionalities) must be performed at run-time.

Very few programming languages or systems provide appropriate constructs either for facilitating this kind of activity, or for ensuring the correctness and the consistency of the changes which are made [2].

In particular, even though its definition had embedded systems as specific target, Ada does not provide any specific high level features for dealing with this issue. In the language definition, it is clearly stated that the structure of the program is fixed at link time, once the main program has been established. In fact, the issue of dynamic (run time) reconfiguration has never been considered for Ada since the definition of the language requirements [3].

However, Ada is likely to be used for the development of long-lived systems (e.g it will be employed in the Columbus ESA project, as well as in NASA projects), and in any case techniques for dynamic reconfiguration should be investigated.

In section 2, we discuss a possible technique for introducing a kernel facility for dynamic reconfiguration into an Ada program, and present ongoing experimentation on a commercially available Ada system.

In section 3, possible ways of integrating the kernel facility at a higher level of abstraction in Ada are briefly discussed focussing the attention on the simplicity of the proposed approach and its smooth integration with the standard semantics.
In section 4, our experimentation is compared with other approaches to dynamic reconfiguration, found in the literature.

2 Description of the kernel facility

In order to introduce the kernel facility proposed we will use the hardware-like familiar concepts of slots and cards.

Software slots: In principle, they can be seen as holes in the program itself that can be empty when program execution starts. They can be filled at run-time with new program components, which can either remain there until the end of the program execution, or can, in their turn, be substituted with new program components without blocking or restarting program activity.

Software cards: They represent the program components which can fill (or change) the content of the software slots of a program already in execution.

In the following we give our answers to the following questions:
- What is the Ada counterpart of a software slot ?
- How can an Ada program with empty software slots be constructed and put into execution ?
- How can a software card be constructed ?
- How can a software card be inserted (loaded) into the software slot ?
- How is control given to the software card ?
- How is a software card substituted ?

In our discussion of these points, we make three basic assumptions:

I) We want to analyse the development of a set of Ada packages providing the functionality of a kernel for the dynamic (run-time) reconfiguration of Ada programs. Note that we are interested in the case in which the reconfiguration process is driven from the Ada program itself (i.e. a kind of auto-configuration). In particular, we do not want to rely on the existence of an external monitoring/reconfiguration system which is responsible for all changes (as happens in [2]).

II) The analysis is carried out using a commercially available compiling system (Verdix VADS 5.5) whose tools (compiler, linker, library manager) must be considered as an untouchable "black box" (i.e. we neither have the possibility nor do we want to modify the existing

software environment tools). Similarly, the underlying operating system is supposed to be a standard, commercially available operating system (BSD Unix version 4.3) whose tools are considered untouchable in the same way as the software environment tools.

III) The Ada counterpart of our notion of " software cards" should be the "separate subprogram body" Ada construct. Separate subprogram bodies of Ada have the twofold advantage of being separately producible objects and of having a very simple external interface (i.e. the subprogram entry point). Moreover, we only consider the case in which these separate subprogram bodies have no new global context, i.e. they have no "with clauses" to a unit not already accessed by its "ancestor" units.

Although this last assumption might seem rather arbitrary, it provides a reasonably simple framework, general enough to experiment significantly the construction of our prototypal kernel. Furthermore, this choice still permits several kinds of dynamic program reconfiguration to be examined. Other advantages of using separate subprograms instead of top-level subprogram bodies are discussed in section 3.

In this section, we will not deal with the various policies adopted to ensure the safe interleaving of loading/use/unloading operations on a software card. Such issues will be briefly discussed in section 3, when presenting a possible way to integrate the kernel facility at language level. The idea is that these policies are induced by the way the integration is performed and should allow for a safe dynamic program reconfiguration, i.e. the reconfiguration step should, in some way, preserve the Ada semantics.

The basic schema illustrated in the following section is still under development to increase its robustness and easiness of use. In particular, the possible advantages of porting the whole system under SUN_OS® 4.0.3 or Unix System V® are still to be investigated.

2.1 How to build programs containing software slots.

The analysis presented in this section has been performed by considering a specific Ada system, the Verdix VADS5.5®, running on a SUN3-120® workstation under BSD-Unix 4.3®.

The main semantics behind the notion of software slots is that it should denote a piece of memory into which the code of the dynamically linked/loaded software component can be written.

Of the several possible alternatives, in modelling software slots, we give here the two which we feel represent the most natural lines of investigation:

1) A software slot corresponds to a piece of memory (of a known static size) located at a static address in the static data segment of the program. In this case, the software slot might be denoted by a standard Ada object whose size is decidable at compile time and whose address is retrievable at link-time (e.g. the declaration of the object appears at the outmost level inside a library package). Figure 1 illustrates an example of such a static slot.

2) A software slot corresponds to a piece of memory (whose size can be defined at run-time) located inside the dynamic data segments of the program (e.g. inside the program stack or heap) at an address which is only known at run-time. In this case, the software slot might be denoted by a standard Ada object, whose size can be defined dynamically and whose declaration can appear inside a task, subprogram, or generic unit, or it can be created by evaluating an allocator. Figure 2 illustrates an example of such a dynamic slot.

There are other ways to allocate the memory resource needed to contain the code of the software cards (e.g. making "holes" in the code segment of the program, or expanding dynamically the virtual address space usable by the program). However, the two alternatives, 1) and 2), above have been preferred because they have an explicit and clear Ada interface, which meets our requirement that the program reconfiguration should be handled from inside the program itself.

2.2 How to build software cards

The production of our counterpart of software slots, i.e. the dynamically linkable separate subprogram bodies puts several requirements on the level of flexibility offered by the underlying software development tools.

Compilation

1) It must be possible to separately compile the subprogram body (this is guaranteed by the Ada standard).

From Compilation to Linking

2) The compiler should actually produce (and make accessible) a separate executable object code for these bodies. In particular, the compiler should not make these pieces of code unaccessible or defer the code generation phase to when the main program is determined and physically constructed (i.e. the subsequent link phase should only merge the various program components, resolving any inter-module references).
   In our case, the VADS system satisfies this requirement, with the restriction that a separate subprogram cannot be part of a generic unit.

Linking

3) It must be possible to resolve the external references of a subprogram body with respect to the other already developed program modules, without merging all the objects into one.
   This somewhat language-independent requirement is satisfied by the Unix linker "ld" tool (which is, in our case, called by the Ada linking tool). The -A option, in fact, allows to specify, to the Unix linker, the name of an executable file whose symbol table has to be used to complete the definitions of the currently linked module.

This property of the Unix linker is not sufficient for our purposes. As we manipulate Ada programs obtained via the standard Ada system, a main program has to be complete, i.e. all the necessary modules have to be provided at link time. This means that our potentially incomplete programs (programs with empty slots) are realized by using the artifice of providing default versions for all the separate bodies we intend to link dynamically.
In this way, when the (dynamic) linking of an actual version of these separate (default) bodies is attempted (with respect to the symbol table of the main program), a "multiply-defined" diagnostics is issued by the linker, since some of the symbols exported by the dynamic subprogram are already defined. In any case, the linked file produced by the linker remains consistent, and only a small patch is needed to make it really executable. A patching routine has thus been defined in order to correctly adjust the header of the file containing the dynamically loadable code, computing and storing the correct value of the card entry-point in it.
This value is obtained by adding the entry-point offset (found inside the header of the file produced by the compiler containing the not yet linked code of the sw-card) to the base address used by the linking operation.

From Linking to Loading

4) The code produced by the compiler for these separate subprograms has to be either a fully position independent code (i.e. can be loaded at any address) or an absolute linking base should be established for the final link operation.
In the second case, which is the one occurring in the VADS system, the absolute linking base should be exactly the physical address of the "software slot" into which the separate subprogram will be loaded. This implies that, if the physical address of the software slot to be used is statically known (and retrievable) the software card might be produced off-line with respect to the main program execution. If the software slot has been dynamically sized, or dynamically allocated in some stack or heap, or dynamically chosen among the static software slots available at a given time, its physical address is only known dynamically. In this case, the linking activity must be driven by the main program itself providing the information on the absolute linking base.

To increase the safety of the dynamic reconfiguration mechanism, some kind of standardization of the structure of the names of the source, object, and loadable files should be introduced, e.g. including the information of the slot address for which they have been prepared.
In particular, we require the source code of a dynamic subunit *unit-name* to be the only unit in the file "*unit-name*.a", the file holding its object code (produced by the compiler) to be named "*unit-name*.o", and the various versions of loadable code prepared for different slots to be stored in files having name "*unit-name.slot-address*.out". All these names could be postfixed with some versioning information.

This simplifies the activity of checking for the existence of an appropriate card for a given slot guaranteeing a high level of consistency between the development system producing the sw_cards and the main program execution loading them at the appropriate address.
An outline of the various steps illustrating the construction of a "software card" is illustrated in Figure 3.

2.3 Dynamic loading of software cards

This is probably the least troublesome of the various activities to be performed.
Once the software slot is known and the software card has been produced, it is sufficient to open the file produced by the linker (and appropriately patched), extract the subprogram entry point and the actual size of the following code from the standard header, and store the entry-point-value and the subprogram code in the Ada object modelling the slot (for an example see Figure 1).
Figure 4 illustrates the skeleton of the routine used to load the software card into the slot.
Figure 4, and the other figures, only illustrate the basic skeleton of the activity in the sense that most of the necessary checks (and corresponding recovery actions) are not illustrated (e.g. the fact that NAME_ERROR can be raised by the OPEN subprogram call is not considered here).

2.4 Passing Control to the Dynamic Subprograms

The Ada interface for a dynamically linked subprogram is just its subprogram specification as it appears in the main program.
Since (as already stated) both the VADS and Unix systems require the main program to be complete in order to produce the corresponding executable file, our solution is to introduce inside the main program a default version of those subprograms which we want to load at run-time. These default versions are written by using machine code insertions and the only thing they do is a jump (with the stack unchanged) to the entry-point of the dynamically loaded subprogram.
Figure 5 illustrates a simple example of a main program with its static empty slot and default component.

3 Towards a safe integration of the kernel facility into the language

An Ada program is composed by several compilation units: the program library units, their bodies, and a certain number of separate bodies. A separate body, in particular, is a separately compiled body of a program unit whose specification is declared inside another program unit. Separate bodies are represented in the declaring compilation unit by a body stub. One of the major differences between library unit bodies and separate bodies is that the former are elaborated just once and before the beginning of the execution of the main program, while the latter are elaborated (possibly many times) each time the corresponding body stub is elaborated, as part of the elaboration of the declarative part in which they appear. The particular dynamicity and flexibility of separate

subprograms, with respect to the library subprograms, allows us to test several patterns of dynamic reconfiguration (see Figures 5, 6, 7 and 8).

For example, Figure 5 illustrates the simplest, but unsafe, use of our kernel facility. In this case, a subprogram directly loads a new version of a software card into a static software slot, and jumps into it. Notice that a very unpredictable behaviour can result if, for example, two calls of MAIN1 occur simultaneously. Notice also that, during the execution of MAIN1, the execution of one statement (the call of LOAD_CARD) can change the current meaning of a subprogram definition already elaborated and potentially in use.

The conclusion from this example is that the use of our kernel facilities should be regulated in some way.

In Figure 6 a safe approach to dynamic reconfiguration is shown: since a unique static slot is used, the access to it is monitored by a task which guarantees that calls of a software card do not occur when the card is not present, and that loading operations do not occur while the previous card is being executed.

In both the examples of Figures 5 and 6 we do not address the issue of how dynamic software cards are produced, because the software slot is static, its address is known in the development environment, and appropriate dynamic cards could be produced "off-line", as discussed in section 2.2.

Instead, in Figure 7, an example of use of dynamic slots is shown. In this case, the task monitoring the slot is a task type, and each task object has its own private slot. The main difference from the previous example, is that now the physical address of the slot is only known at run-time. Hence dynamic software cards cannot be produced completely off-line. In the example of Figure 7, it is the task object which drives the construction of the appropriate cards, calling a LINK_CARD subprogram with the necessary slot address information (this LINK_CARD subprogram could either directly activate the unix linker "ld" with the appropriate parameters, or send a message to some other "external agent" requiring the linking operation to be performed). Notice how our naming conventions ensure the correctness of the load operation for both the static and the dynamic slots.

Finally, in Figure 8, an example of a particularly dynamic yet safe use of our reconfiguration mechanism is given. In this case, each call of the EXECUTE_STMT subprogram results in compiling, linking, loading and executing a new version of the software card. In this way, the direct interpretation of the Ada statement provided as string parameter to the procedure is implemented. When a call of this subprogram is completed, the memory used for the slot is automatically released. Since compiling, linking and loading operations are in general rather complex, a call of this subprogram is in general very inefficient; however this inefficiency can, in particular cases, be acceptable (e.g. during a recovery routine activated in consequence of the detections of a severe error, and aimed at loading an ad hoc software card to inspect the status of the system).

This example also illustrates how the activity of dynamically linking a new version of a software card can be restricted, allowing it only immediately before the corresponding default separate body is elaborated, and not allowing further reconfigurations until the execution of the frame into which it appears is terminated. This approach is, from our point of view, rather "clean" with respect to the standard Ada semantics, because the operation of loading a new dynamic body does not have the semantics of changing an already existing definition of a subprogram in the environment. Since the software cards have no new context it is possible to look at the elaboration of the body of the software slot as if the new dynamic body were part of the initial program. Furthermore, it is guaranteed that it remains active (and unchangeable) for all the time in which it is visible. The trasparency of the change (in the sense of [4]) is guaranteed by the program structure itself.

4 Conclusions and Comparisons

Although dynamic configuration for high level languages, such as Ada, has been under discussion for several years, up until now only a very few projects have addressed it in a satisfactory way. In this section, we will only mention those projects that, to our knowledge, have had dynamic reconfiguration as a primary goal. In this respect, we can roughly divide the various approaches into two: i) methods which rely on the existence of an external system performing dynamic reconfiguration of a program (the program is written in a suitable language and satisfies a certain number of *reconfiguration* constraints); ii) methods which add capabilities to the language in which the reconfigurable program is written so that the change can be performed at run time by the program itself.

In the first class we put the popular system CONIC [2, 5], while in the second we include the DRAGON project [6] and the Cnet project [8, 9, 10, 11].

CONIC is a well known system devoted to the development of distributed systems; its main feature concerns configuration facilities including dynamic configuration. A specific configuration language has been defined separately from the application language, in order to specify the configuration of modular components into a system. The system provides a set of tools to allow static configuration and dynamic configuration to be performed in a safe and consistent way. With respect to dynamic configuration, a number of constraints on the way the software components are built and on the way they can interact have to be satisfied. Once the configuration program is written, it is the system which validates the required changes and provides the necessary run time support to accomplish them. The constraints to be satisfied are quite strong and visibility constraints can be very severe. This appears reasonable since the CONIC system particularly addresses the issue of run time configuration of distributed systems.

In the second class of projects, there are a number of different approaches which try to cope with run time configuration in a more general context.

DRAGOON [7] is an object oriented language developed in the ESPRIT DRAGON project it provides dynamic binding facilities which can be used to specify, at run time, the element of a class which is to provide the required service. An interesting feature of DRAGOON is that it can be compiled into an Ada program; the main difference from our approach is that it is possible, at run time, to change the binding between a specification and its body by choosing among a set of bodies fixed at link time, but it is not possible to produce, on line, a real program change without restarting the whole program.

The Cnet project was a project funded by a Special Program for Computer Science of the Italian National Research Council (C.N.R), aimed at the implementation of a distributed system on a local area network. One of the goals of the project was to study the feasibility of Ada as a unique development language, i.e. as a system as well as an application language. Some new capabilities had to be added to those already available in Ada, in order to ease the process of software design in a distributed environment and in particular in the area of system (re)configuration. Facilities were in fact proposed for each of the phases in the life of a software system: compile time configuration for the design phase, run time configuration for the activation phase, and run time reconfiguration for the adaptation phase.

For dynamic configuration, the idea was to integrate in the language a dynamic linking facility by means of a linguistic extension based on the declaration of a **dynamic package** whose body is computed at run time as the value of an expression of a predefined (sub)type BODY_TYPE.

A final report on Cnet can be found in [9] and a brief overview is given in [8].

```
package DYN_TYPES is
   type WORD_TYPE is range -2**16 .. 2**16-1;
   for WORD_TYPE'SIZE use 32;
   type CODE_TYPE is
      array (INTEGER range <>) of WORD_TYPE;
end DYN_TYPES ;

with SYSTEM, DYN_TYPES;
package SLOT1 is
   SLOT_CODE: DYN_TYPES .CODE_TYPE(1..10_000);
   SLOT_ENTRY: SYSTEM.ADDRESS := SYSTEM.NO_ADDR;
    -- NO_ADDR is an implementation dependent null address value
   pragma EXTERNAL_NAME(SLOT_CODE,"SLOT1");
    -- This non-standard pragma associates an external symbol with the
    -- SLOT_CODE object.
    -- The link time value of this symbol can be accessed
    -- from the program symbol table using the unix "nm" facility.
    -- This value is the same as SLOT_CODE(1)'ADDRESS, and
    --  is needed to produce sw-cards for this slot.
end SLOT1;
```

Fig 1 : A static slot.

```
with SYSTEM, DYN_TYPES ;
package SLOTS  is
   type CODE_REF is access DYN_TYPES.CODE_TYPE;
   CODE_PTR: CODE_REF:= new DYN_TYPES.CODE_TYPE(1..10_000);
   SLOT_ENTRY.SYSTEM.ADDRESS:= SYSTEM.NO_ADDR;
   --  NO_ADDR is an implementation dependent null address value
   -- The value of CODE_PTR.all(1)'ADDRESS is used
   -- for the construction of sw-cards for the slot.
end SLOTS ;
```

Fig 2 : A dynamic slot.

```
-- << PRODUCTION OF THE MAIN PROGRAM >>

ada  -M  main.a  -o main.out
-- Compiles and links (-M option) the program inside the file main.a
-- Constructs an executable file named main.out
-- The program contains a main subprogram called "main" and
-- the default version of a separate subprogram called "sw_card".

-----------------------------------------------------------

-- << COMPILATION OF SOURCE CODE OF A SW_CARD >>

ada  main.sw_card.a
-- Compiles a second version of the separate subprogram "sw_card"
-- (whose source code is in file main.sw_card.a).
-- The result of the compilation is stored under the directory .objects

-- << LINKING OF OBJECT CODE OF A SW_CARD >>

ld   -A main.out                                \
     -T <hex_slot1_address>                     \
     -o sw_card.<hex_slot1_address> .out        \
     .objects/sw_card.o
-- Links the result of the compilation (the file .objects/sw_card.o)
-- with respect to the symbol table of the main program (main.out).
-- Uses <hex-slot-address> as linking base, and
-- producing a loadable absolute file called sw_card.out .
-- <hex_slot1_address> can be either retrieved from the main program
-- symbol table by looking for the appropriate symbol value with the "nm"
-- unix facility, or from the executing main ('ADDRESS attribute).

-- << PATCHING THE LOADABLE CODE HEADER >>

fixheader .objects/sw_card.o                    \
          <slot_address>                        \
          sw_card.<hex_slot1_address> .out
-- calls a patching routine to fix the problem of multiply-defined symbols
```

Fig 3 : The outline of a small script illustrating the construction of a "software card" .

```
with DYN_TYPES; use DYN_TYPES;
with SEQUENTIAL_IO, SYSTEM;
procedure LOAD_CARD (
          FILE_NAME: STRING;
          INTO_ENTRY: out SYSTEM.ADDRESS;
          INTO_CODE : in out DYN_TYPES.CODE_TYPE) is
  package WORD_IO is
        new SEQUENTIAL_IO (DYN_TYPES.WORD_TYPE);
  CARD_FILE: WORD_IO.FILE_TYPE;
  WORD: DYN_TYPES.WORD_TYPE;
  CODE_SIZE : WORD_TYPE;
begin
  WORD_IO.OPEN(CARD_FILE,WORD_IO.IN_FILE, FILE_NAME);
   -- Skip the first 4 bytes of the header (machine type and magic number)
  WORD_IO.READ(CARD_FILE, WORD);
   -- Read text size of loadable file
  WORD_IO.READ(CARD_FILE, CODE_SIZE);
   -- Read data size of loadable file
  WORD_IO.READ(CARD_FILE, WORD);
  CODE_SIZE := CODE_SIZE + WORD;
   -- Read BSS size of loadable file
  WORD_IO.READ(CARD_FILE, WORD);
  CODE_SIZE := CODE_SIZE + WORD;
   -- Convert the size from bytes to words
  CODE_SIZE := (CODE_SIZE +3) / 4;
   -- Skip next  word from the header (symbol table size)
  WORD_IO.READ(CARD_FILE, WORD);
   --    Read entry point field
  WORD_IO.READ(CARD_FILE, WORD);
  INTO_ENTRY :=
          SYSTEM.PHYSICAL_ADDRESS(INTEGER(WORD));
   -- The implementation defined PHYSICAL_ADDRESS  function
   --    converts an integer value into an address value.
   -- Skip next  2 words from the header (reloc text and data size)
  WORD_IO.READ(CARD_FILE, WORD);
  WORD_IO.READ(CARD_FILE, WORD);
   -- Read code into slot
  for I in 1.. CODE_SIZE loop
     WORD_IO.READ(CARD_FILE, WORD);
     INTO_CODE(I):= WORD;
  end loop;
  WORD_IO.CLOSE(CARD_FILE);
end LOAD_CARD ;
```

Fig 4 : Loading the code and the entry-point into a software slot.

```
with DYN_TYPES;                    -- the slot types          (see Fig. 1)
with LOAD_CARD;                    -- the loading routine     (see Fig. 4)
with SLOT1;                        -- a static software slot  (see Fig. 1)
with HEX_IMAGE;                    -- a user defined ADDRESS to STRING
with TEXT_IO;                      --                  conversion function
procedure MAIN1 is
   -- construct the appropriate file name for the sw_card
   SLOT_ADDR: constant STRING :=
                          HEX_IMAGE(SLOT1.SLOT_CODE(1)'ADDRESS);
   FILE_NAME: constant STRING := "sw_card." & SLOT_ADDR & ".out";
   -- the software card interface
   procedure SW_CARD (...) is separate;
      -- the default body of SW_CARD performs an immediate jump
      -- to the address stored in SLOT1.SLOT_ENTRY
begin
   -- load the sw-card into the sw-slot;
   LOAD_CARD (
         FROM_FILE  => FILE_NAME;
         INTO_ENTRY => SLOT1.SLOT_ENTRY;
         INTO_CODE => SLOT1.SLOT_CODE );
   -- calls the card functionality
   SW_CARD(...);
end MAIN1;


-- the default definition of SW_CARD performing the indirect call
with SLOT1;
with MACHINE_CODE; use MACHINE_CODE;
separate (MAIN1)
procedure SW_CARD (...) is
   -- this non-standard implementation defined pragma is needed
   -- to disable the standard code manipulating the stack
     pragma IMPLICIT_CODE(OFF);
begin
   -- the implementation defined 'REF attribute can be used to specify
   -- in machine code instructions references to visible Ada entities.
   CODE_2'(MOVEA_L, SLOT1.SLOT_ENTRY'REF, A0);
   CODE_1'(JMP, INDR(A0));
end SW_CARD ;


-- the dynamic card for MAIN1
separate (MAIN1)
procedure SW_CARD (...) is
   STR: STRING(1..50);
begin
   STR := (others => 0);
   TEXT_IO.PUT_LINE(STR);
   TEXT_IO.PUT_LINE("hello: " & FILE_NAME & " being executed");
   TEXT_IO.PUT_LINE(STR);
end SW_CARD ;
```

Fig 5 : The default body for dynamically linked subprograms allows indirect jumps to the correct dynamic entry points.

```
package DYN_UNIT is
  task RECONFIGURABLE_SERVICE is
      entry SERVICE(...);
      entry RECONFIGURE(...);
  end RECONFIGURABLE_SERVICE ;
end DYN_UNIT;

package body DYN_UNIT is
  task body RECONFIGURABLE_SERVICE is separate;
begin
   null;
end DYN_UNIT;

with LOAD_CARD, SLOT1, HEX_IMAGE;
-- HEX_IMAGE is an ADDRESS to STRING conversion function
separate (DYN_UNIT)
task body RECONFIGURABLE_SERVICE is
    -- construct the appropriate file name for the sw_card
    SLOT_ADDR: constant STRING :=
                          HEX_IMAGE(SLOT1.SLOT_CODE(1)'ADDRESS);
    FILE_NAME: constant STRING := "sw_card." & SLOT_ADDR & ".out";
    -- the software card interface
   procedure SW_CARD (...) is separate;
      -- the default body of SW_CARD performs an immediate jump
      -- to the address stored in SLOT1.SLOT_ENTRY
begin
   loop
     select
      when SLOT1.SLOT_ENTRY /= SYSTEM.NO_ADDR =>
         accept SERVICE (...) do
           SW_CARD (...)
         end SERVICE ;
     or accept RECONFIGURE (...) do
         -- the file "sw_card.out" is should always contain the
         -- last version of the software card for CALL_SERVICE.
           LOAD_CARD(
                   FILE_NAME,
                   SLOT1.SLOT_ENTRY,
                   SLOT1.SLOT_CODE);
       end RECONFIGURE ;
     or terminate;
     end select;
   end loop;
end RECONFIGURABLE_SERVICE ;
```

Fig 6 : Monitoring reconfigurations and accesses of a software card inside a task

```
package DYN_UNIT is
  task type RECONFIGURABLE_SERVICE is
    entry SERVICE(...);
    entry RECONFIGURE;
  end RECONFIGURABLE_SERVICE ;
end DYN_UNIT;

package body DYN_UNIT is
 task body RECONFIGURABLE_SERVICE is separate;
begin
   null;
end DYN_UNIT;

with DYN_TYPES, HEX_IMAGE, LINK_CARD, LOAD_CARD, SYSTEM;
-- HEX_IMAGE is an ADDRESS to STRING conversion function
separate (DYN_UNIT)
task body RECONFIGURABLE_SERVICE is
   MY_CODE:  DYN_TYPES.SLOT_TYPE(10_000);
   MY_ENTRY: SYSTEM.ADDRESS:= SYSTEM.NO_ADDR;
   -- NO_ADDR is an implementation dependent null address value
   -- construct the appropriate file name for my sw_card
   MY_ADDR:  constant STRING := HEX_IMAGE(MY_CODE(1)'ADDRESS);
   FILE_NAME: constant STRING := "sw_card." & MY_ADDR & ".out";
   -- the software card interface
   procedure SW_CARD (...) is separate;
      -- the default body of SW_CARD performs an immediate jump
      -- to the address stored in MY_ENTRY
begin
   loop
     select
      when MY_ENTRY /= SYSTEM.NO_ADDR =>
        accept SERVICE (...) do
           SW_CARD (...);
        end accept;
     or accept RECONFIGURE (...) do
           LINK_CARD( "sw_card.o", MY_ADDR);
           LOAD_CARD (FILE_NAME, MY_ENTRY, MY_SLOT);
        end accept;
     or terminate;
     end select;
   end loop;
end RECONFIGURABLE_SERVICE ;
```

Fig 7 : A reconfigurable_service task type: each task has its own dynamic version

```
with DYN_TYPES, HEX_IMAGE, SYSTEM, TEXT_IO; use TEXT_IO;
 with  COMPILE_CARD, LINK_CARD, LOAD_CARD;
procedure EXECUTE_STMT (STMT: string) is
      MY_CODE:  DYN_TYPES.SLOT_TYPE(10_000);
      MY_ENTRY: SYSTEM.ADDRESS;
      -- construct the source code for the sw_card;

      -- construct  the appropriate  file name for my  sw_card
      MY_ADDR:   constant STRING := HEX_IMAGE(MY_CODE(1)'ADDRESS);
      FILE_NAME: constant STRING := "sw_card." & MY_ADDR & ".out";
      package MAKE_NOW is
      end MAKE_NOW ;
      package body MAKE_NOW is
      begin
          OPEN (SOURCE, "sw_card.a");
          PUT_LINE(SOURCE,"separate (EXECUTE_STMT)");
          PUT_LINE(SOURCE, "procedure SW_CARD is " );
          PUT_LINE(SOURCE, "begin" );
          PUT_LINE (SOURCE,    STMT);
          PUT_LINE (SOURCE, "end SW_CARD");
          CLOSE(SOURCE);
          COMPILE_CARD("sw_card.a");
          LINK_CARD( "sw_card.o", MY_ADDR);
          LOAD_CARD (FILE_NAME, MY_ENTRY, MY_CODE);
      end MAKE_NOW ;
       procedure SW_CARD  is separate;
      -- the default body of SW_CARD performs an immediate jump
      -- to the address stored in  MY_ENTRY
  begin
     SW_CARD(...);
  end EXECUTE_STMT ;
```

Fig 8 : A subprogram which auto-reconfigure at each call

References

[1] A.J.P.O: "Reference Manual for the Ada Programming Language", ANSI/MIL-STD 1815 A, January 1983.

[2] J.Kramer, and J.Magee: "Dynamic Configuration for Distributed Systems", IEEE Transactions on Software Engineering, vol. SE-11, no. 4, pp 424-436, April. 1985.

[3] Department of Defense: "Requirements for High Order Computer Programming Languages - STEELMAN", June 1978.

[4] K.W.Tindell: "Dynamic Code Replacement and Ada" To appear in AdaLetters

[5] J.Kramer, and J.Magee: " Constructing Distributed Systems in CONIC", IEEE Transactions on Software Engineering, vol. SE-15, no. 6, pp 663-676, June 1989.

[6] A.Di Maio, F.Bott, I.Sommerville. R.Bayan, M.Wirsing : "The DRAGON Project", 1989 Esprit Conference, Brussels, Nov/Dec 1989, pp. 554-567.

[7] C.Atkinson: "An Object-Oriented Language for Software Reuse and Distribution" PhD Thesis, Department of Computing, Imperial College, London, 1990.

[8] L.Svobodova: "Summary ACM SIGOPS Workshop on Operating Systems in Computer Networks", Jan. 28-30 (1985) Ruschlikon, Switzerland, ACM SIGOPS Vol. 19, N. 2, April 1985.

[9] P.Inverardi, F.Mazzanti, U.Montanari, C. Montangero, P.Rasoini, G.N.Vallario: "Distributed System Design, Configuration and Reconfiguration" In "Distributed Systems on Local Networks", Proceedings of the Final Conference of "Progetto Finalizzato Informatica, Obiettivo Cnet" - CNR - Pisa, June 1985.

[10] P.Inverardi, F.Mazzanti, C. Montangero: "The use of Ada in the Design of Distributed Systems" In "Ada in Use", Proceedings of the Ada International Conference 1985, The Ada Companion Series, Cambridge University Press (May 1985).

[11] P.Inverardi, F.Mazzanti, C. Montangero: "Configuration of Distributed Systems in Ada" Internal Report S-86-7 del Dipartimento di Informatica, Università di Pisa.