SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(1), 1–14 (JANUARY 1993)

Experimenting with Dynamic Linking with Ada

PAOLA INVERARDI AND FRANCO MAZZANTI
Istituto di Elaborazione dell'Informazione (CNR), Via S. Maria 46, I-56100 Pisa, Italy

SUMMARY

An approach to achieving dynamic reconfiguration within the framework of Ada[1] is described. A technique for introducing a kernel facility for dynamic reconfiguration in Ada is illustrated, and its implementation using the Verdix VADS 5.5 Ada compiling system on a Sun3-120 running the 4.3 BSD Unix operating system is discussed. This experimental kernel allows an Ada program to change its own configuration dynamically, linking new pieces of code at run-time. It is shown how this dynamic facility can be integrated consistently at the Ada language level, without introducing severe inconsistencies with respect to the Standard semantics.

KEY WORDS Ada Dynamic reconfiguration Dynamic linking

INTRODUCTION

Certain kinds of software systems are expected to run for a long time (months or years) without interruption, e.g. telecommunication systems or spacecraft control systems. For these systems, code maintenance (i.e. correction of errors detected after the program has been installed and when it is actively working) or, more generally, 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 that are made.[2]

In particular, Ada does not provide any specific high-level features for dealing with this dynamic reconfiguration, even though its definition had embedded systems as the specific target. The language definition carefully avoids defining implementation issues beyond what is minimally essential for an adequate language design to meet the imposed requirements:[3] any issues that go beyond this are quite properly left to the implementer. In particular, the Standard deliberately avoids either prohibiting or requiring dynamic reconfiguration.

Thus, since 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), techniques for dynamic reconfiguration need to be investigated.

In fact, the proposed revision of the Ada Standard (the so-called '9X' version) explicitly demands the implementation of dynamic reconfiguration facilities.[4]

0038–0644/93/010001–14$12.00

Received 13 July 1990
Revised 20 April 1991

In the next section, we discuss a possible technique for introducing a kernel facility for dynamic reconfiguration into an Ada program, and present ongoing experimental work on a commercially-available Ada system.

In the section after that, possible ways of integrating the kernel facility at a higher level of abstraction in Ada are discussed briefly, focusing attention on the simplicity of the approach proposed and its smooth integration with the Standard semantics. To conclude, we compare our proposal with other approaches to dynamic reconfiguration found in the literature.

DESCRIPTION OF THE KERNEL FACILITY

In order to introduce our kernel facility, we shall use familiar hardware-like concepts of slots and cards.

(a) *Software slots*. In principle, these slots can be seen as holes in the program itself and may be empty when program execution starts. They can be filled at run-time with new program components, which either remain until the end of the program execution, or can, in their turn, be replaced by new program components without blocking or restarting program activity.

(b) *Software cards*. These cards 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 discuss six questions:

1. What is the Ada counterpart of a software slot?
2. How can an Ada program with empty software slots be constructed and executed?
3. How can a software card be constructed?
4. How can a software card be inserted (loaded) into a software slot?
5. How is control given to a software card?
6. 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 by 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 Ref. 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 untouchable (i.e. we neither can nor want to modify the existing software-environment tools). Similarly, the underlying operating system is presumed to be a standard commercially-available operating system (BSD Unix version 4.3) with tools equally untouchable.

(iii) The Ada counterpart of our notion of 'software card' is the 'separate subprogram body' Ada construct (i.e. subunit). Separate subprogram bodies in Ada have the twofold advantage of being separately-producible objects and of having a very simple external interface (i.e. the subprogram's start-point).

```
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 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;
```

Figure 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 ;
```

Figure 2. A dynamic slot

the alternatives (1) and (2) above have been preferred because they have an explicit and clear Ada interface, and so meet our requirement that the program reconfiguration should be handled from inside the program itself.

How to build software cards

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

1. Compilation

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

2. From compilation to linking

The compiler should actually produce (and make accessible) separate executable object code for these bodies. In particular, the compiler should not make these

is generated. On the other hand, if the slot is to be allocated dynamically, its size can be determined exactly by retrieving it from the file header. This is what actually happens in our implementation, even though it has not been shown in the examples for the sake of simplicity.

In order to increase the safety of the dynamic reconfiguration mechanism, some kind of standardization for 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 version information.

This simplifies the activity of checking for the existence of an appropriate card for a given slot and guarantees a high level of consistency between the development system producing the software 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. Of course, this example is VADS-dependent. It is possible to do the same things in slightly different ways using other commercial systems provided that they satisfy the above-mentioned requirements.

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 adjusted), extract the subprogram's start-point and the actual size of the following code from the standard header, and store the start-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.

In all the Figures, only the basic schema of the activity is illustrated. Most of the necessary checks (and corresponding recovery actions) are not shown (e.g. the fact that NAME_ERROR can be raised by the OPEN subprogram call is not considered here), for simplicity.

Passing control to the dynamic subprograms

The Ada interface for a dynamically-linked subprogram is just the subprogram specification as it appears in the source program.

Since, as already stated, both the VADS and Unix systems require the program to be complete in order to produce the corresponding executable file, our solution is to introduce a default version of those subprograms that we want to load at run-time inside the program. These default versions are written using machine-code insertions, and the only thing they do is to jump (with the stack unchanged) to the start-point of the dynamically-loaded subprogram.

```
-- << 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).

-- << ADJUSTING THE LOADABLE CODE HEADER >>

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

Figure 3. A small script illustrating the construction of a 'software card'

Figure 5 illustrates a simple example of a program with a static empty slot and default component.

TOWARDS A SAFE INTEGRATION OF THE KERNEL FACILITY INTO THE LANGUAGE

An Ada program is composed of several compilation units: the program-library units, their bodies, and a certain number of separate bodies. A separate body 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 before the beginning of the execution of the main program, whereas 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 dynamic character and flexibility of the separate subprograms with respect to library subprograms allows us to test several patterns of dynamic reconfiguration (see Figures 5–8).

```
-- << 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).

-- << ADJUSTING THE LOADABLE CODE HEADER >>

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

Figure 3. A small script illustrating the construction of a 'software card'

Figure 5 illustrates a simple example of a program with a static empty slot and default component.

TOWARDS A SAFE INTEGRATION OF THE KERNEL FACILITY INTO THE LANGUAGE

An Ada program is composed of several compilation units: the program-library units, their bodies, and a certain number of separate bodies. A separate body 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 before the beginning of the execution of the main program, whereas 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 dynamic character and flexibility of the separate subprograms with respect to library subprograms allows us to test several patterns of dynamic reconfiguration (see Figures 5–8).

```
with 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;
 TEXT_SIZE : DYN_TYPES.WORD_TYPE;
 DATA_SIZE : DYN_TYPES.WORD_TYPE;
 BSS_SIZE : DYN_TYPES.WORD_TYPE;
 START_POINT : DYN_TYPES.WORD_TYPE;
 CODE_SIZE : DYN_TYPES.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, TEXT_SIZE);
 -- Read data size of loadable file
 WORD_IO.READ(CARD_FILE, DATA_SIZE);
 CODE_SIZE := TEXT_SIZE+ DATA_SIZE;
 -- Read BSS size of loadable file
 WORD_IO.READ(CARD_FILE, BSS_SIZE);
 CODE_SIZE := CODE_SIZE + BSS_SIZE;
 -- 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 start point field
 WORD_IO.READ(CARD_FILE, START_POINT);
 INTO_ENTRY :=
        SYSTEM.PHYSICAL_ADDRESS(INTEGER(START_POINT));
 -- 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 ;
```

Figure 4. Loading the code and the start-point into a software slot

For example, Figure 5 illustrates the simplest (but least secure) 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 to it. Note that very unpredictable behaviour can result if, for example, two calls of MAIN1 occur simultaneously. Note also that, during 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.

From this it is clear 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

```
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 implementation defined pragma is needed to disable
   -- the the production of implicit 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 ;
```

Figure 5. The default body for dynamically-linked subprograms, allowing indirect jumps to the correct dynamic start-points

static slot is used, the access to it is monitored by a task which guarantees that calls to 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 the examples of Figures 5 and 6 we do not address the issue of how dynamic software cards are produced. This is because the software slot is static, its address

```
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 ;
```

Figure 6. Monitoring reconfigurations of and accesses to a software card inside a task

is known in the development environment, and appropriate dynamic cards may be produced 'off-line', as we have mentioned in the previous section.

Instead, in Figure 7, an example of the 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 here the physical address of the slot is known only at run-time. Hence dynamic software cards cannot be produced completely off-line. In the example of Figure 7, it is the task object that drives the construction of the appropriate cards, calling a LINK_CARD subprogram with the necessary slot-address information (this LINK_CARD subprogram could either activate the Unix linker ld directly with the appropriate parameters, or send a message to some other 'external agent' requiring the linking operation to be performed). Note

```
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;
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.CODE_TYPE(1..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 ;
```

Figure 7. A reconfigurable_service *task type: each task has its own dynamic version*

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 will compile, link, load and execute 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 usually very inefficient; however, this inefficiency can, in particular cases, be acceptable (e.g. during a recovery routine activated in consequence of the detection 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, by authorizing it only immediately before

```
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.CODE_TYPE(1..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 ;
```

Figure 8. A self-reconfiguring subprogram

the corresponding default separate body is elaborated, and not allowing further reconfigurations until the execution of the frame into which it appears is terminated. From our point of view, this approach is 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 consider 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 the new body remains active (and unchangeable) for all the time it is visible. The transparency of the change (in the sense of Reference 5) is guaranteed by the program structure itself.

CONCLUSIONS AND COMPARISONS

Although dynamic configuration for high-level languages, such as Ada, has been under discussion for several years, until now only a very few projects have addressed it in a satisfactory way. In this section, we shall 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 that rely on the existence of an external system performing dynamic program reconfiguration (the program is written in a suitable language and satisfies a certain number of *reconfiguration* constraints); (ii) methods that 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,[2,6] whereas in the second we include the DRAGON project[7] and the Cnet project.[8–11]

CONIC is a well-known system devoted to the development of distributed systems; its main features are its configuration facilities which include 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 designed to allow both static 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 that validates the changes required 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 pays particular attention to the issue of run-time configuration of distributed systems.

In the second class of projects, we can identify a number of different approaches that try to cope with run-time configuration in a more general context.

DRAGOON[12] 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 the class that 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 in DRAGOON it is possible, at run time, to change the binding between a specification and its body by choosing from 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 funded by a Special Program for Computer Science of the Italian National Research Council (CNR), and 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 language as well as an application language. It was necessary to add some new capabilities 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-cycle 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 a dynamic linking facility into the language, by means of a linguistic extension based on the declaration of a *dynamic package* whose body would be 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 Reference 9, and a brief overview is given by Svobodova.[8]

ACKNOWLEDGEMENTS

We thank Carol Peters for having patiently reviewed the English text and the anonymous referees for their careful examination of the paper.

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 Trans. Software Engineering*, **SE-11**, (4), 424–436 (1985).
3. U.S. Department of Defense, *Requirements for High Order Computer Programming Languages—STEELMAN*, June 1978.
4. *Ada 9X Requirements*, Ada 9X Project Report, December 1990.
5. K. W. Tindell, 'Dynamic code replacement and Ada', to appear in *Ada Letters*.
6. J. Kramer and J. Magee, 'Constructing distributed systems in CONIC', *IEEE Trans. Software Engineering*, **SE-15**, (6), 663–676 (1989).
7. A. Di Maio, F. Bott, I. Sommerville, R. Bayan and M. Wirsing, 'The DRAGON project', *1989 Esprit Conference*, Brussels, Nov./Dec. 1989, pp. 554–567.
8. L. Svobodova, 'Summary ACM SIGOPS Workshop on Operating Systems in Computer Networks, 28–30 January 1985, Ruschlikon, Switzerland', *ACM SIGOPS*, **19**, (2), 6–39 (1985).
9. P. Inverardi, F. Mazzanti, U. Montanari, C. Montangero, P. Rasoini and 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 and 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 and C. Montangero, 'Configuration of distributed systems in Ada', *Internal Report S-86-7*, Dipartimento di Informatica, Università di Pisa, 1986.
12. C. Atkinson, 'An object-oriented language for software reuse and distribution', *Ph.D. Thesis*, Department of Computing, Imperial College, London, 1990.