1989

Ada and the graphical kernel system

Richard R. Wessman

Rochester Institute of Technology
School of Computer Science

Ada and the Graphical Kernel System

by
Richard R. Wessman

A thesis, submitted to
The Faculty of the School of Computer Science,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

9/22/88
Professor Andrew T. Kitchen

26 Sept 88
Professor Peter G. Anderson

9/28/88
Professor Guy Johnson

September 12, 1988

Title of Thesis: Ada and the Graphical Kernel System

I, Richard R. Wessman, hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

July 11, 1989
~~May 15,~~ 1989

_____

TABLE OF CONTENTS

Abstract

Since the introduction of the standard language in 1983, Ada has been the subject of controversy about its capabilities and its desirability. Proponents have claimed that Ada may be the ultimate computer language because it is a high-level, structured language that may be used in applications ranging from controlling jet fighters to business.

Detractors have argued that Ada's very versatility may be its undoing because it can do many things but not as well as more specialized languages.

This thesis attempts to evaluate the capabilities and suitability of Ada for large projects by using it to implement the Graphical Kernel System.

1. Project Goals

The goal of this project was to investigate the suitability of Ada for large projects. Ada has been claimed to be the "language of the 80's." It has been claimed that Ada can be used in place of everything from assembly language to COBOL. One of the claims made is that the modularization abilities of Ada make it easier to build large projects. GKS seemed to be a good project to test that claim. Limited as it is, my implementation is still over 15,000 lines long. Another of the claims made frequently is that the strong typing and the information-hiding of Ada facilitate programming and reduce errors found at execution time. The complexity of GKS seemed to make it a good candidate to prove or disprove this claim. Another reason why I picked GKS for this project is that it contains both application-level routines and device interfaces. I figured that it would be a good test of the versatility of Ada.

1.1. Introductions

Ada

Ada was conceived by the United States Department of Defense in 1974 as a means to reduce the cost of software related to the development and maintenance of embedded computer systems. Embedded systems are used in real-time applications such as planes and guided missiles.

Because embedded systems need to handle errors particularly well, e.g., a computer controlling a jet fighter on descent cannot shut down because one process reports an error, Ada was designed expressly to manage errors.

Another major reason why Ada was developed was to reduce the costs associated with having many versions of the same program written in the different languages. Frequently, when a developer would leave, much time would be required to train his or her successor in the particular, often esoteric, language in which the application was written.

Yet another major reason was to reduce the costs of documentation. Ada was to do this by having clear and consistent syntax.

GKS

GKS, the Graphical Kernel System, was developed in 1981 to be the first international standard for programming computer graphics applications. It was created so that applications could be written independent of the hardware used in a particular application.

According to Enderle [ENDER], the main reasons for introducing the standard were

- to allow application programs involving graphics to be easily portable between different installations;
- to aid the understanding and use of graphics methods by application programmers;
- to serve manufacturers of graphics equipment as a guideline in providing useful combinations of graphics capabilities in a device.

GKS consists of a language binding, a standard function interface, an interface between GKS and the device drivers, and an interface between an interactive workstation and an operator.

1.2. Project Limitations

This project was not meant to be a test of GKS or to be a complete graphics package. GKS was only picked as a vehicle to use in investigating Ada.

GKS has many forms of input: locator, pick, stroke, string, and valuator. In order to keep the size of the project down, I decided to implement only string input. However, I did put in the interfaces for other forms of input, down to the interface to the device level.

Also, in order to keep the size of the project manageable, I deleted metafiles from this project. Metafiles are text files that GKS produces to hold instructions from one invocation to another. From the standpoint of investigating Ada, to implement metafiles would not have added an important dimension to the project, in my opinion. However, from the standpoint of the user, metafiles would make the graphics package more usable.

2. Literature Search

2.1. Introduction

The purpose of this literature search was to investigate other bodies of work involving the evaluation of Ada.

To collect the needed references, various databases and abstracts were used, including INSPEC and the Computing Surveys published by the Association for Computing Machinery.

To find the articles that resulted from the data base and abstract searches, the libraries of the Rochester Institute of Technology, the University of Rochester, and Computer Consoles, Incorporated were used. The majority of articles were located at RIT.

The resulting literature has been divided into six classes. They are: general evaluations of Ada, implementations of various projects in Ada, critiques of specific facets of Ada, comparisons of Ada against other languages, conversions of programs written in other languages to Ada, and implementations of GKS in Ada.

Because the area covered in Ada literature is so wide, some of the articles overlap in content. This was unavoidable, and every effort has been made to prevent this from occurring.

This review is not meant to be exhaustive. To list and review all of the literature written about Ada would require a thesis in itself. Instead, a representative selection has been culled and evaluated.

2.2. General Evaluation

Most of the general evaluations of Ada concentrated on the debates concerning the unique aspects of Ada.

In "Why Ada Is Not Just Another Programming Language" [SAMME], Jean Sammet argues that

> Ada is unique due to sociological, economic, and political reasons - non-technical issues not traditionally applicable or considered heavily in the evaluation of other languages.

Sammet goes on to detail the development of Ada, its test and evaluation, its control by the Department of Defense, its development environment, its uses as a design language, its uniqueness technically, among other things.

This article is representative of many others that I found. It goes into some technical detail, but mainly talks in generalities about various aspects of the language.

The next two articles in this sample typify another type of evaluation in which general portions of the language of the language are enumerated and criticized or defended.

In "Scaling Down Ada (or Towards a Standard Ada Subset)" [LEDGA], Henry F. Ledgard and Andrew Singer enumerate reasons why the size of the language should be reduced.

For example, they argue that if the size of Ada were reduced:

> Teaching is simpler. The successful teaching of a new language to potential users is vital to is acceptance. When a language is large, the development of good tutorials is increasingly difficult. If the entire language is to be covered, the tutorial can become so long that even its physical length is a deterrent to its use. If only a portion of the language is covered, there is the question of which features to exclude. When features are excluded, the student may be left wondering if, in fact, there isn't something that must be learned that might be vital to a problem.

In "Is Ada Too Big? A Designer Answers the Critics" [WICHM], Brian A. Wichmann answers that question with:

> It is, of course, easier to teach a smaller language. But how can one teach the principles of concurrency with a sequential language? Many academics welcome Ada because they can use it to demonstrate concepts such as abstract data types, concurrency, and error-handling within the context of a single language.

Ledgard, Singer, and Wichmann carry on this debate about many other points. Interestingly enough, the two articles appeared exactly two years apart in the same publication.

Another sort of debate occurred in "Is Ada an Object Oriented Language?" [TOUAT] Herve Touati carries on two fictional debates in which Socrates argues with a pupil. In the first debate, Socrates argues that Ada is, in fact, object-oriented. In the second, he argues the reverse.

2.3. Implementations

Because Ada is such a large language and includes many other facets besides just the actual language, the variety of articles that I found was wide-ranging.

In "Engineering VAX Ada for a Multi-language Programming Environment" [MITCH], Charles Z. Mitchell describes the utilization of Ada in an environment where many languages are being used and combined. He goes on to discuss the VAX environment where libraries written in other languages can be used in programs written in different languages. Specifically, he describes the integration of Ada with VAX/VMS[1] Common Language Environment.

By contrast, Norman H. Cohen discusses in his article "Four Uses of Derived Types, and a Complication" [COHEN], the uses of derived types in various packages. He also describes the problems that he encountered in the process. He concludes the derived types are useful.

In "Development and Implementation of the Magnavox Generic Ada Basic Mathematics Package" [REHME], Karl A. Rehmer describes the process of designing and implementing of a mathematics library for Ada, something that is left to be implementation-defined. He discusses both the positive and negative aspects of Ada.

For yet another example, Karl A. Nyberg, in "Using Representative Clause as an Operating System Interface" [NYBER], describes his implementation of software for producing billing invoices using data collected by the Ultrix[2] operating system.

He discusses the difficulties that he encountered in implementing the software, including the disappointing performance of the program. He also discusses the good points of the implementation.

---

[1] VMS is a registered trademark of Digital Equipment Corporation.

[2] Ultrix is a registered trademark of the Digital Equipment Corporation.

2.4. Specific Critiques

As it was in the case of the previous categories, there were many kinds of critiques. Some dealt with specific features of the language, while others dealt with the environment that surrounds Ada.

As an example of the latter, G. Vittorio Frigo evaluates the Ada Programming Support Environment (APSE), in "Evaluation of the VAX Ada Compiler and APSE by Means of a Real Program." He uses a taxi simulation to test the tasking facilities of Ada. He also utilizes it to test the Ada compiler and debugger, and to evaluate the documentation.

The environment surrounding Ada is the subject of much literature because it is integral to Ada.

An example of a critique of a specific part of Ada can be found in "Overloading in the Ada Language: Is it too restrictive?" The authors use a typical complex arithmetic package to argue that the prohibition against the overloading of the ":=" and "=" operators is too inflexible. They also argue that type and function names should be overloaded, as well.

As another example, Piotrowski criticizes Ada for not allowing enough information hiding [PIOTR]. He argues that because data cannot be held in a function or procedure between invocations, it is impossible to hide information completely from the outside.

In another realm, the performance effects of Ada are analyzed by Sarkan and Wong in "Impact of Ada Features on Real-Time performances" [SARKA]. They describe the translation of a JOVIAL program into Ada. They continue on the describe the adverse impact of Ada on performance and memory use.

2.5. Comparisons with Other Languages

There were also several types of comparisons found. One type had to do with the reliability of Ada programs versus those of other languages. Another was more a "cookbook" listing of the differences between Ada and other languages. The last kind treated the implementation of a project in Ada as opposed to another language.

An example of the first type can be found in "An Empirical Study of FORTRAN and Ada" [GOEL] by Goel et al. The authors studied the implementation of an anti-missile system by six programmers. Three did it in Ada, the other three implemented it in FORTRAN. They found that Ada programs had about seventy percent fewer errors of all types than FORTRAN programs.

An example of the second type can be found in "A Comparison of the Computer Languages Pascal, C, Lisp, and Ada" [MARZJ], by Marteza Marzjanni. This article compares the four languages in terms of parameter transmission, block structure (or lack thereof), data types, encapsulation of data, referencing environment, type checking, among other things.

An example of the third type can be found in "ADA and NIL: Two Concurrent Languages for GKS" [MILAN], in which Milanese describes the implementation of GKS in Ada and NIL. NIL is a language designed by IBM in order to implement a variety of large projects on different machines. The article concentrates on the asynchronous aspects of GKS and how Ada and NIL can both be used to implement them. It also concentrates on modularity and data safety.

2.6. Conversion of Other Languages to Ada

This area, as well, contained different types of articles. There were general articles describing the conversion of programs that were not written in Ada to Ada. There were also articles detailing specific conversion of non-Ada programs to Ada. In addition, there was an article about automatic language converters. In his paper, "Non-Ada to Ada Conversion" [MARTI], Donald G. Martin discusses the problems inherent in translating programs to Ada. He concludes that because Ada is designed for top-down implementation and most other languages are not, a strict translation is probably not possible. Instead, a careful rethinking will be needed of the entire design.

P.J.L. Wallis, in "Automatic Language Conversion and its place in the Transition to Ada" [WALLI], discusses the possibilities for converting programs written in other languages to Ada using some sort of converter. He uses specific differences between Ada and other languages to conclude that while automatic conversion is possible, the difficulties involved and the slow, inefficient code produced do not make it useful as an alternative.

In "Design Experience with Ada versus FORTRAN" [SHOCH], David D. Shochat describes a study which attempted to find areas where FORTRAN is traditionally used and to determine the effect of Ada, instead.

He describes the conversion of a targeting program into Ada from FORTRAN and then proceeds to compare the two. The study concludes that Ada is more maintainable while FORTRAN is more efficient.

2.7. Ada and GKS

Two articles were found that dealt directly with implementing GKS in Ada. Thomas M. Leonard discussed his in "Ada and the Graphical Kernel System" [LEONA]. Kathleen Gilroy described her implementation in "Experience with Graphical Kernel System" [GILRO].

These two implementations were very similar to each other, which was not surprising because Leonard used Gilroy as a source. What was somewhat surprising, though, was that the two implementations were quite similar to my own. My surprise was occasioned by the fact that I found these articles after my implementation was complete.

The structures of the implementations were very similar. Each was layered and attempted to hide the device-dependent details of the implementation from the user. Leonard's implementation was closer to mine in that both used a workstation manager to route output to the various workstations. Since Gilroy's only supported one workstation, hers lacked this feature.

The primary difference was the level of GKS that was implemented. Gilroy's implementation was very basic, level 0a. Basically, a level 0a implementation only supports string input, only one workstation, and no segmentation. Leonard implemented supported GKS in its entirety. The level of my implementation, 2a, lies roughly in the middle. Like that of Gilroy, my implementation only supports string input, but it also supports multiple workstations and segmentation as does Leonard's. However, my implementation is not designed for a distributed environment whereas Leonard's is. Leonard's implementation was designed to be multi-tasking, unlike that of Gilroy and mine.

The data types employed in the other implementations differed in minor ways from those used in my implementation. All three implementations used derived types of integers and floating point so that the predefined operators were available. The two other implementations used visible types for those utilized for input devices, where mine used private types. All three bindings employed private types for list manipulations.

Each implementations mapped each GKS function one-to-one to an Ada function or procedure. Gilroy's implementation used the GKS function name as its Ada name, as does mine. Leonard did not state his naming convention.

As to error handling, the methods used differed. Gilroy used a queuing mechanism to offset not knowing were an error originated. Leonard propagated exceptions back to the user directly. My implementation used an exception handler which stamped the error into a file, which is the closest to the standard.

The packaging methods used in the three implementations were similar. Gilroy used a single package, named GKS, along with a generic list package, a generic coordinates package, and a generic list package. Leonard used two packages. One, GKS_TYPES, contained the various types used in the implementation. The other, GKS_xx, where **xx** corresponded with the level of GKS to be used, contained the functions and procedures associated with that level. The levels of GKS that Gilroy and my implementations supported were both fixed and the respective level of each, where in Leonard's implementation, the used could pick the level that he or she wanted to use.

As far as comments regarding the experiences during their respective implementations, Leonard actually had very few, and those that he had were general in nature. He did comment that packages were "probably the single most important language feature" used in the implementation. The packaging mechanism allowed different applications to use different levels of GKS. He also commented that

> The multiple workstation feature of GKS is also easily accommodated with Ada packages. The implementation of a particular level of GKS may support various workstations. To gain access to the Workstation Driver which supports a particular device, the Workstation Manager merely "with"'s the package which provides the interface to the workstation. In this way the system is easily reconfigured for new devices.

He concludes by stating that

> The implementation of GKS/Ada has proved that Ada is a very natural language for implementing systems outside of the embedded systems environment. Ada's package and generic features proved most useful in creating software which was reusable both within a particular implementation of GKS, and among implementations of GKS at different levels. The multi-tasking features of the language provided very natural solutions to the GKS Event Queue and multiple open workstations. This implementation of GKS/Ada accommodates distributed graphics environments, is easily configurable, and portable to any environment supporting a validated Ada compiler.

The description that Gilroy gives of her experience, was, on the other hand, very specific. For example, she described in detail, how the GKS data types were mapped to Ada.

In many ways, her experiences paralleled mine. As to the complier, the

> Telesoft Ada compiler implementation which we used did not support the full capabilities of the Ada language. This presented several problems during development of the prototype system, but the implementation was complete enough to allow us to develop a fairly good working subset of the graphics system.

She also encountered problems with unsupported types and limitations on the size of the symbol table.

> Other data typing facilities which were needed but not supported included integer type definitions, derived types, record typing involving discriminants, and array aggregates. We generally relied on the predefined data types and statically constrained arrays to work around the problems, although the semantics of their use was inconsistent with the intention of the binding specification. Limitations on symbol table size also presented problems in attempting to compile GKS as a single package. We ended up dividing GKS into five separate packages. One package contained all of the GKS interface data types, and each of the others contained a subset of the GKS functions.

These difficulties are similar to those that I had with the Verdix compiler. It ran out of room if too many generic packages were instantiated.

Another problem with which she had to deal was the lack of precision of exception handling.

> ... the exception which is detected by a program may not express the true nature of the cause of the error condition.

In this area, as well, our experiences matched. I found that exceptions were very useful, but care had to be taken that a handler be set up for each exception so that the user would know from where an exception originated.

However, she did find, as I did, that Ada's strong typing did provide an advantage.

> It is possible to define the GKS function parameters in such a way that a maximum amount of checking be performed on parameters and other data objects. Emphasis is intended on detecting logic errors at compile time, but run time checking would also be performed. The strong data typing allowed us to off-load checking for many of the GKS errors to the compiler.

On the other hand, strong typing presented her with a problem:

> This seems to be a good thing, but in order to implement it to the maximum extent, the binding would be lost in a sea of data type declarations which would be confusing at best.

In the first respect, our experiences were the same. I found that the strong typing and the run-time checking eliminated much manual error checking. However, in the second case, I did not find that strong type-checking was necessarily followed by the need for a large number of types. In fact, I thought that having specific types led to more understandable code.

The lack of completeness in the compiler posed a problem for her binding. Should a "least common denominator" approach be used? That is, should the binding be defined in terms of the data types that every compiler could support. As she put it, doing this would hinder the binding in that

> The use of a compiler which does not provide the full capabilities of the language would result in the inability to implement the system as intended, or to exploit the power of the language as it was designed to gain the benefits of reliability, extensibility, etc.

A conclusion that she reached is definitely validated by my experience.

> The programming support environment is very important. A non-validated compiler can be a lot of trouble if features likely to be needed are not implemented. The same goes for support packages. Validation also does not guarantee that the run-time system provides the necessary support for a GKS implementation.

2.8. Conclusion

Because Ada literature covers such a wide span, it was impossible to cover the entire range. I think that a thorough analysis would be interesting, because few programming languages have been the subject of so much discussion.

3. Implementing GKS in Ada

3.1. Configuration

Initially, in order to implement GKS in Ada, I decided to use UNIX[3] as the operating system, Telesoft as the compiler, the Digital Equipment Corporation (DEC) VAX[4] 11/780 as the target computer, and the DEC GIGI[5] graphics terminal as the target terminal. However, events conspired to change the plan.

Because of the limited availability and the heavy load on the VAX at RIT that was running Ada, I decided to switch to a Computer Consoles Power 6/32[6] superminicomputer. Since the Telesoft compiler was not available on the 6/32, I had to switch to the Verdix 5.2 compiler. This switch turned out to be advantageous because of the lighter load on the greater speed of the 6/32 (8 times as fast as a VAX) and the constant availability of the machine (I work on one).

As a result of the unavailability of a GIGI (they were being used in the undergraduate classes), I was forced to switch to using a DEC VT-240[7] terminal. It runs the ReGis[8] graphics language, as does the GIGI, but without color. I did not judge this to be a serious problem because the object of the thesis was to be a language investigation, rather than a test of the graphics package.

UNIX was kept as the operating system, because the 6/32 runs UNIX, as does the VAX. I used it because of my familiarity with the operating system. Also, I judged that the type of operating system was not critical to the project.

---

[3] UNIX is a registered trademark of American Telephone and Telegraph.

[4] VAX is a trademark of Digital Equipment Corporation.

[5] GIGI is a trademark of Digital Equipment Corporation.

[6] Power 6/32 is a trademark of Computer Consoles, Incorporated.

[7] VT-240 is a trademark of Digital Equipment Corporation.

[8] ReGis is a trademark of Digital Equipment Corporation.

3.2. Implementation Structure

This implementation is composed of three major parts: the main interface to the application, the output portion, and the input portion. Since input and output work differently, especially with regards to segmentation, the decision was taken to separate them. See Figure 1 for a pictorial description of the structure of this implementation.

The application interface, which is one package, contains all of the types, functions, and procedures that the application needs in order to use GKS.

The virtual output driver package contains routines that compose commands for the segment manipulation routines and/or the workstation output driver packages. It then sends the command to each active workstation.

There exists a output device driver for each type of workstation supported by the implementation. The device driver decodes the command sent to it by the virtual output driver and does the actual work.

For input, the situation is somewhat different because input commands are not stored in segments. In this case, the application calls the virtual input driver package. The virtual input driver package then passes, for each active workstation which is capable of input, the command on to the input driver package appropriate to the type of the workstation. The workstation input driver package then does the actual work.

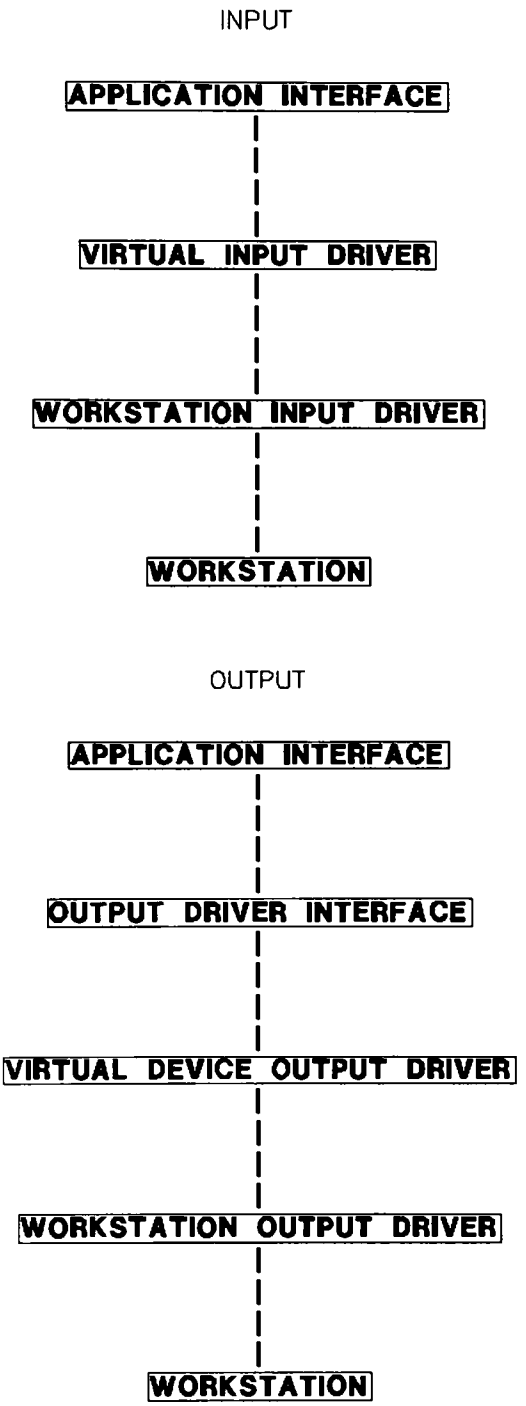
INPUT
APPLICATION INTERFACE
VIRTUAL INPUT DRIVER
WORKSTATION INPUT DRIVER
WORKSTATION
OUTPUT
APPLICATION INTERFACE
OUTPUT DRIVER INTERFACE
VIRTUAL DEVICE OUTPUT DRIVER
WORKSTATION OUTPUT DRIVER
WORKSTATION

Figure 1.

3.3. Translating FORTRAN to Ada

Because the graphics world is essentially numerically-oriented, FORTRAN is the language of choice. It is probably the best at "number crunching." For this reason, the GKS standard was first written in FORTRAN. Since then, it has been adapted to other languages, including Ada. The binding in this project is my own, however. I have not even seen the proposed binding.

As is the case with translating human languages, the translation of the FORTRAN binding was not done directly. Rather, I examined the function of the procedure or the variable in question, and then determined the binding to use.

Since Ada supports call-by-value parameter passing, as well as call-by-reference, while FORTRAN only supports call-by-reference, flags were able to be passed in directly to procedures without first assigning them to variables.

FORTRAN only supports 3 data types: integer, floating point, and character. It also supports one-dimensional arrays of each type. Ada, on the other hand, supports many more, including enumerated types, records, and sub ranges of existing types.

The flexibility of Ada allowed the creation of procedures which were passed types which were specifically designed for the application, instead of having to fit the available types to the application. Being able to have specific types meant that the functions of procedures and other identifiers were made easier to understand. Since Ada also supports multidimensional arrays of any type, arrays of coordinate pairs did not have to be separated into arrays of x- and y- coordinates, as they had to be in FORTRAN.

In addition, FORTRAN only allows identifiers with names that are eight characters or less. In addition, variable names starting with I through N are considered by default to be integers, unless specifically stated otherwise. Another limitation is the FORTRAN ignores the case of identifier names, e.g. LIMIT and limit are considered to denote the same identifier.

Except for case insensitivity, Ada has none of these limitations. This permitted the creation of identifiers whose functions were clearly identifiable by their names, thus making maintenance easier. Ease of maintenance is one of the claims that has been made for Ada.

In addition, Ada supports private and limited private types, the concept of which is not even available in FORTRAN. Through their use, types were able to be created of whose structures the user had no knowledge. Also, the values of variables whose types are limited private may not be changed without the permission of the package. In essence, the application was made more secure using limited types.

Example

A good example of the differences between Ada and FORTRAN can be found in the routine to initialize a stroke device. In FORTRAN, this routine is called GINSK, and is called in the following manner:

```
GINSK (WKID, SKDNR, TNR, N, IPX, IPY, PET, XMIN, XMAX, YMIN, YMAX,
       IL, IA).
```

The parameters have the following declarations:

```
INTEGER   WKID      Workstation Identifier

INTEGER   SKDNR     Stroke Device Number

INTEGER   TNR       Initial normalization transformation number

INTEGER   N         Number of points in the initial stroke

REAL      IPX(N)    X coordinates of points in the initial stroke

REAL      IPY(N)    Y coordinates of points in the initial stroke

INTEGER   PET       Prompt and echo type

REAL      XMIN      Echo Area limits

REAL      XMAX

REAL      YMIN

REAL      YMAX

INTEGER   IL        length of stroke data record
```

As can be seen, the parameters the FORTRAN subroutine declaration may only be integers, floating point number, and arrays of one dimension. Since FORTRAN arrays can have only one dimension, the x and y coordinates must be passed in separately.

The flexibility of Ada permitted a equivalent routine to be created that was much easier to understand. The equivalent routine in the binding that I created is called "INITIALIZE_STROKE." It is called in the following way:

```
INITIALIZE_STROKE(workstation, device, initial_normalization,
        initial_number_of_points, initial_stroke_points,
        prompt_and_echo_type, buffer_length, prompt_area,
        data_record);
```

The parameters and their descriptions follow:

Parameter	Type	Description
workstation	STRING	work-station identifier
device	STRING	device identifier
initial_normalization	NORMALIZATION_RANGE	initial normalization number
initial_number_of_points	POSITIVE	number of points in the initial strokes
initial_stroke_points	WORLD_COORDINATE_ARRAY	array of the points in the initial strokes
prompt_and_echo_type	STROKE_PROMPT_AND_ECHO_TYPE	prompt and echo type to use
buffer_length	POSITIVE	length of the buffer
prompt_area	PROMPT_AREA_LIMIT_TYPE	record that describes the area in which the prompt will be displayed
data_record	STROKE_DEVICE_DATA_RECORD_TYPE	contains miscellaneous data

Because many of the types used are not standard Ada types, the following is a description of the types used.

```
type NORMALIZATION_RANGE is NATURAL;

type WORLD_COORDINATE_ARRAY is array(INTEGER RANGE <>) of
       WORLD_COORDINATES;

type WORLD_COORDINATES is record
                            x: FLOAT;
                            y: FLOAT;
                            end record;

type STROKE_PROMPT_AND_ECHO_TYPE is (IMPLEMENTATION_STANDARD,
                                     DEVICE_DEPENDENT,
                                     DIGITAL_REPRESENTATION,
                                     ECHO_USING_MARKERS,
                                     JOIN_STROKE_POINTS_WITH_LINES
                                     );

type PROMPT_AREA_LIMIT_TYPE is record
                                 minimum: DEVICE_COORDINATES;
                                 maximum: DEVICE_COORDINATES;
                                 end record;

type DEVICE_COORDINATES is record
                             x: FLOAT;
                             y: FLOAT;
                             end record;

type STROKE_DEVICE_RECORD_TYPE is limited private;
```

The type NATURAL is a predefined type in Ada that includes all integers from 0 and infinity. The type POSITIVE is also predefined by Ada. It refers to all integers from 1 and infinity. The type STROKE_DEVICE_RECORD_TYPE is defined as limited private in order to prevent the user from changing the values in the record without the package knowing about it.

It should be obvious that despite the verbosity of Ada, the description of the Ada call is much easier to understand.

In use, the actual call is also easier to comprehend when it is written in Ada. In FORTRAN, the piece of code needed to initialize a stroke device could look like this:

```
  DATA X /1.0, 2.0, 3.0, 4.0, 5.0/
  DATA Y /6.0, 9.0, 7.5, 4.5, -1.0/

  WKID = 2
  SKDNR = 3
  TNR = 0
  N = 5
  PET = 2
  XMIN = 0.0
  YMIN = -3.0
  XMAX = 6.0
  YMAX = 7.0
  IL = 3
C IT IS ASSUMED THAT THE ROUTINE TO PACK THE DATA RECORD WAS
C CALLED TO CREATE THE DATA RECORD, IA.
  GINSK (WKID, SKDNR, TNR, N, IPX, IPY, PET, XMIN, XMAX, YMIN,
        YMAX, IL, IA)
```

It would take a while for someone to understand this call. The situation is much different in Ada. The equivalent call would be:

```
INITIALIZE_STROKE    (
                     workstation => "/dev/tty01",
                     device => "light pen",
                     initial_normalization => 0,
                     initial_number_of_points => 5,
                     initial_stroke_points => (
                          ( 1.0, 6.0 ),
                          ( 2.0, 9.0 ),
                          ( 3.0, 7.5 ),
                          ( 4.0, 4.5 ),
                          ( 5.0, -1.0 )
                          ),

                     prompt_and_echo_type =>  DEVICE_DEPENDENT,
                     prompt_area => (
                          minimum => ( 0.0, -3.0 ),
                          maximum => ( 6.0, 7.0 )
                          ),
                     data_record => data_record
                     -- It is assumed that the record has already
                     -- been created.
                     );
```

The above comparison shows how much more clear the Ada call is, despite the verbosity. It shows especially in the case of the prompt and echo type. Instead of a number, which would have to be looked up by the user and which could be mis-typed, an enumerated type is used. This argument can be checked in order to determine if it is in the proper range at the time of compilation by the compiler instead of at time of execution by the procedure called. Thus, the cause of a potential error is eliminated.

4. Evaluation of Ada

4.1. Introduction

In the following subsections, I will give my evaluation of Ada as it pertained to my experiences using it to implement the GKS graphics standard. Before I begin my evaluation, I would like to point out that my comments will at times pertain to only the Verdix 5.2 compiler with which I worked. However, I feel that the comments, as specific as they may be, can still contribute legitimately to the judgment of Ada as a whole.

The prime reason why I think that my specific comments about the Verdix compiler can be extrapolated into a general evaluation of Ada is that the compiler is validated. I assume that other validated compilers have similar characteristics. Indeed, from the postings that I have seen on USENET in the comp.lang.ada newsgroup, this assumption seems to be confirmed.

4.2. Strong points

Strong Typing

The strong typing of Ada facilitates the prevention of errors at the time of compilation rather than having to debug them at the time of execution. It has been my experience in operating system programming that I have had to spend much time tracking down bugs that were caused by passing the wrong type of parameters, incorrect number of parameters, or assigning the value of a variable of one type to a variable of another type. The probability that the program will run the first time is increased by using Ada. I think that strong typing is the most attractive attribute of Ada.

Packaging

The packaging ability of Ada made the development of the graphics package easier. For example, it became necessary to change the workstation list package. All that had to be done was to change the body of the package. It also made it easier to identify to which module a procedure, function, or type belonged. Instead of the call

```
PUT(variable);
```

which tells the programmer nothing about where the procedure is declared, one may state

```
INTEGER_IO.PUT(variable);
```

which is much more clear.

Another positive aspect about the packaging of Ada is that the routines the user was to use could be declared once and then left alone. Past a certain point in the design process, the only time that the main package had to be changed was to add a new error code.

Generics

This implementation of GKS makes heavy use of generic packages in the graphics package. Many data structures, including the workstation descriptors and segment descriptors, were kept in lists. Ada permitted the creation of only one list package that was able to be used for all of the lists in the package. Any bugs found were able to be fixed in just one place. Generic packages saved much time, because duplicate fixes did not have to be made.

Exceptions

The ability to raise exceptions rather than having to return error codes proved to make for much cleaner code than the typical testing of return codes. Exceptions allowed the collection of all error processing at one point. This method is much more understandable than the normal method. The following example will illustrate this point.

In C, a typical piece of code might look like this:

```
if (open(workstation, O_RDONLY) < 0)
     {
     error_handler("open_workstation", CANNOT_OPEN_WORKSTATION);
     }

if((workstation_descriptor = find_workstation(workstation)) ==
      (struct workstation *) 0)
     {
     error_handler("open_workstation",
          WORKSTATION_DOES_NOT_EXIST);
     }
```

The same code in Ada might look like this:

```
TEXT_IO.OPEN(workstation, file_descriptor, INPUT);
FIND_WORKSTATION(workstation, workstation_descriptor);

exception

when STATUS_ERROR =>
     ERROR_HANDLER("OPEN_WORKSTATION", CANNOT_OPEN_WORKSTATION);

when WORKSTATION_NOT_IN_LIST =>
     ERROR_HANDLER("OPEN_WORKSTATION",WORKSTATION_DOES_NOT_EXIST);
```

This code is much easier to understand and to debug.

Information-hiding

The ability to hide information that was specific to the implementation facilitated the hiding of the details of the implementation from the user. It also prevented the user from accessing variables of which he or she did not need knowledge. For example, the details of the structure of the workstation descriptor were able to be hidden. Thus, if the package needed to be changed, which it did (many times) during development, the package interface itself was left undisturbed. It did not even have to be re-compiled. Also, the list routines were able to be hidden from the rest of the package body.

The ability to prevent users from changing the values of variables whose types were declared as limited private was very handy. For example, by declaring that the type CHOICE_DEVICE_DATA_RECORD_TYPE was limited private, an accurate list of choices was ensured for input to the package. The user did not have to be relied upon to maintain an accurate list of choices, because he or she could not change the choices once they were installed in the record. Thus, the problem of null pointers was eliminated. The user could not suddenly decide to change a value to null and an explicit check for null pointers did not have to be performed at the time the menu was displayed.

Ranges

The ability to establish subtypes of scalar types with specific ranges facilitated the changing of ranges and the understanding of code. To understand the value of ranges, another comparison of C and Ada should suffice.

```
#define MAXIMUM 100
#define MINIMUM -1

if ((value < MINIMUM) || (value > MAXIMUM))
     error_handler("create_segment", ILLEGAL_VALUE);
```

Contrast that with Ada:

```
subtype VALUE_TYPE is INTEGER range -1 .. 100;

if value not in VALUE_TYPE'RANGE then
     ERROR_HANDLER("CREATE_SEGMENT", ILLEGAL_VALUE);
end if;
```

In the C example, at least two identifiers would have to be changed. In Ada, only one would have to changed. Also, more importantly, checking is done in Ada to ensure by the that the minimum would indeed be less than the maximum. In C, no such checking is done.

Enumerated types

The ability to use an enumerated type for error code make understanding the error codes generated by procedures easier. Instead of, as in FORTRAN, returning a value of, say, 12, when FIND_WORKSTATION is called and the workstation does not exist, the value WORKSTATION_DOES_NOT_EXIST is returned. If the enumerated type is given a clear name, the need for a table of error codes, as the FORTRAN binding of GKS has, is eliminated.

Identifier names

The ability to create identifiers that are longer that 8 characters long is a definite asset, Instead of having to name a routine GINSK, thus requiring a table to explain the functionality of each routine, it was possible to name it INITIALIZE_STROKE, which should be self-explanatory.

4.3. Weak points

Complexity of Ada

The complexity of Ada made it a difficult language to learn. Because it is supposed to "do everything," its rules are, at times, daunting, at others, inconsistent.

The Ada Language Reference Manual (LRM), which is supposed to be the "bible" to which all other books refer, is extremely hard to read. The Verdix compiler refers to the LRM in its error messages. Unfortunately, I sometimes found myself unable to decipher about what the Manual was talking.

This complexity leads to many questions because there are many interpretations of what the Language Reference Manual says. It leads to different compilers doing different things for the same statement, and to controversy, as evidenced in the aforementioned newsgroup, comp.lang.ada.

Inconsistencies in Ada

The language itself is inconsistent, It is supposed to be strongly typed, yet it allows the following declarations.

```
type FRUITS is (APPLE, ORANGE, PEAR, GRAPE);
type FRUIT_ARRAY is array (FRUITS range GRAPE .. ORANGE);
```

This declaration is clearly impossible. Yet, the compiler does not catch it at compile time. The exception PROGRAM_ERROR is raised at run time. This does not help much, because no indication is given about where the error occurred.

Restrictiveness of Ada

Ada imposes some restrictions which, in my opinion, are too limiting. An example of which can be found in the declaration of records. If an array is to be used in a record, it must first be declared as a type. Only then, may the array be used in the record. This differs markedly from C, where arrays can be used in structures (the C equivalent of records).

For example, in C, the following is legal:

```
struct car
     {
     int condition;
     int year;
     };

struct car_lot
     {
     char names[20][30];
     struct car[30];
     };
```

In Ada, more levels of indirection are needed:

```
type CAR_TYPE is record
                    condition: INTEGER;
                    year: INTEGER;
                 end record;

type CAR_ARRAY is array(1 .. 30) of CAR_TYPE;

type CAR_NAME is STRING(1 .. 20);

type CAR_NAME_ARRAY is array(1 .. 30) of CAR_NAME;

type CAR_LOT is record
                    names: CAR_NAME_ARRAY;
                    car: CAR_ARRAY;
                end record;
```

In my opinion, all of this typing is needless, because, in all likelihood, these types will never be used again. It can be argued that while space may be being wasted because the symbol table has to be larger, most computers have enough memory to handle it. However, this may not always be the case, as in the case of a personal computer running Ada.

Also, Ada is supposed to be a very sophisticated language. If a compiler for a much simpler language like C can handle this kind of structure, surely a more sophisticated compiler could handle it, too.

Pointer manipulation is extremely clumsy, especially for a language which is supposed to be able to work on a hardware level. For reasons which are not clear, at least to me, one may not take the address of a variable and assign it to variable which is a pointer to that type.

In a list routine that was to be constructed for the package, the function was to return a pointer to the list element that it found (It was to be used later), the following was found to be illegal under Ada.

```
type LIST_ELEMENT;

type LIST_POINTER is access LIST_ELEMENT;

type LIST_ELEMENT is
     record
     previous: LIST_POINTER;
     item:     INTEGER;
     next:     LIST_POINTER;
     end record;
     .
-- Irrelevant statements omitted
     .
     .
pointer: LIST_POINTER;
element: LIST_ELEMENT;

pointer := element'ADDRESS; -- illegal
```

This restriction, to me, is needless. To get around it, an UNCHECKED_CONVERSION procedure had to be instantiated in order to do the assignment. All UNCHECKED_CONVERSION procedures do is basically fool the compiler into allowing the conversion, at the cost of some speed. Also, the behavior of UNCHECKED_CONVERSION procedures is defined by the implementation. Thus, these list routines may not be portable.

On the other hand, to do the same thing in C would have only required the following statements.

```
struct list_element
     {
     struct list_element *previous;
     int item;
     struct list_element *next;
     };

struct list_element *pointer;
struct list_element element;
     .
     .
/* Irrelevant statements omitted */
     .
     .
pointer = &element;
```

The preceding, in my opinion, is much easier to use.

Compiler problems

The compiler produced objects that were extremely large. For a fifty line program that used the package, the object was over 750000 bytes. The object was larger than the size of PERPOS[9], which is the multiprocessor operating system on which I work.

The start-up time for the object was noticeable, because of all of the run-time checking that had to be done. Ada is supposed to be able to run embedded system, which I would think would require fast start-up times. What happens in that case?

Debugging using the Verdix development environment was very difficult. The symbolic debugger supplied with the package would only dump core. The only debugging method remaining was to insert statements to print values, which was slow because re-compilation and reloading were necessary every time that a debugging statement had to be inserted.

Many re-compilations of a module caused the compiler to become confused. The compiler mixed up the old module with the new. The only way to ensure that this did not happen was to remove the object before re-compiling the source. Obviously this should not have been necessary.

Also, the loader would occasionally not find an object if many re-compilations had been done on another object. Re-compilation from scratch of the entire package was frequently the only solution.

The compiler was slow and put a high load on the system. The load factor on the lightly loaded Power 6/32, on which the compiler was running, was frequently doubled or tripled when the compiler was running.

Occasionally, the loader would not find a number of objects. Fortunately, the loader had a verbose option that indicated how the load was being done. It turned out that the loader was constructed such that beyond a certain number of objects, it was to build an intermediate object from some of the modules. Then, it was to load the rest of the objects and the intermediate object, using the C loader into the final object. The call to the C loader was failing because the command was built incorrectly. The bug was worked around by redirecting the output of the verbose option of compiler into a file. Then, the command to the loader was corrected and the command was executed manually. The fact that anything had to be done by hand is unacceptable.

---

[9] PERPOS is a registered trademark of Computer Consoles, Incorporated.

Generic Packages

Generic packages were difficult to understand. None of the text books consulted or the Language Reference Manual were clear about how generics were to be declared. The syntax is confusing. For example, in the declaration

```
generic
     type FIRST_TYPE is private;
     type SECOND_TYPE is (<>);
package GENERIC_PACKAGE is
          .
          .
various declarations
          .
          .
end GENERIC_PACKAGE;
```

the syntax is not clear about what type can be passed to what parameter. I learned (finally), but I had a hard time figuring out the meanings. This is unfortunate, because the ability to have generic packages and procedures is, for me, one of the most appealing abilities of Ada.

The compiler allowed the declaration of a generic type that was a variant record, but when an attempt was made to use it in order to create a generic workstation type, the compiler panicked and exited with an error. As a result of this problem, the idea of having a single generic workstation had to be abandoned. Instead, a main record type that pointed to a workstation-specific list of characteristics had to be created.

The compiler ran out of memory in the main package because too many generic packages were instantiated. This problem recurred frequently. Much time had to be spent restructuring the project to get around this problem. I contacted a representative from Verdix, who claimed that our system was set up incorrectly. This claim was disputed by the system administrator. The Verdix representative also claimed that the compiler did not use pools for memory. A disassembly of the compiler proved otherwise. This compiler must be extremely complicated. This package, though of good size, is not as big as some. What would happen on a project that is even bigger?

The rules determining the types that can be passed to a generic package are not clear. For example, if a generic package was declared in this way:

```
generic
     type GENERIC_TYPE is private;
package GENERIC_PACKAGE is
          .
          .
various declarations
          .
          .
end GENERIC_PACKAGE;
```

and the following record type was declared:

```
type FILE_DATA is
     record
     name:      NAME_STRING;
     descriptor: TEXT_IO.FILE_TYPE;
     end record;
```

Because the type **TEXT_IO.FILE_TYPE** is limited private, meaning that assignments cannot be made to a variable of that type, FILE_DATA cannot be used as a parameter to GENERIC_PACKAGE. The reason why is unclear. In my opinion, any kind of type, restricted or non-restricted should be able to be passed into a generic package. Also, the characteristics of each type parameter should be inherited by the instantiated package.

Also, even if GENERIC_TYPE were declared as a limited private type, any attempt to instantiate the package using FILE_DATA would fail as well.

Bit-wise operators

Bit-wise operators are **optional**. I consider this omission to be serious deficit in a language that is to be used in embedded systems. When it was discovered that a GIGI was going to be unavailable, a decision was initially made to use a Tektronix 4014. Commands for a 4014 are composed by using bit-wise operators. However, an attempt was made to use them as in the following example, the compiler responded with:

```
package BIT_WISE is

WORD : constant := 4;

type STATE     is (A, M, W, P);
type MODE is (FIX, DEC, EXP, SIGNIF);

type BYTE_MASK      is array (0 .. 7) of BOOLEAN;
type STATE_MASK     is array (STATE) of BOOLEAN;
type MODE_MASK      is array (MODE) of BOOLEAN;

type PROGRAM_STATUS_WORD is
     record
     system_mask:   BYTE_MASK;
     protection_key: INTEGER range 0 .. 3;
     machine_state: STATE_MASK;
     ilc:       INTEGER range 0 .. 3;
     cc:        INTEGER range 0 .. 3;
     program_mask:  MODE_MASK;
     end record;

for PROGRAM_STATUS_WORD use
     record at mod 8;
     system_mask    at   0*WORD range 0 .. 7;
----^A
###
--### A:error: LRM 13.4(7): not enough space allocated for field
     protection_key at   0*WORD range 10 .. 11;
----^A
###
--### A:error: LRM 13.4(7): not enough space allocated for field
     machine_state  at   0*WORD range 12 .. 15;
----^A
###
--### A:error: LRM 13.4(7): not enough space allocated for field
     ilc        at   0*WORD range 16 .. 31;
----^A
###
--### A:error: LRM 13.4(7): not enough space allocated for field
     cc         at   1*WORD range 0 .. 8;
----^A
###
--### A:error: LRM 13.4(7): not enough space allocated for field
     program_mask   at   1*WORD range 9 .. 15;
----^A
###
--### A:error: LRM 13.4(7): not enough space allocated for field
     end record;

end BIT_WISE;
```

The error message caused me much confusion until I contacted someone from Verdix, who explained that the compiler did not support bit-wise operators. Why is this feature optional? Even a relatively primitive language like C supports them.

A more general question can be asked: why are optional features allowed in the first place? The policy of the Department of Defense is that there shall be no subsets of Ada. Allowing optional features directly leads to subsets.

Overloading

The overloading of procedures and functions can be confusing to the user and the compiler. A variant of the procedure PUT in the package FLOAT_IO was once needed in order to convert a floating point number to a string, but the compiler could not find it. No matter what was tried, the compiler would not recognize that the procedure existed. This included explicitly associating the parameters with the arguments in the call to the procedure. As they appeared in the package declaration, they were as follows:

```
procedure put  (file:    in file_type;
                item:    in num;
                fore:    in field := default_fore;
                aft:     in field := default_aft;
                exp:     in field := default_exp);

procedure put  (item:    in num;
                fore:    in field := default_fore;
                aft:     in field := default_aft;
                exp:     in field := default_exp);

procedure put  (to: out string;
                item:    in num;
                aft:     in field := default_aft;
                exp:     in field := default_exp);
```

I wanted to use the third version, but to no avail.

Exceptions

If an exception occurs and is not caught, no indication is given as to the procedure or function in which it happened. This omission is annoying. I would think that the symbol table could be consulted for the location of the exception. Much debugging time would be saved.

Input and Output

Input and output of different types were done incompatibly. If the input of an enumerated type was done just before the input of a text string, the input of the string was skipped. I was finally able to input the enumerated type by reading it in as a string and using of PUT in the enumerated I/O package to convert it from a string. I consider this to be a serious problem because it negated a valuable ability to input enumerated types and because input processing was complicated unnecessarily.

Input and output file descriptors may be depleted unnecessarily rapidly because different packages must be used for different types. Extra file descriptors are needed because the same file must be opened multiple times to do input and output of different types. Either that or the file must be repeatedly opened, read to or written from, and then closed immediately. Depending on the operating system, there may be a large amount of overhead associated with the increased file system activity.

File descriptors that are capable of both input and output are not available for text. The mode of the descriptor must be switched before the other type of I/O can be used. If it can be, since this ability is not required of the package. It is left up to the implementation whether to implement this ability. Even if it is implemented, having to switch is inconvenient, and I think unnecessary. Input-output file descriptors are available for other types of input and output, such as direct and sequential. The lack of input/output file descriptors is all the more puzzling because text I/O is the most used. Again, input-output file descriptors are available in other, less sophisticated, languages such as C. I cannot figure out why they are not available in such an advanced language as Ada.

Operating System Interface

The fact the Ada does not have an interface to the operating system on which it is running caused much difficulty. The only way to access the operating system for special functions is to use another language that did have an interface. In case of this implementation, that was C.

The problem was encountered in trying to raw input and output. The ability to do raw input and output was required in order to input strings while allowing the editing of the string. Some means was needed to get and put characters to the workstation, without them being buffered. Unfortunately, without making use of characteristics specific to the compiler this would have been impossible. As it was, it was difficult.

In order to do raw input and output in UNIX, it is necessary to use the UNIX I/O control call, ioctl(2). Before that was possible, it was necessary to get the UNIX file descriptor from Ada. Unfortunately, the Ada type FILE_TYPE is not equivalent. In the Verdix implementation, it is a pointer to a record type called FILE_RECORD. This record, in turn, includes the UNIX file number. These data types have the following declarations.

```
type file_descriptor is new integer; -- this is the UNIX file
                                     -- number

type file_ptr is access file_record;

type file_record is
     record
          fd              : file_descriptor;
          name       : string_ptr;
          mode       : file_mode;
          openmode   : file_mode;
          resetable  : boolean;
          index           : natural;
          linelength      : natural;
          pagelength      : natural;
          line       : natural;
          page       : natural;
          pos             : file_pos;
          delete          : boolean;
          file_id         : file_id_ptr;
          eof_char   : character;
          test_eof   : boolean := false;

          -- for buffering in the file.
          buffer          : access_bytes;
          last       : ptr_as_int;
          last_lf         : ptr_as_int;
          in_ptr          : ptr_as_int;
          out_ptr         : ptr_as_int;
          always_flush: boolean;

     -- for linked list of file descriptors: all open files
          next       : file_ptr;
     end record;

type file_type is new file_support.file_ptr;
```

The problem was compounded because the descriptor could not be accessed directly because FILE_TYPE is a limited private type. If a pointer type is limited private, neither the address contained in a variable of that type or the contents of the address contained in the variable may be accessed. Neither could the FILE_TYPE variable be passed to a C routine because Ada prohibits the passing of limited private types to routines written in other languages.

As a result of FILE_TYPE being a limited type, it could not be used in the workstation descriptor record and be used in the generic list package. In order to get around this problem, a type had to be declared, FILE_POINTER, that was a pointer to FILE_TYPE. As a result of the memory limitations of the compiler, the direct I/O package could not be used (Its instantiation caused the compiler to run out of memory.). In order to get around this handicap, an array of file descriptor pointers had to be declared, one for input, the other for output. This array was made an element of the workstation descriptor record.

```
type FILE_POINTER is access FILE_TYPE;

type FILE_DESCRIPTOR_TYPES is (INPUT, OUTPUT);

type FILE_DESCRIPTOR_ARRAY_TYPE is array(FILE_DESCRIPTOR_TYPES) of
     FILE_POINTER;

type WORKSTATION_TYPE is
     record
     driver_type:          DRIVER_TYPES;
     workstation_name:     STRING_POINTER;
     file_descriptors:     FILE_DESCRIPTOR_ARRAY_TYPE;
     window:               WORKSTATION_WINDOW_TYPE;
     state:                WORKSTATION_STATE;
     deferral_state:            DEFERRAL_STATES;
     implicit_regeneration_mode: IMPLICIT_REGENERATION_MODES;
     capability:           WORKSTATION_CAPABILITY_TYPE;
     is_written:           BOOLEAN;
     segment_list:  WORKSTATION_SEGMENT_HEAD_ELEMENT;
     deferred_instruction_list:    DEFERRED_INSTRUCTION_ARRAY;
     driver_table:         DRIVER_TABLE_POINTER;
     end record;
```

In order to do the I/O control call, a C module was used that did the following:

```
/*
     This structure is compatible with Verdix' FILE_RECORD type.
     The rest of the fields are omitted, because they are not
     relevant.
*/

struct file
     {
     int fd;
     };

typedef struct file *FILE_TYPE;
```

```
turn_on_raw_mode(file_descriptor)
FILE_TYPE *file_descriptor;
{
     register int fd;
     FILE *terminal_file;
     /*
      * Find the file pointer associated with the descriptor.
      */
     fd = (*file_descriptor)->fd;
     terminal_file = fdopen(fd, "r");

     /*
          Irrelevant statements omitted.
     */

}
```

Finally, to use the routine, the following was done:

```
pragma INTERFACE(C, TURN_ON_RAW_MODE);
TURN_ON_RAW_MODE(workstation.all.file_descriptors(INPUT));
```

As can be seen, the internals of the implementation of the compiler had to be relied upon in order to do the I/O control call. Consequently, this application is now not automatically portable from one operating system to another.

Also, to be able to do this, the internal workings of the operating system had to be known. The fact that this knowledge had to be possessed violates at least the spirit of Ada, which holds that information should be hidden wherever possible.

In addition, the compatibility of the output of the Ada and the C compilers, with regards to structure offsets had to be relied upon. It is fortunate that the Ada and 4.2 BSD C compiler lay out their record structures in the same manner. Otherwise, more non-portable steps would have had to be taken.

Finally, coping with these restrictions added to the amount of work that had to be done. This seemingly runs counter to the philosophy of Ada, which is to reduce the amount of work needed to be done by the programmer.

Antiquated restrictions

The case-insensitivity of Ada with regards to identifiers is annoying. It makes it necessary to create new names, I do not understand why a modern, sophisticated language such as Ada holds onto this restriction. Even C does not have this restriction.

Comments are limited to only one line. This is another antiquated restriction which I do not understand. Again, C does not have this restriction. There is much controversy about whether comments should be one line or multiple lines. A sophisticated language such as Ada should support both.

Passing arrays

Passing arrays that were not strings was not handled correctly by the compiler. The compiler could not correctly determine the length of the array. That meant that the following call would not work:

```
POLYLINE(5, (1, 2, 3, 4, 5));
```

The package routine called will get an erroneous value when in checks the length of the array. It was worked around by first declaring the array and then passing it in to the procedure. A user, though, would not know about the problem, and would get erroneous results.

5. Extensions and Revisions

Extensions

If I had more time, I would implement the rest of the package, with special attention to the remaining input modes: locator, pick, stroke, and valuator. Possibly more information about Ada could be learned by implementing them.

For completeness, I would implement metafiles. Metafiles are files used by GKS to hold instructions between invocations of the package. One can "re-run" commands given in a previous session.

Revisions

I think that one change that I would make would be to split the main package into smaller pieces. In this version, I renamed all of the routines in the various packages so that user could use them directly. I think that it would have been more efficient to keep the packages separate, and have the user call them specifically. The one disadvantage of this approach would be that the user would have to keep track of which package held which routine. Now, the user does not have to know.

Another possible revision would be to split the main package body into smaller pieces. The compiler memory limitation could be circumvented if this were done. I have developed different schemes for doing this, but none that would satisfy the needs of various sub-packages. Maybe someone else could come up with something.

6. Summary

In summary, I found that the Ada has the following advantages.

1. The strong typing of Ada facilitates the prevention of errors at the time of compilation rather than having to debug them at the time of execution.

2. The packaging ability of Ada made the development of the graphics package easier. Another positive aspect about the packaging of Ada is that the routines the user was to use could be declared once and then left alone.

3. Ada permitted the creation of only one list package that was able to be used for all of the lists in the package. Generic packages saved much time, because the same fix did not have to be made in many places.

4. The ability to raise exceptions rather than having to return error codes proved to make for much cleaner code than the typical testing of return codes.

5. The ability to hide information that was specific to the implementation facilitated the hiding of the details of the implementation from the user.

6. The ability to establish subtypes of scalar types with specific ranges facilitated the changing of ranges and the understanding of code.

7. The ability to use an enumerated type for error code make understanding the error codes generated by procedures easier.

8. The ability to create identifiers that are longer that 8 characters long is a definite asset.

9. The capability of creating variables of whose structures the users had no knowledge and whose values the user could not affect without the permission of the permission allowed greater security to the graphics package.

On the other hand, I found that Ada has the following deficiencies. They can be broken into two categories: problems with Ada itself and difficulties encountered with the compiler. As I stated earlier, I feel that compiler problems should be included because the compiler is validated.

The following is that of the problems that I encountered with Ada itself.

1. The complexity of Ada made it a difficult language to learn.

2. Because it is supposed to "do everything," its rules are at times, daunting, at others, inconsistent.

3. Ada imposes some restrictions which, in my opinion, are too limiting.

4. Pointer manipulation is extremely clumsy, especially for a language which is supposed to be able to work on a hardware level.

5. One may not take the address of a variable and assign it to variable which is a pointer to that type.

6. Generic packages were difficult to understand. None of the text books consulted or the Language Reference Manual were clear about how generics were to be declared.

7. The rules determining the types that can be passed to a generic package are not clear.

8. Bit-wise operators are **optional**.

9. The overloading of procedures and functions can be confusing to the user and the compiler.

10. If an exception occurs and is not caught, no indication is given as to the procedure or function in which it happened.

11. File descriptors that are capable of both input and output are not available for text.

12. The fact the Ada does not have an interface to the operating system on which it is running caused trouble because the internal structure of the implementation of the compiler and the operating system had to be relied upon in order to do the input and output of which Ada was not capable.

13. The case-insensitivity of Ada with regards to identifiers is annoying.

14. Comments are limited to only one line.

The following is a list of problems that I found with the Verdix Ada compiler. In some cases, I could not determine if the operating system or the compiler was at fault. In those situations, I put the problems in the compiler list.

1. Input and output of different types were done incompatibly. I am not certain if this is a deficiency in the language or in the compiler.

2. Input and output file descriptors may be depleted unnecessarily rapidly because different file descriptors must be used for different types. I am not sure if the language requires that the file descriptors be separate or if the implementation requires it.

3. The compiler produced objects that were extremely large.

4. The start-up time for the object was noticeable, because of all of the run-time checking that had to be done.

5. Debugging using the Verdix development environment was very difficult. The symbolic debugger would do nothing except dump core.

6. Many re-compilations of a module caused the compiler to become confused.

7. The loader would occasionally not find an object if many re-compilations had been done on another object.

8. The compiler was slow and put a high load on the system.

9. The compiler allowed the declaration of a generic type that was a variant record, but when an attempt was made to use it in order to create a generic workstation type, the compiler panicked.

10. Generic packages use too much memory. The compiler ran out of memory in the main package because too many generic packages were instantiated.

11. Passing arrays that were not strings was not handled correctly by the compiler.

7. Conclusions

I found that Ada is a basically good concept, but with a bad implementation. The idea of a language that is strongly-typed, but which can be used directly with hardware is something that is clearly needed. The present languages, such as C, which have this capability are handicapped by the fact that they are too loosely-typed. They allow programmers to "get away with too much." What usually happens is that the programs are done quickly, but much time is spent debugging them. A language such as Ada would reduce the amount of time needed for debugging.

Before I started the project, I was a proponent of strongly-typed, modular languages. I still am, despite the problems that I encountered during my implementation of GKS in Ada. Unfortunately, I am not a proponent of Ada.

Using Ada has been a series of struggles. The compiler made life difficult by refusing to accept a large project with many instantiations because of a supposed lack of memory. The complexity of the language meant that I had to spend a lot of time trying to comprehend exactly what was or was not permitted. The fact that some things, like bit-wise operations, were optional made it necessary to spend time attempting to implementing them myself. The inconsistency of input and output forced me to use extra file descriptors when they really should not have been needed, in addition to spending time figuring out how to get around the problem. The lack of an interface to the operating system required me to learn the details of the implementation of the compiler, which is something that Ada is supposed to prevent. Restrictions that were antiquated, such as case insensitivity, required me to spend time creating variable names which really had to purpose other than to satisfy the compiler.

The more that I worked with Ada, I found that using C looked more attractive. It is a language that is small, easy to learn (at least in comparison to Ada), and reasonably consistent in terms of syntax. The main problem with C is that is too permissive in terms of typing.

That I feel this way is unfortunate, because Ada does have a lot going for it. Its strong typing, packages, use of exceptions for error handling, the ability to create generic packages, enumerated types, the ease of developing identifiers that were clear and understandable all should have combined to make a language that is superior to many others.

The difficulties presented by Ada may be traced to the fact it was designed to do too much. It was designed to be for embedded systems, but I have seen it used or suggested for everything from controlling jet fighters to replacing COBOL in business applications. It may be impossible to design a language to do all of these tasks.

Ada would be a good teaching language because of its strong typing and modularity. In this way, it is similar to Pascal. It also would be a good design language, because its structures allow very clear designs to be built. However, in terms of use outside of academic circles for a programming language, I find it lacking due to the aforementioned factors.

8. Bibliography

[AJPO] Ada Joint Program Office,
Military Standard - Ada Programming Language (ANSI/MIL-STD-1815A).
United States Department of Defense,
Washington D.C., January 22, 1983.

[BOOCH] Booch, Grady,
Software Engineering with Ada.
Benjamin/Cummings Publishing Company, Inc.,
Menlo Park, Ca., 1983.

[COHEN] Cohen, Norman, H.,
Four Uses for Derived Types, and a Complication.
IEEE Computer Society 1985 International Conference on Ada Applications and Environments,
St. Paul, Minnesota. October 15-18, 1984.

[ECKAR] Eckart, J. Dana and LeBlanc, Richard J.,
Overloading in the Ada Language: Is it too restrictive?
Computer Languages. Volume 12, Number 3/4, 1987.

[PFAFF] Enderle, Gunter, Kansi, Klaus, and Pfaff, Gunther,
Computer Graphics Programming,
Springer-Verlag, Berlin, 1984.

[FOLEY] Foley, James D.,
Fundamentals of Interactive Computer Graphics,
Addison-Wesley, Reading, Massachusetts, 1982.

[FRIGO] Frigo, G. Vittorio,
Evaluation of the VAX Ada Compiler and APSE by means of a real program.
Ada Letters, May/June, 1987. Volume 7, Number 3.

[GILRO] Gilroy, Kathleen, Experience with Ada for the Graphical Kernel System.
Ada Letters, September/October 1984. Volume 4, Number 2.

[GOEL] Goel, Amrit L., Cavano, Joseph, Farhat, F.Y., and Little, Tom,
An Empirical Study of Fortran and Ada Reliability.
Proceedings of COMPSAC '87, Eleventh Annual International Computer
Software and Applications Conference.
Tokyo, October 7-9, 1987.

[HOPGO] Hopgood, F.R.A.,
Introduction to the Graphical Kernel System (GKS),
Academic Press, London, 1983.

[LEDGA] Ledgard Henry F. and Singer Andrew,
Scaling Down Ada (Or Towards a Standard Ada).
Communications of the ACM, February, 1982. Volume 25, Number 2.

[LEONA] Leonard, Thomas M.,
Ada and the Graphical Kernel System.
Ada in Use, Proceedings of the Ada International Conference.
Paris, 14-16 May 1985.
Reprinted in Ada Letters, September/October 1985. Volume 5, Number 2.

[MARTI] Martin, Donald G.,
Non-Ada to Ada Conversion.
Ada Letters, January/February, 1986. Volume 6, Number 1.

[MARZJ] Marzjarani, Morteza,
A Comparison of the Computer Languages Pascal, C, Lisp, and Ada.
Journal of Pascal, Ada, and Modula-2. January/February 1988. Volume 7, Number 1.

[MILAN] Milanese, V.,
ADA and NIL: Two Concurrent Languages for GKS.
Computer Graphics Forum 6. 1987.

[MITCH] Mitchell, Charles, Z.,
Engineering VAX Ada for a Multi-Language Programming Environment.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium,
Palo Alto, California. December 9-11, 1986.
Reprinted in Sigplan Notices, January 1987. Volume 22, Number 1.

[NYBER] Nyberg, Karl A.,
Using Representative Clauses as an Operating System Interface,
Ada Letters, July/August 1987. Volume 7, Number 4.

[PIOTR] Piotrowski, Walter G.,
Ada Information Hiding - A Design Goal Missed?
Ada Letters, May/June 1986. Volume 6, Number 3.

[REHME] Rehmer, Karl,
Development and Implementation of the Magnavox Generic Ada Basic Mathematics Package.
Ada Letters, September/October 1987. Volume 7, Number 5.

[SAMME] Sammet, Jean E.,
Why Ada Is Not Just Another Programming Language.
Communications of the ACM, August 1986. Volume 29, Number 8.

[SARKA] Sarkar, J.P. and Wong, T.T.,
Impact of Ada Features on Real-Time Performances,
Proceedings of the International Workshop on Real-Time Ada Issues,
Moretonhampstead, Devon, U.K., 13-15 May 1987.
Reprinted in Ada Letters, Volume 7, Number 6.

[SHOCH] Shochat, David D.,
Design Experience with Ada Versus FORTRAN.
IEEE Computer Society Second International Conference on Ada
Applications and Environments, Miami Beach, Florida.
April 8-10, 1986.

[STANL] Stanley, James, Ada, a programmer's guide with microcomputer examples, Addison-Wesley, Reading, Massachusetts, 1985.

[TOUAT] Touati, Herve,
Is Ada an Object Oriented Programming Language?
Sigplan Notices, May 1987. Volume 22, Number 5.

[WALLI] Wallis, F. J. L.,
Automatic Language Conversion and its Place in the Transition to Ada.
Ada in Use, Proceedings of the Ada International Conference.
Paris, 14-16 May 1985.
Reprinted in Ada Letters, September/October 1985. Volume 5, Number 2.

[WICHM] Wichmann, Brian A.,
Is Ada Too Big? A Designer Answers His Critics,
Communications of the ACM, February, 1984. Volume 27, Number 2.