Migrating Large Applications from Ada83 to Ada95

Philippe Waroquiers, Stef Van Vlierberghe,
Dirk Craeynest, Andrew Hately, and Erik Duvinage

Eurocontrol/CFMU, Development Division
Rue de la Fusée, 96,
B-1130 Brussels, Belgium
{philippe.waroquiers, stef.van-vlierberghe, dirk.craeynest,
andrew.hately, erik.duvinage}@eurocontrol.be

Abstract. The CFMU has developed mission critical applications for Europe-wide flight plan processing and air traffic management activities using Ada83. This paper presents the techniques and tools used for the migration from an Ada83 to an Ada95 compiler and run-time. It puts a particular emphasis on both the software management aspects and the technical aspects e.g. language aspects, run-time evolution, how to cater for incompatibilities between Ada83 and Ada95, elaboration order, etc…

1 Introduction

EUROCONTROL, the European Organization for the Safety of Air Navigation, was tasked in the late 80's by its control body - the ministers of transport of its member states - to establish a Central Flow Management Unit. The CFMU is responsible for the following activities:

- Flight plan processing: receiving flight plans filed by the Aircraft Operators; flight plan validation and correction - manual or automatic. The corrected flight plans are redistributed to the Aircraft Operators and the overflown Airspace Control Centers.
- Air Traffic Flow Management: when the planned traffic load exceeds the capacity of Air Traffic Control (ATC), the CFMU is responsible for balancing the number of flights and available ATC capacity, the objective being optimum use of European airspace and prevention of air traffic congestion.

To support flight plan processing and short term - two days ahead - tactical air traffic flow management, the CFMU has developed the IFPS (Integrated Flight plan Processing System) and the TACT (tactical) system. Both systems were developed with common tools and techniques: HP servers and workstations, Ada83, Motif, Oracle, …. A large proportion of the code is common between the TACT and the IFPS system (see [1] for a more detailed description of some aspects of the architecture and development of the TACT system).

The TACT and IFPS systems became operational in 1995. Since then, major functional evolutions have been applied to both systems to cope with the expansion of air traffic. The traffic has grown by 27% between 1996 and 2000 (see [2]). The number of messages to be handled by the TACT system increased by almost 50%

D. Craeynest and A. Strohmeier (Eds.): Ada-Europe 2001, LNCS 2043, pp. 380-391, 2001.

over the same period (see figure 1). Currently, the system handles ± 25000 flights per day, with peak days up to 28000.

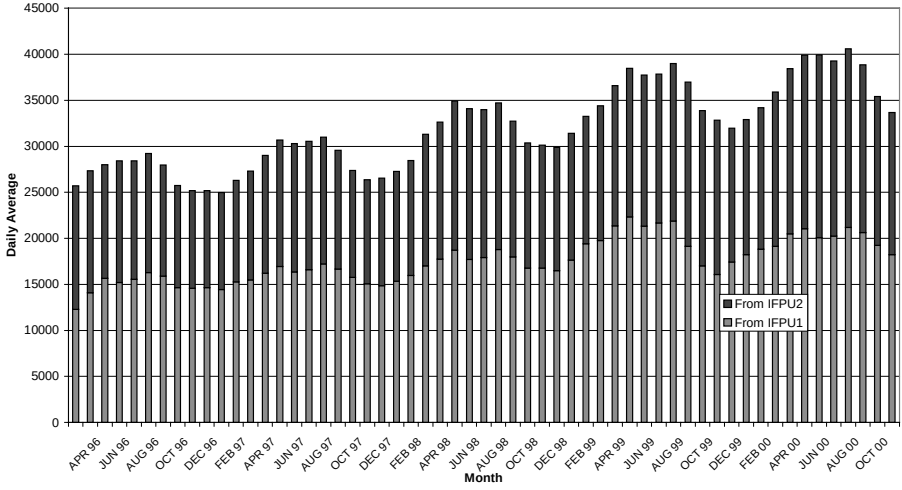

Fig. 1. Daily average of number of messages received by TACT system, per month

The systems have also been upgraded technically to follow up the evolution of the system software; operating system, database and to cope with the increasing demand both in volume and in terms of functionality, all of which necessitated increasing the power of the computers.

The software is large: in total, more than 1.4 million lines of Ada83 source code and 150 thousand lines of C, ksh, TCL/Tk, awk, PL/SQL, … The software is structured in more than 60 software sub-systems and contains more than 2100 Ada packages.

The systems are also complex and integrate a lot of different techniques and technologies to provide the needed functions:

- Shared memories, semaphore, …
- Network access: synchronous and asynchronous IO, TCP/IP, SNA, …
- Low level access, e.g. for tracing (signal tracing, signal mask, stack trace, memory use, …)
- Alsys Ada83 compiler
- Oracle database with SQL language access
- Usage of a POSIX Ada binding
- X and Motif
- …

The short and medium term technical evolutions (2001/2002) are the introduction of version 11 of the HP-UX operating system and the use of CORBA for interfacing with other components.

On the functional side, major evolutions are also under way, in particular for the TACT system which will receive and process radar position data for flights over Europe. These radar positions will enable better prediction of the time, altitude and geographical positions of airborne flights leading to better-optimized management of

airspace capacity. Receiving radar data will imply that the peak number of messages handled by the TACT system will increase from 10 per second to about 200 per second.

These heavy technical and functional evolutions are the main factors that have lead to the project of Ada95 compiler evaluations. The expected benefits of migrating to an Ada95 compiler were:

- The Ada83 compilers (e.g. Alsys) have become mature over the years. With the standardization of the new version of the Ada language [3], a lot of effort has been invested in new compilers. No major evolution or improvement of the Ada83 compiler was to be expected, which means that the use of newly available features such as the thread support by the HP-UX 11 operating system would not be possible while staying with Alsys Ada83.
- Use of CORBA with an Ada83 compiler is possible (see [4] for the description of an Ada83 targeted CORBA IDL compiler) but does not follow the standard CORBA Ada Mapping ([5]) thereby making the selection of a CORBA ORB product more difficult and making the code less portable.
- The Ada95 features like child units, protected objects, controlled types, tagged types, are also believed to bring better ways to develop and maintain our systems.
- Improvement of compilation speed.
- Easier integration of other technologies (e.g. C++).

2 Selection of an Ada95 Compiler

There are some major planning constraints that are imposed by others systems that depend on CFMU systems: air traffic control systems, aircraft operators fleet management systems, and national air traffic management systems. Delivery of new versions must be planned well in advance to give the external users time to adapt their operational procedures and systems. Meeting these planned installation dates - each year around March/April before the summer season - is crucial to the CFMU.

When planning a major change, having a contingency solution in case unexpected technical difficulties are encountered reduces risk. Hence it was decided to select and use an Ada95 compiler for the release to be installed in March 2001, but to keep the complete TACT and IFPS software compatible with Ada83. The evaluation of different Ada95 compilers was done between December 1999 and March 2000. After initial selection, the evaluated compilers were GNAT and ObjectAda. The main criteria used to evaluate the compilers are

- quality/performance of the generated code,
- quality of the associated tools (debugger, profiler, coverage, editor, ….),
- integration (multi-language, available products, ….), and
- quality of support provided by the compiler supplier.

From the beginning, a full-scale evaluation was deemed necessary to minimize the uncertainty. A complete port of TACT and IFPS to GNAT and ObjectAda was made between December 1999 and February 2000 by 4 people working ± 50% of their time on this activity. Our automated battery of regression tests was used to evaluate the functional quality of the code generated by the compiler. Some limited performance measurements were done with different scenarios.

Due to the very limited incompatibilities between Ada83 and Ada95, the necessary adaptations of the software - for example needed by the switch to a thread-based Ada tasking runtime - were done together with the development and finalization of the March 2000 release of the CFMU software. At the end of the compiler evaluation, the baseline code could be compiled by both the Alsys Ada83 compiler and by both of the Ada95 compilers.

After evaluation, the GNAT compiler was selected. The main positive points of GNAT are the quality of the support provided by the compiler supplier, the compilation speed, in particular on a multi-CPU computer, the very well appreciated simplicity of use, the robust make facility, the integration with other tools and the availability of products for GNAT. The main negative points were the poor quality of some aspects closely related to the operating system and the processor, poor quality of the gdb debugger, no support for zero cost exception handling, relative lack of performance of the generated code, no support for task switches in gdb, bugs in the profiling support, ….

EUROCONTROL has signed a contract with ACT-Europe in order to improve various aspects of the GNAT compiler (e.g. additional tracing features for exceptions) and to enhance most of the points mentioned above.

3 Ada83/Ada95 Differences and Incompatibilities Encountered

One of the main design factors of Ada95 was to keep a very high level of compatibility with the Ada83 version of the language (see [6] e.g. I.2, I.4 and [7]). This section explains the incompatibilities that were encountered and how they were solved. In most cases, an incompatibility between Ada83 and Ada95 can be removed by trivial adaptation of the code. In some cases, such a compatible modification is not possible, e.g. the generic formal indefinite types.

In case no compatible code can be written, two techniques have been used to still have one common source file to be handled by both the Ada83 and the Ada95 compiler: using a preprocessor, and automatic editing of the file using emacs scripts (see [8] and [9] for a description of the GNU emacs editor).

Preprocessing techniques are widely used in programming despite their drawbacks. A common problem is that the preprocessor directives make Ada source code syntactically invalid, as is the case for the preprocessor delivered with GNAT.
To preprocess our code, we use “ccf” (see [10]), a tool that provides preprocessing features by “commenting/uncommenting” the lines. Using this particular preprocessor in contrast to most others, for example gnatprep, m4, the C preprocessor or the Oracle embedded SQL preprocessor, has as advantage that the source files are still valid Ada and can thus still be handled properly by all the tools that expect “valid” Ada (code analyzers, syntax-aware editors, …).

3.1 Indefinite Generic Parameter

In our code six generic packages are instantiated with indefinite actual parameter type. As explained in [6] Appendix X.4, this incompatibility is likely to occur. To remedy this incompatibility - which was detected at compile time- required a trivial

change; add (<>) after the generic formal type name. However, this change is not compatible with our Ada83 compiler. So all the generic packages that were instantiated with unconstrained types were changed to obtain a preprocessable source as in

```
--## if language = Ada95 then
--##!type T (<>) is private;
--## else
type T is private;
--## end if
```

The ccf preprocessor can switch the sources between Ada83 and Ada95 version by commenting/uncommenting the relevant lines. Currently, the default source layout is Ada83. A preprocessing step is thus needed to compile our sources with GNAT. We plan to drop this preprocessing step when we no longer need compatibility with the Ada83 compiler.

3.2 Character Type Having 256 Values Instead of 128

This incompatibility, also characterized as likely to happen in [6] was present in one test program. The program looped over all characters in Ada 83. To continue to give the expected result, the loop was changed to explicitly loop on the first 128 character values. Note that this is the only incompatibility that was not detected at compile time.

3.3 String and Wide_String

The addition of Wide_Character and Wide_String is classified as an unlikely source of incompatibility in [6]. This unlikely incompatibility was present at multiple places (> 20) in our software due to the use of the POSIX function To_Posix_String with a string literal argument. The incompatibility, detected at compile time, was trivially avoided by qualifying the string literal with String'.

3.4 Generic Elementary Functions

The software is using some services of generic elementary functions (sin, cos, …) present in the Alsys provided package Generic_Elementary_Functions, which was not in the Ada83 standard. As these functions are now part of the Ada95 standard, we use library level renaming of generics, provided in Ada95, to resolve this issue. Preprocessing is used to avoid compiling this with the Alsys 83 compiler:

```
--## if language = Ada95
--##!with Ada.Numerics.Generic_Elementary_Functions;
--##!generic package Generic_Elementary_Functions
--##!renames Ada.Numerics.Generic_Elementary_Functions;
--## end if
```

3.5 Numeric Attributes

An obsolete numeric attribute was used in a test program that was checking boundary values. This incompatibility, detected at compile time, was solved by replacing 'Small by 'Model_Small.

3.6 Illegal Non-required Package Bodies

A few empty package bodies had to be removed. For a few packages, a pragma Elaborate_Body was added. These illegal "non-required" package bodies were detected at compile time.

3.7 Use of New Ada95 Keywords

There were a few uses of the new reserved keyword "protected". These incompatibilities, also detected at compile time, were trivially solved by changing the name of the variable or parameter.

4 Adaptation of Non-portable Code

The TACT and IFPS code has proven to be quite portable; indicated by the limited time and effort needed to port it from one compiler to another. However, some non-portable features had to be used, for example Alsys specific pragmas, or compiler/run-time/operating system specific features. The following sections list the different points at which non-portable code had to be changed or alternative code developed, either due to the migration to Ada95 or to use the new Ada tasking run-time based on operating system threads.

4.1 Alsys Specific Mutex

For performance reasons Alsys specific low level mutexes had been used to implement critical sections, instead of implementing this via a monitor task. The direct use of these non-portable mutexes was centralized inside one package body (safe_mutex.adb). It was thus very easy to implement an alternate body for Ada95.

The performance of two different versions was compared: pure Ada95, based on protected objects, and access to the low-level pthread mutex via GNAT specific packages. Implementing a mutex on top of a protected object is an abstraction inversion: we are using a high level structure to implement a low-level concept. The direct access to the low-level resource was more efficient by a factor of 6 and was, similarly to Alsys code, centralized into the safe_mutex package body.

4.2 Access to Command Arguments

Access to the command arguments was done via an Alsys specific package. An alternate implementation of this package was implemented on top of the new standard Ada.Command_Line package.

4.3 “System.Offset”

In some packages, some low-level pointer arithmetic was done (e.g. to pack some bytes into a piece of shared memory). Some Alsys specific features - Offset type and related functions – were used. These incompatibilities were solved by doing ‘renames’ using the preprocessor, renaming either the Alsys type and functions, or the new Ada95 standard features provided in System.Storage_Elements.

4.4 Heap and Memory Use Information

When suspecting a memory leak or when measuring the memory use of some algorithm, the state of the heap has to be reported. The state of the Alsys heap was reported using Alsys specific calls. The package body reporting the heap status was modified, with preprocessing, so as to have either code that reported the state of the Alsys heap or the state of the GNAT heap, which is based on the operating system memory allocation malloc library.

4.5 Pragmas

The graphical interface of the TACT and IFPS system is based on Motif. Motif widgets make intensive use of callbacks; a callback has to be defined for each action such as a button push, a list selection, etc. In Ada83, there was no portable way to define these callbacks. Hence there were a large number of uses - more than 500 - of non-portable Alsys specific or obsolete Ada83 pragmas (Call_In, External_Name, Interface, Interface_Name). Due to the number of these non-portable pragmas we did not want to use the preprocessor to solve this incompatibility. Instead, a GNU emacs lisp script [9] was written that automatically converts the old style pragmas into a set of pragmas accepted by GNAT.

Note that this automatic conversion is not switching to a pure Ada95 solution but rather to an intermediate solution (accepted by GNAT) that is halfway between the Alsys specific version and the pure Ada95 version:

Ada Calls C (e.g. Access to Motif, System Calls):

Alsys Ada83 specific:

```
function A_System_Call (Foo: Bar);
pragma Interface (C, A_System_Call);
pragma Interface_Name (A_System_Call, "system_call");
```

Ada95:

```
pragma Import (C, A_System_Call, "system_call");
```

C Calls Ada

Alsys Ada83 specific:

```
procedure Doit (Some_Thing: Some_Type);
pragma Call_In (C, Doit);
pragma External_Name (Doit, "externaldoit");
```

Ada95:

```
pragma Export (C, Doit, "externaldoit");
```

C Calls Pointer to Ada (Motif Callbacks)

Alsys Ada83

```
procedure Button_Clicked (Button: Motif.Button.T);
pragma Call_In (C, Button_Clicked);
  -- Button_Clicked callable from C
pragma External_Name
  (Button_Clicked, "screenx_buttony_click");
  -- via linkname screenx_buttony_click

procedure Button_Clicked_Entry
  (Button: Motif.Button.T);
pragma Call_In (C, Button_Clicked_Entry);
pragma Interface_Name
  (Button_Clicked_Entry, "screenx_buttony_click");

Motif.Button.Create
  (When_Click => Button_Clicked_Entry'Address);
```

Pure Ada95: elegant, type safe, no name collision e.g. in generics

```
procedure Button_Clicked (Button: Motif.Button.T);
pragma Convention (C, Button_Clicked);

Motif.Button.Create
  (When_Click => Button_Clicked'Access);
```

"Alsys-ified" Ada95 Accepted by GNAT:

```
pragma Export + pragma Import + 'Address
```

4.6 Differences Due to Tasking Run-Time Behavior

We did not encounter incompatibilities or run-time differences for “pure” use of tasking, e.g. when using monitor tasks protecting access to an object. However, some code providing access to TCP/IP uses non-trivial Unix features such as asynchronous signal based IO to avoid having the complete process being blocked when one task calls a blocking Unix system-call during TCP/IP input-output.

The first idea was to make the minimum of changes to this complex code to get it work with the new runtime. This was not too difficult but afterwards we decided to simplify the code using the thread-based Ada tasking run-time: with GNAT, Ada tasks are mapped on operating system HP-UX threads. From HP-UX 10 onwards (DCE threads) or HP-UX 11 (real kernel pthreads), a blocking system call does not block the complete process.

The package implementing the non-blocking TCP/IP layer was then simplified for the Ada95 version as each task could directly do its “own” IO system calls instead of going though a central IO task intercepting the SIGIO signal and waking up the task(s) for which IO was possible. This new IO schema based on the improved Ada tasking based on OS threads is also more efficient than the signal based IO.

4.7 Elaboration Order

During the development of the TACT and IFPS system, little attention was paid to elaboration problems, mainly because the Alsys compiler is quite good at finding a working elaboration order.

The GNAT compiler has multiple ways to handle elaboration order. The default GNAT behavior is to find a working static order. This means that at compile time, the compiler determines an elaboration order that is guaranteed to work always.

When such an order cannot be found, the GNAT binder reports this fact. Multiple options are then available to obtain an elaboration order:

- Change the code to allow a static elaboration order to be found. There are many advantages to this solution. See section 9 of the GNAT user manual [11] for a detailed explanation of elaboration handling in GNAT and the advantages of this static solution.
- Ask the compiler to try to find a “good” order. In this case, the compiler inserts elaboration checks to make sure at run-time that the order is valid.
- Tell the compiler that the programmer has taken care of elaboration order problems. This is a dangerous option to use because, in case of human error, some packages might not have been elaborated yet when used. Note that this is the standard C++ and Java approach (see the Java and C++ language specifications [12] [13]).

We had a few executables where no static elaboration order was found by the GNAT compiler. We first tried to have GNAT finding a “good” elaboration. This did not work very well for us, either because the Alsys compiler is better at finding a working order or because our code was “naturally” developed over the years to fit well with the Alsys way of choosing an elaboration order.

We then decided to change the code in order to obtain a static elaboration order. This was quite easy to do in all cases except one. What is interesting to note is that the

code changes needed to obtain a static elaboration resulted in better-structured code. When the GNAT compiler could not find a static elaboration order, in most cases this indicated that the package structure was not very good and that artificial dependencies had been introduced by this inadequate structure.

The case for which we were not able to obtain a static elaboration order was tasking code that had not been developed at Eurocontrol. We took over the support of this code due to contractual changes with the product supplier. With limited knowledge of this code and its complex structure we were unable to change it to elaborate statically. For this particular case the solution chosen was to tell the compiler that there was no elaboration problem.

4.8 By-Reference Types

We had code that needed by-reference types for memory management. If an access type value is passed mode **in out** to a procedure that deallocates it, then, if control returns to the caller via exception propagation, we do not want the caller to still have access to the dangling reference. The Alsys compiler chose a by-reference mechanism when an access value was encapsulated in a record type, while GNAT uses by-value parameter passing for such types. Luckily these access values were already encapsulated in limited private types, so we could easily use the Ada95 non-private limited type construct to force the appropriate mechanism on the full type declaration.

```
generic
  type Access_Type is private;
  -- unprotected reference type to be encapsulated
package Mem_Mgmt_G is
  type T is limited private;
  -- "managed" reference type to be re-exported
private
  type T is limited record -- becomes by-reference
    Component: Access_Type;
  end record;
end Mem_Mgmt_G;
```

5 Software Management Aspects

As explained in the introduction, one of the main factors for the CFMU at Eurocontrol is to minimize the risk related to the technology change. This is why the compatibility with Ada83 must be maintained until the code compiled with the new Ada95 compiler has enjoyed a long period of operational validation.

Also we do not want to have two separate sets of sources, one each for Ada83 and Ada95. Hence our compilation and build procedures have been modified to support the compilation by multiple compilers. Currently the source control system contains the version of the sources that can be compiled directly by the Ada83 compiler.

As long as Ada83 compatibility is kept, all developers have the option of using either the Ada83 or the GNAT compiler to compile, link and test. If the developer chooses to use the Ada83 compiler, he compiles the source files directly. To use the GNAT compiler, the developer uses a script that wraps up the compiler so that the following actions are executed automatically:

- Copy the source file into a sub-directory.
- Launch the preprocessor in order to switch from the Ada83 to the Ada95 version.
- If the source file contains pragmas then it is automatically edited by emacs to convert the pragmas to the Ada95 version.
- The GNAT compiler is then launched on the modified source file.
- In case of compilation error, the developer edits the Ada83 version.

While not completely comfortable, the above scheme worked quite well, particularly as the editing scripts and preprocessor do not modify the code much. The editing scripts have been constructed so that the line numbers stay the same. Hence compilation errors referring to preprocessed automatically edited code refer to the correct line in the Ada83 version even if the code to change was the "commented" Ada95 code.

6 Future Developments

As explained before, in a first phase, the strategy was to keep compatible code between Ada83 and Ada95. The final validation of the new compiler was deemed OK after a period of 3 months of operational evaluation (between November 2000 and January 2001). After this period the software to be put into operation in March 2001 should be stable. From then on development of the next version for March 2002 can be done without the constraints of keeping the compatibility with Ada83.

Once Ada83 compatibility is no longer needed, the source files in our source management system will be converted to the Ada95 form, so that preprocessing and automatic editing is not needed anymore when compiling with GNAT. From that point onwards, the source lines specific for Alsys or Ada83 becomes useless and can also be removed.

We also intend to increase the use of Ada95 features in the coming months. One of the first Ada95 specific features we expect to use is child units. Previously our code made a lot of use of Ada83's subunits (separate bodies). Child units provide better code structure. Also, the GNAT compilation model has not been optimized for the compilation of subunits: modifying one subunit means you have to recompile the parent unit and all the other "sibling" subunits. We hope to obtain better-structured code and faster compilation by limited restructuring to replace some subunits with child units.

We have also decided that using controlled objects can ease the development and maintenance of the system in cases where dynamic memory is heavily used, or had previously been necessary but not used due to the difficulty of avoiding memory leaks and dangling pointers.

7 Conclusion

On these large real life applications, the compatibility between Ada83 and Ada95 is very good. Where not compatible, the differences were almost always trivial to solve and were all - except one - detected during compilation. Porting from Ada83 to Ada95 was in fact a non-event that was done in parallel with normal system evolution. The progress in compiler technology visible in the Ada95 compilers - both GNAT and ObjectAda - has already brought in benefits even during the port. For example both compilers give warning messages for some dubious constructs. In some cases these warnings have revealed genuine problems in our code, or code that could be optimized.

Despite some aspects of the technology that have still to be improved, in particular the gdb debugger, the GNAT compiler and tool suite brings some major improvement and comfort in many aspects, for example compilation speed and better diagnostics. We also expect that for future developments, the increased use of Ada95 concepts will make the evolution of our systems easier and our software even more portable.

References

[1] Waroquiers, P.: Ada Tasking and Dynamic Memory: To Use or Not To Use, That's a Question!. In Alfred Strohmeier (Ed.): Reliable Software Technologies – Ada-Europe'96. Springer-Verlag, Lecture Notes in Computer Science vol. 1088, ISBN 3-540-61317-X (1996) pp. 460-470
[2] ATFM Operations Summer 2000 Report, Eurocontrol/CFMU (2000)
[3] Taft, T.S., Duff, R.A., (Eds.): Ada 95 Reference Manual: Language and Standard Libraries, International Standard ISO/IEC 8652:1995(E). Springer-Verlag, Lecture Notes in Computer Science vol. 1246, ISBN 3-540-63144-5 (1997)
[4] Ada 83 CORBA IDL Compiler. http://www.eurocontrol.fr/projects/arh-spv_ext/comp/ (1996)
[5] Object Management Group: Ada Language Mapping Specification. Edition June 1999, http://www.omg.org/technology/documents/formal/ada_language_mapping.htm (1999)
[6] Barnes, J., (Ed.): Ada 95 Rationale: The Language, The Standard Libraries. Springer-Verlag, Lecture Notes in Computer Science vol. 1247, ISBN 3-540-63143-7 (1997)
[7] Bill Taylor: Ada Compatibility Guide, Version 6.0, http://www.adaic.com/AdaIC/docs/compat-guide (1995)
[8] Richard M. Stallman: GNU Emacs Manual. Thirteenth Edition (1997)
[9] Bill Lewis, Dans LaLiberte and the GNU Manual Group: The GNU Emacs Lisp Reference Manual. Edition 2.5 (1998)
[10] CCF: a Conditional Compilation Facility. http://users.swing.be/imw/ccf/ (2000)
[11] Ada Core Technologies: GNAT User's Guide. Document revision level 1.317 (2000)
[12] Java Language Specification. The part where something like elaboration is mentioned (Static Initializers): http://www.javasoft.com/docs/books/jls/html/8.doc.html#39245 (1996)
[13] C++ Language Standard: ISO/IEC 14882:1998. http://www.ncits.org/cplusplus.htm (1998)