46431

p. 18

Ada Issues in Implementing ART-Ada

(NASA-CR-188941) Ada ISSUES IN IMPLEMENTING ART-Ada (Research Inst. for Advanced Computer Science) 18 p CSCL 09B

N91-32838

Unclas

G3/61 0046431

S. Daniel Lee

Inference Corporation

November 1990

Cooperative Agreement NCC 9-16
Research Activity No. SE.19

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division

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

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

Ada Issues in Implementing ART-Ada

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Inference Corporation. Dr. Charles McKay served as RICIS research coordinator.

Funding has been provided by the Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Robert T. Savely, of the Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

Ada Issues in Implementing ART-Ada*

S. Daniel Lee

Inference Corporation
550 North Continental Boulevard
El Segundo, California 90245
Email: lee@inference.com

Abstract

Due to the Ada mandate of such government agencies as DoD, NASA and FAA, interest in deploying expert systems in Ada has increased. Recently, several Ada-based expert system tools have been developed. According to a recent benchmark report, these tools do not perform as well as similar tools written in C. While poorly implemented Ada compilers also contribute to the poor benchmark result, some fundamental problems of the Ada language itself have been uncovered. In this paper, we describe Ada language issues encountered during the development of ART-Ada, an expert system tool for Ada deployment. ART-Ada is being used to implement several prototype expert systems for the Space Station Freedom and the U.S. Air Force.

1. Introduction

As the government mandate to standardize on Ada as the language for software development is being actively enforced by government agencies, including DoD, NASA and FAA, interest in making expert systems technology readily available in Ada environments has increased. An example project that exhibits the need for expert systems in Ada is NASA's Space Station Freedom. Another large-scale application of Ada-based expert systems is the Pilot's Associate (PA) project for military combat aircraft [Hugh 88].

Recently, several Ada-based expert system tools are developed to address this need of government agencies. Since some of these tools were originally implemented in C, they are based on the same algorithm as that of C-based tools. According to a benchmark, Ada-based tools do not perform as well as C-based ones. While poorly implemented

*This paper will appear in the proceedings of third Annual NASA Ada User's Symposium, Houston, Texas, November 1990.

Ada compilers also contribute to the poor benchmark result, some fundamental problems of the Ada language itself have been uncovered.

In this paper, we describe Ada language issues encountered during the development of ART-Ada, an expert system tool for Ada deployment [Lee & Allen 89], [Lee & Allen 90a], [Lee & Allen 90b]. ART-Ada allows applications of a C-based expert system tool called ART-IM to be deployed in various Ada environments. While ART-IM's inference engine was reimplemented in Ada, ART-IM's front-end (its parser/analyzer and graphical user interface) was reused as the ART-Ada development environment. The ART-IM kernel was enhanced to generate Ada source code that would be used to initialize Ada data structures equivalent to ART-IM's internal C data structures, and also to interface with user-written Ada code. Once the development is complete, the application is automatically converted to Ada source code. It is then compiled and linked with the Ada runtime kernel, which is an Ada-based inference engine. The overview of ART-Ada is depicted in figure 1-1. ART-Ada is being used to implement several prototype expert systems for the Space Station Freedom and the U.S. Air Force.

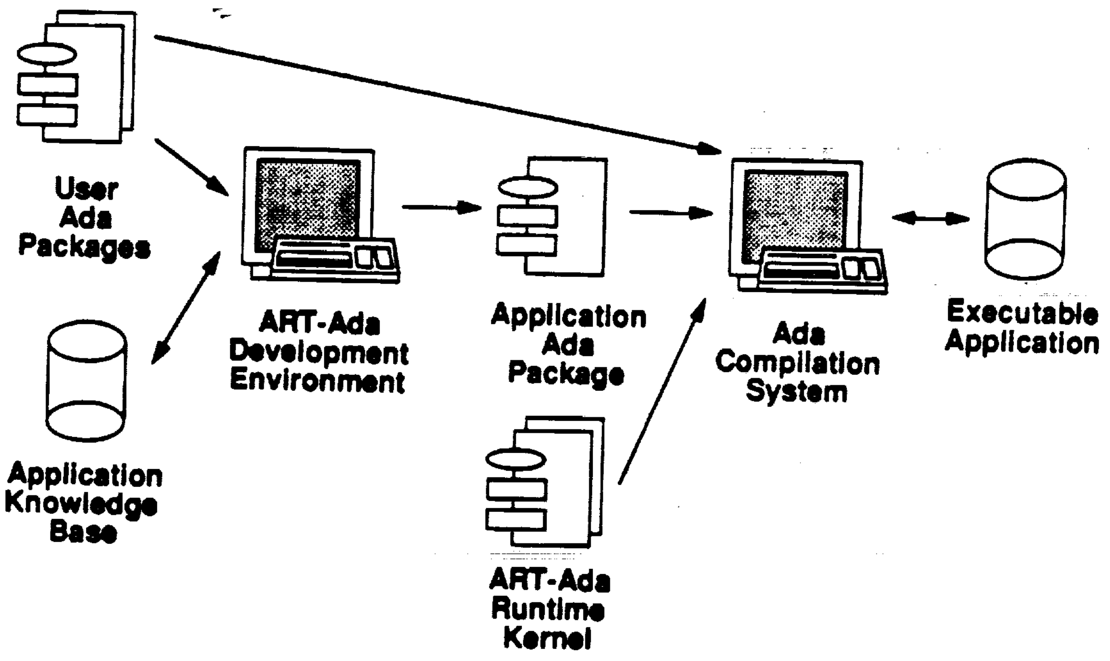

Figure 1-1: Overview of ART-Ada

While Ada compilers are improving, they still have not reached the maturity of C compilers. In fact, because of numerous bugs found in the Ada compilers used for this project, we could not make some of the obvious performance optimizations that could have made ART-Ada faster and smaller. It has also been observed that both the speed and size of ART-Ada vary up to 30% depending on which Ada compiler is used. A recent paper discusses the key technical issues involved in producing high-quality Ada compilers [Ganapathi 89]. As Ada compiler technology advances, ART-Ada's performance will improve.

In addition to the compiler problems, we also discovered some fundamental issues with the Ada language itself that also affected the performance of ART-Ada. Various Ada language issues are being studied by the Ada 9X Project team. We believe that the issues discussed in this paper should also be considered for the Ada 9X standard. In fact, they have been presented to several members of the Ada 9X Project in a meeting held in Washington, D.C. in March, 1990.

2. Compiler Problems

Several reports from Ada compiler vendors indicate that some Ada programs might run faster than the equivalent C programs. Contrary to these claims, our Ada implementation is slower and larger than the C implementation. Although we believe the main reason is the restrictive nature of the Ada language itself, Ada compiler bugs also contribute to the poor performance. We used the Verdix Ada compiler on a Sun workstation and the DEC Ada compiler on a VAXstation running the VMS operating system.

- The bit-level representation clause or the pragma pack can be used to reduce the size of data structures. For example, a boolean field in a record, which is normally a byte, can be reduced to a single bit. These features did not work in one of the compilers we used: an illegal instruction error occurred when the single-bit boolean field was referenced. This is probably a bug in the code generator. Due to this bug, no attempt was made to reduce the size of ART-Ada by using these features.

- In ART-Ada, we reuse several Booch components [Booch 87]. These software components are used to implement data structures (e.g. linked lists and strings) and other utilities (e.g. quick sort). Most Booch components are implemented as generic packages using object-oriented design methodology. This means that a large number of subprograms are provided in each generic package, which may be instantiated multiple times. Unfortunately, one of the compilers does not support a feature called *selective linking* --- a linker feature that makes it possible to include only those subprograms actually used in the program. The underlying mechanism used by the compiler is the Unix linker (ld), which does not support selective linking. As a result, whenever a generic package is instantiated and included using the *with* statement, all subprograms in the package are always included in the executable image regardless of their actual usage. This increases the size of the executable image.

- We could not use an optimizer in one of the compilers because it generated bad code.

3. Dynamic Memory Allocation

Due to the dynamic nature of expert systems, it is necessary to allocate memory dynamically at runtime in ART-Ada. In Ada, *new* is used to allocate memory and *unchecked_deallocation* is used to deallocate it. Our experiment shows that the average overhead of *new* in the Verdix compiler is about eighteen bytes, i.e. every time *new* is called, an extra eighteen bytes are wasted. This result is obtained by using a program that allocates the same data structure multiple times using *new* and measuring its process size with the Unix command "ps aux". We repeated the same experiment using several data structures of different size. According to Verdix, *new* eventually calls *malloc*. We tried similar experiments using the Sun C compiler. The average overhead of *malloc* was about eight bytes, which was significantly smaller than that of Ada. It is not clear why it is necessary to add extra ten bytes to every *malloc*. The only information needed to call *free* is the size of the memory, which can be obtained from the data type used to instantiate the generic procedure *unchecked_deallocation*. The exceptions are unconstrained arrays and variant records whose size can vary. For these data types, it would be necessary to add four bytes to store the size information. The actual measurement results are summarized in Tables I-1 and I-2 in Appendix I. Units in these tables are bytes. The C and Ada program used are shown in the Appendix II.

The real problem with this overhead is that in ART-Ada *new* is called very frequently to allocate relatively small blocks while in ART-IM (Inference's C-based expert system tool), *malloc* is called only to allocate large blocks (e.g. 100 Kbytes). In order to achieve maximum time and space efficiency, ART-IM has been optimized in ways that are not portable to Ada. For example, the type cast feature of the C language has been used both to optimize data structures and to implement an internal memory manager. ART-IM's memory manager maintains its own free lists and handles all allocation and deallocation requests from the ART-IM kernel; it allocates large blocks of memory from the system, and then fulfills individual (relatively small) requests for storage from the large blocks. As storage is released, it is added to internally maintained free lists; the blocks themselves are never released back to the system. There are several advantages to this approach:

- The free space is managed in a common pool by a single facility and is available for allocation of arbitrary data types by using the type cast capability in C.

- The overhead of this approach consists of a fixed overhead and a very small incremental overhead for each large block. The fixed overhead is 1 Kbyte. Internally, all small blocks freed from ART-IM are maintained in free lists. There are 256 free lists, each of which holds memory blocks with different sizes. All blocks in a free list are of the same size. The head of these linked lists consumes 4 bytes. Therefore, the total overhead to maintain these linked lists is only 1 Kbytes. The subsequent items in these linked lists store the next pointer within the small block itself, which results in absolutely no overhead. When a large block (e.g. 100 Kbytes) is allocated from the operating system, it is maintained in a linked list. Each item in this linked list consumes 12 bytes, and therefore the overhead is only 12 bytes per every 100 Kbytes, which is negligible.

- It is faster than using system routines for small requests.

The success of ART-IM's use of type casting relies on other features of the C language definition: there is a direct correspondence between addresses and pointer types; the mapping between data types, including structures and arrays, is well defined and straightforward. Ada does provide a facility for converting between data types, although this feature has intentionally been made difficult to use. In order to convert from one data type to another, the generic function *unchecked_conversion* must be instantiated for each conversion required. The implementation of a type cast capability in Ada is insufficient to implement the ART-IM features described above, however. No correspondence is guaranteed between the type SYSTEM.ADDRESS and Ada access types. Indeed, on some implementations the underlying representation is different for addresses and access types. The constraint checking requirements of Ada require that the representation of many objects include descriptor information. The format of these descriptors is not defined by the language. Hence, it is impossible to implement the ART-IM style memory manager in Ada using *unchecked_conversion.*

Another related problem was how to port C code similar to the one shown below to Ada. In this example, the & operator is used to resolve the pointer reference at compile time through the static array initialization. C code similar to this example is used to convert the ART-IM internal data structures into C source code.

```
struct foo {
  long *bar_ptr;
};

struct bar {



};

struct bar bar1[10] = { ... };

struct foo foo1[10] = {
  {&bar1[5]},  /* foo1[0] points to bar1[5] */



};
```

There are two problems in implementing this in Ada:

- As mentioned earlier, *unchecked_conversion* is not as flexible as the & operator.

- Even if it is possible to emulate the & operator with *unchecked_conversion*, it is not possible to free these data structures using *unchecked_deallocation* because they are not created dynamically through *new*.

As a consequence, we had to create all data structures dynamically using *new*. To resolve the pointer references, we used the following method:

1. When a data structure is created, its pointer value returned by *new* is stored in a temporary pointer array.

2. When a data structure has a pointer reference, the index of the temporary pointer array and the data type of both referencer and referencee are stored in a cross reference table for later processing.

3. After all data structures are created, the cross reference table is processed. The actual pointer value is fetched from the referencee pointer array and stored in the referencer.

4. After all pointer references are resolved, the temporary pointer arrays and the cross reference table are freed.

The disadvantage of this approach is that large blocks of memory must be allocated and freed at runtime. The size of the cross reference table could be quite large. In fact, we could not use the 16-bit integer as an array index because it overflowed on a large test case.

The problems of dynamic memory allocation in Ada can be summarized as follows:

- The direct use of *new* and *unchecked_deallocation* is the only dynamic memory management method available in Ada. The problem with this method is that *new* incurs a fixed overhead associated with each call and it is called very frequently to allocate a relatively small block for an individual data structure. It results in a performance penalty in size and the slower execution speed. This is also aggravated by the poor implementation of *new* in the Ada compiler.

- The existing Ada features, *new*, *unchecked_deallocation*, and *unchecked_conversion*, are too restrictive and totally inadequate for a complex system that requires efficient memory management. More flexible features (perhaps in addition to the existing ones) should be provided. This is particularly important in embedded system environments that impose a severe restriction on the memory size.

4. Other Language Issues Related to Performance

The issue of dynamic memory management is, we believe, by far the dominant factor for the overhead in ART-Ada performance compared with that of ART-IM. Other issues in the Ada language that also contribute to the overhead are summarized below:

- ART-IM has an interpreter (similar to a Lisp interpreter) that calls a C function using a C function pointer. To emulate ART-IM's function call mechanism, the Ada code generator automatically generates Ada source code for a procedure called FUNCALL that has a large case statement. This case statement contains all the Ada subprograms that are called from an ART-Ada application. Each subprogram is assigned with an ID number. To call an Ada subprogram, the procedure FUNCALL is called with a subprogram ID number. While it may cause maintenance problems, the use of function pointers can provide better performance than the use of the Ada case statement.

- Bit operations (e.g. bitwise exclusive OR, bitwise shift operations, etc.) that may be used to implement efficient hashing algorithms are not provided in Ada. They may be implemented in Ada but only with poor performance.

- Ada strings are stored in a record with a length field in ART-Ada. A generic string package from the Booch component library is used for internal string storage and manipulation [Booch 87]. Since STANDARD.STRING is used for all public interfaces, a Booch string is converted to

STANDARD.STRING or vice versa. It would be more efficient if the standard Ada string is one with a length specification that can be manipulated easily using a set of predefined standard string operations.

5. Portability

Although Ada is quite portable (probably more portable than C), Ada is not 100% portable.

- Since the development environment of ART-Ada is written mostly in C, an Ada binding is developed to interface it with Ada. We found it extremely hard (if not impossible) to write portable binding code for multiple compilers running on multiple platforms. The pragmas for importing and exporting subprograms are not portable. The parameter passing mechanism between Ada and C is not standardized. Because of this, a mechanism for string conversion between Ada and C is not portable.

- The standard syntax for most pragmas are not defined in the Ada Language Reference Manual. Consequently, the pragma syntax often varies among different compilers.

- No standards exist for INTEGER, FLOAT, LONG_INTEGER, LONG_FLOAT, SMALL_INTEGER, SMALL_FLOAT, etc. ART-Ada supports 32-bit integers and 64-bit floats internally. We had to define INTEGER_TYPE and FLOAT_TYPE as subtypes of whatever a compiler defines as such. For example, in the Verdix compiler STANDARD.FLOAT is 64-bit while in the DEC compiler STANDARD.LONG_FLOAT is.

- Since the math library, which is part of the standard C language, is not part of standard Ada, it is hard to write portable Ada code that uses math functions.

- The representation clause is not portable because different Ada compilers and hardware platforms may use a different memory boundary.

- Some code is simply not portable. For example, in ART-Ada, a public function is provided to invoke the operating system commands. Obviously, the implementation of this function is not portable among different operating systems.

- Different Ada compilers or even different versions of the same compiler often have a different set of bugs. It may be necessary to maintain multiple versions of the same code to work around them.

In C, conditional compilation facilitated by preprocessor directives (e.g #define and #if) allows maintaining a single source file for multiple platforms. In Ada, no such facility exists, and multiple files may have to be maintained for multiple platforms. Since we had to maintain ART-Ada on multiple platforms (possibly on multiple compilers on the same hardware), we did not want to maintain multiple files. At first, we were going to write a preprocessor in Ada or in C. After some experiments, however, we found the C preprocessor (cpp) on a Sun quite adequate for preprocessing the Ada master file with cpp macros embedded (e.g. #if, #endif, etc.).

The master file includes Ada code and appropriate cpp commands for multiple platforms :

```
#if VERDIX
   subtype FLOAT_TYPE is FLOAT;
#endif

#if VMS
   subtype FLOAT_TYPE is LONG_FLOAT;
#endif
```

We define app as follows:

```
/lib/cpp $1 $2 $3 $4 $5 $6 $7 $8 $9 | grep -v "^#"
```

Then, we execute the following commands:

```
app -DVERDIX foo.a.master > foo.a
app -DVMS foo.a.master > foo.ada
```

The first one creates a file for the Verdix compiler on a Sun, and the second, for the DEC Ada compiler on a VAX/VMS.

The problem with this is that the Ada master file is still not a compilable Ada file and has to be preprocessed manually. We also have to maintain multiple Ada files generated by cpp. It would be better if the preprocessor is part of the standard Ada language so that only a single source file is maintained and processed directly by the Ada compiler.

6. Acknowledgments

The author wishes to acknowledge the guidance and support of Chris Culbert and Bob Savely of NASA Johnson Space Center, Greg Swietek of NASA Headquarters, and Captain Mark Gersh of the U.S. Air Force. Brad Allen, Mark Auburn, and Mike Stoler of

Inference Corporation contributed to the project. Don Pilipovich and Mark Wright who were formerly with Inference Corporation also contributed to the project.

References

[Booch 87] Booch, G.
Software Components With Ada.
Benjamin/Cummings Publishing, 1987.

[Ganapathi 89] Ganapathi, M., Mendal, G.O.
Issues in Ada Compiler Technology.
Computer 22(2), February, 1989.

[Hugh 88] Hugh, D.A.
The Future of Flying.
AI Expert 3(1), January, 1988.

[Lee & Allen 89] Lee, S.D., Allen, B.P.
Deploying Expert Systems in Ada.
In *Proceedings of the TRI-Ada Conference.* ACM, October, 1989.

[Lee & Allen 90a]
Lee, S.D., Allen, B.P.
ART-Ada Design Project - Phase II, Final Report.
Technical Report, Inference Corporation, February, 1990.

[Lee & Allen 90b]
Lee, S.D., Allen, B.P.
ART-Ada: An Ada-Based Expert System Tool.
In *Proceedings of the Space Operations, Applications and Research Symposium (SOAR).* NASA, June, 1990.

ORIGINAL PAGE IS
OF POOR QUALITY

I. Memory Allocation Benchmark Results

Item Size	Item Count	Ideal Size	Actual Size	Overhead	Overhead/Item
8	100,000	800 K	2496 K	1696 K	16.96
16	100,000	1600 K	3312 K	1712 K	17.12
24	100,000	2400 K	4128 K	1728 K	17.28
32	100,000	3200 K	4808 K	1608 K	16.08
8	50,000	400 K	1408 K	1008 K	20.16
16	50,000	800 K	1816 K	1016 K	20.32
24	50,000	1200 K	2224 K	1024 K	20.48
32	50,000	1600 K	2496 K	896 K	17.92
Average	N/A	N/A	N/A	N/A	18.29

Table I-1: Overhead of Dynamic Memory Allocation using new in Verdix Ada

Item Size	Item Count	Ideal Size	Actual Size	Overhead	Overhead/Item
8	100,000	800 K	1600 K	800 K	8.0
16	100,000	1600 K	2384 K	784 K	7.84
24	100,000	2400 K	3160 K	760 K	7.60
32	100,000	3200 K	3944 K	744 K	7.44
8	50,000	400 K	816 K	416 K	8.32
16	50,000	800 K	1208 K	408 K	8.16
24	50,000	1200 K	1600 K	400 K	8.0
32	50,000	1600 K	1922 K	392 K	7.84
Average	N/A	N/A	N/A	N/A	7.9

Table I-2: Overhead of Dynamic Memory Allocation using malloc in Sun C

II. C and Ada Programs for Memory Allocation Benchmark

```
#include <stdio.h>

main()
{
  int i;

  for(i=0; i<100000; i++) {
    malloc(32);
  }
  getchar();  /* measure the process size at this point */
}
```

```
with TEXT_IO;
procedure TEST_NEW is

  type ELEMENT is
    record
      FIELD1 : INTEGER;  -- 4 byte
      FIELD2 : INTEGER;  -- 4 byte
      FIELD3 : INTEGER;  -- 4 byte
      FIELD4 : INTEGER;  -- 4 byte
      FIELD5 : INTEGER;  -- 4 byte
      FIELD6 : INTEGER;  -- 4 byte
      FIELD7 : INTEGER;  -- 4 byte
      FIELD8 : INTEGER;  -- 4 byte
    end record;

  type ELEMENT_PTR is access ELEMENT;
  PTR : ELEMENT_PTR;
  CHAR : CHARACTER;

begin
  for I in 1..100000 loop
    PTR := new ELEMENT;
  end loop;
  TEXT_IO.GET(CHAR);  -- measure the process size at this point
end;
```