The Ada–AI interface

by JORGE L. DIAZ-HERRERA

George Mason University
Fairfax, Virginia

ABSTRACT

In this paper we investigate the interactions between artificial intelligence and Ada. The Ada language has been mandated for use in all U.S. Department of Defense mission critical embedded systems. Artificial intelligence has become an important ingredient in such systems. Currently, LISP is the language of choice among DoD AI implementors, and its continued use may retard the expected widespread use of Ada. However, many algorithms used in typical AI applications are procedural in nature, and thus are better suited to languages like Ada. Key pivotal questions addressed here are: What are the specific linguistic needs of AI applications software development? What has Ada to offer? Is there a missing link between AI and Ada? One main conclusion drawn is that Ada provides adequate support for the conventional techniques used in AI (which represent 75 percent to 80 percent of typical AI code); the other non-conventional techniques may not be directly supported by the language itself but through the programming environment (APSE), the program library, and the run-time system.

Ada® is a registered trademark of US DoD, AJPO
INTRODUCTION

Several factors motivate this discussion about artificial intelligence (AI) and the programming language Ada. The most important is the fact that Ada has become the standard computer programming language for the U.S. Department of Defense (DoD) and recently it has been mandated for use in all information systems, and in particular for mission critical embedded software. The language may eventually dominate the software world, since it has a high level of standardization and it is expected to have a wide dissemination.

The Ada language is intended to be used in a great variety of applications; however, there may be some application areas for which the language may not be suitable. One of these areas is AI. Although no standard language exists, almost all AI programming within DoD is done in LISP. If this trend continues, Ada's expected usage and acceptance may be hampered.

Another important aspect indicates that most AI techniques do not seem incompatible with Ada. It has been reported that only about 20 percent to 25 percent of the code written for a typical AI application is “pure” AI code; 

in other words, 75 percent to 80 percent of AI code is inherently procedural (i.e., not appropriate for LISP or PROLOG but ideally suited to a language like Ada).

In this paper we discuss some of these factors, paying particular attention to the needs of AI and the features offered by Ada in this area. We do not intend to compare programming languages or discuss the benefits and pitfalls of AI, the Ada language, or their methods for software development. Instead, we provide a positive view about their coexistence. First of all, we explore the interrelations between these two radically different approaches to problem solving. We analyze AI and Ada approaches to software development within the framework of modern software engineering and current system complexity problems and reliability needs. We then identify the general requirements of typical AI applications and present the relevant aspects concerning the use of Ada for AI applications.

SOFTWARE ENGINEERING, ADA AND AI

For several years there has been considerable general discontent with the process of designing and producing software and the quality of the software produced. Efforts are underway to find ways for greatly increasing programmer productivity and for enhancing the quality of the products.

Two main research directions have been proposed. On the one hand, a popular revolutionary (or transformational) approach strives to develop a refined “programming environment” that provides full automation of the software production process and which is centered around one standard language and a standard set of interfaces. On the other hand, a less widespread but more revolutionary (or breakthrough seeking) approach tries to devise a new programming paradigm by adopting “knowledge based” tools into the software production environment. Both approaches are centered around the idea of sophisticated software engineering environments (SEE).

Ada

The Ada language is the cornerstone of many efforts within the evolutionary approach. The language is the result of an international competition for a new standard higher-order language specially designed for programming large real-time embedded applications. The effort came as a response to the increasing cost of software mainly caused by the difficulties of software maintenance and the huge number of languages and dialects in use.

Ada is basically a block-structured language, with excellent information hiding capabilities and system-level structuring features. The language provides a unified set of concurrent programming constructs and a well-defined program library and configuration management system. Ada is a design and implementation language, supporting both bottom-up and top-down incremental programming in which programs are made up of one or more (typically many) separately compiled units.

The Ada language defines a standard multi-layered open-ended programming support environment (APSE) as an integral part of the solution. The environment includes all facilities and tools that a software designer requires throughout the software life cycle, including methodology-specific, language-specific, and applications-specific tools.

AI

The revolutionary approach to the software problem is based on AI research, seeking ways out of the “von Neumann bottleneck” through newer computational formalisms for the software process. However, AI researchers have rarely concerned themselves with software reliability and maintainability. Software engineering techniques must be used during development, although various AI techniques can add new power to existing development tools. In AI, open-ended “knowledge representation systems” are the paradigms for programming in the future.
AI has made progress towards the development of concepts, linguistic tools, and techniques for knowledge representation. Different computational paradigms have been put forward and some of the most widely known are: functional programming, logic or predicate programming, rule-based and knowledge based systems, object-oriented or message passing systems, networks, and frame-based and production systems. In fact, the sheer number of programming paradigms may prevent the development of a standard environment. Recent developments call for more consolidated environments combining features originally found only in individual paradigm environments.

It is clear that programming is a problem solving activity. Thus it is expected that future programming environments will include “intelligent” tools. From an AI point of view, programming should be made as easy as possible by shifting the burden from the programmer to the machine through the construction of programming environments.

### ADA SUPPORT FOR AI APPLICATIONS DEVELOPMENT

AI languages focus on symbol manipulation and list processing, supporting dynamically changing representations and flexible (non-procedural) control flow. The linguistic and computational needs of AI can be grouped into two broad categories; namely, traditional and non-traditional requirements.

#### Ada Support for Traditional AI Requirements

Traditional AI requirements include basic features such as dynamic data structures, recursion, symbol manipulation, pattern matching, data as objects, reusable functions, and relaxed typing. Ada certainly is capable of handling traditional AI techniques.

All these basic requirements are satisfied by Ada’s clear and up-to-date control and data structuring facilities, powerful data abstraction mechanisms, and comprehensive support for modularity.

#### Dynamic data structuring

Ada defines a rich basic set of data primitives as well as novel facilities for specifying programmer-defined new data types. Dynamic data structuring: that is, defining and constructing data structures at run-time, is done in Ada as it is done in LISP. Automatic storage management is not required in Ada, although the Language Reference Manual does not preclude garbage collection (an implementation-dependent feature).

#### Recursion

Ada provides good recursive programming facilities.

### Symbol processing and pattern matching

Ada library packages are the means for providing LISP-like list processing and pattern-directed computation on list structures.

#### Data as objects

Ada provides specific language mechanisms which unify the representation and operations of programmer defined data types. This is achieved by using private types in packages, one of the unique features of the Ada language. Although not a “complete” object-oriented language, Ada possesses many of the required features.

#### Relaxed type checking

Even though Ada is a strongly typed language, it allows for the creation of “type-less” programs by using generic program units.

#### Reusable functions

Reusability is one of Ada’s strong points. It also is achieved through library units—an intrinsic concept in Ada—in the form of self-contained (generic) packages.

### Ada Support for Non-traditional AI Requirements

Non-traditional requirements are AI techniques not usually found in procedural programming. These include functional programming and programs as data, logic and predicate programming, incremental (and interactive) compilation, and rapid prototyping. These requirements are not directly supported by Ada at the language level, but can conveniently be satisfied by the Ada environment, its library system, and the Ada run-time system.

Non-traditional AI requirements can be satisfied by the Ada programming system at the programming environment level and at the standard libraries level. The environment level incorporates special AI tools. The set of standard libraries provides interoperability among the tools for supporting specific AI applications.

#### Functional programming and programs as data

Functional programming refers to elementary forms of function definition (no side effects). The idea is to use factor-forming operations combining primitive functions into more complex functions, and so on. This approach also eliminates the need for variables and “procedural” descriptions. In general mappings define applicative operators in which a function takes another function as input.

Ada satisfies the requirements for functional programming by providing a rich set of objects and primitive functions, data abstraction facilities for defining new types of objects and new primitive operations, and a mechanism for defining function forming operations—generic functions—with other functions as parameters to derive still other functions.
functions is a topic closely related to self-generating code, a technique used in LISP and other interpreted languages in which a function or program segment is developed at runtime. Employing this technique, data structures can be constructed and directly executed.

Ada does not directly offer the capability of producing "self-generating" code. In fact, this is possible if and only if the language involved is the "native" language of the underlying computational system (either as a virtual or physical machine). Dynamically definable functions might be possible with an Ada machine that directly executes Ada code (either virtually or physically).

Logic programming

Logic programming requires at least an elementary form for defining facts and rules (declarative programs). The main program component is information about the application, not procedural instructions. Algorithms are not completely under the control of the programmer. Instead an underlying mechanism known as an inference engine controls the algorithms. Programmers must master this underlying process in order to specify a correct set of assertions. These control mechanisms are inherently procedural and can be written entirely in Ada, in which case traditional declarative programs (e.g., PROLOG Programs) may be considered as pure data.

Incremental and interactive compilation

A key tool in modern programming environments is an incremental compiler, which operates (usually interactively) as the source program is changed by recompiling only what is necessary. The simplest approach to incremental compilation is to determine the minimal separately-compilable unit. In System-oriented languages such as Ada which have comprehensive automatic configuration management facilities, a simple change can easily cause several compilation units to be compiled. Syntax-directed editors and the maintenance of an online intermediate program representation (e.g., DIANA for Ada) makes incremental compilation more feasible for complex languages like Ada.

Rapid prototyping

Rapid prototyping is a methodology that can be applied in any programming environment. For example, logic programming considers programs as executable statements of the requirements analysis. As such, logic programming can assist in the early stages of the software life cycle, unifying executable systems analysis with databases containing rules as well as explicitly stored data, and using the same formalism for both programs and specifications. The executable analysis becomes some sort of system prototype, which could be automatically converted into Ada programs. Furthermore, "interface programs" (written in Ada) can be used to handle necessary typing, subprogram calls, and error handling.

Can Ada Coexist with Other AI Languages?

The Ada language definition makes provision for the possibility of inserting "foreign" code in an Ada program in the form of a pragma. However, multi-lingual, multi-paradigm programming environments in which traditional software engineering languages like Ada can directly interact with non-traditional AI languages are difficult to define for several reasons.

AI languages are interpreted languages, thus there is no common (low-level) language which can be used by a linker to produce running programs made up of Ada and LISP/PROLOG code. Furthermore, a direct interface from Ada to other languages is difficult, because Ada's run-time kernel depends heavily on its data types and exception handling mechanism. Finally, pragmas are recommendations; it is up to the implementer whether to provide them.

The Ada programming environment facilitates coexistence with other AI languages. Tools can be provided for automatically converting sentences written in a "functional notation" to Ada (generic) instantiations. In such a case, we are not using a programming language; we are using a program generator: a computer-aided program generator from ("executable") specifications. Furthermore, the run-time system provides the virtual machine needed, since it can provide LISP/PROLOG-like interpretative capabilities.

AN ADDITIONAL ADA FEATURE: TASKING

Unrestricted self-modifying functions are mathematically un-decidable and therefore should be avoided. Ada provides a different view of programs as data, in the form of task objects, which opens up new possibilities for "controlled" generative programs. In Ada, a (potentially concurrent) process is realized as a (constant) task object which is a data object consisting of: (1) a particular sequence of statements, (2) local data, and (3) entries for interprocess communication. As are other data objects in Ada, a task object belongs to a type: a task type. This type is a limited private type which can be used anywhere a limited private type object can be used. For example, it can be used:

- (chiefly) as an actual generic parameter
- as a subprogram actual parameter
- as a private package

However, even though we know their structure, we cannot manipulate task types as literal values. This restriction prevents the dynamic generation of completely new tasks.

Task types can be used to define task templates which, when combined with access types, can be conveniently used for dynamically creating (activating) as many tasks as needed at run-time. This is a very powerful technique leading to convenient designs of inference schemas corresponding to PROLOG clauses. Ada tasks also provide an executable model of the system, something quite useful for simulation and rapid prototyping purposes.
CONCLUSIONS

The successful support that Ada gives to AI applications creates new software systems with a higher degree of portability and reliability, increasing the chances for creating reusable software and for alleviating maintenance problems.

Both traditional and non-traditional requirements of AI applications development can be satisfied by the Ada programming system at three levels: (1) the language level, (2) the programming environment level, and (3) the standard libraries level. The language support level provides basic linguistic features and programming building blocks. The environment level incorporates special AI tools. The set of standard libraries provides interoperability among the tools for supporting specific AI applications.

Large portions of AI code are procedural by nature. Other AI-intensive code can be tackled by using functional and declarative notations, automatic conversion tools, and Ada runtime kernel (RTK) support. The factors involved are Ada language features and methods, APSE tools, and RTK (CAIS dependent).

Many practical results have already been reported, some of which are:

- Several efforts have produced systems for generating Ada packages from natural language specifications as well as PROLOG prototypes.
- Several inference engines for expert systems have been implemented in Ada.
- LISP programs have been automatically rewritten in Ada. Further, it has been shown that run-time performance is much better in Ada than in current interpreted functional languages!
- Semantic networks have been implemented in Ada, producing more flexible and enhanced networks.

REFERENCES