An Evaluation of Ada* for AI Applications

David R. Wallace, Intermetrics, Inc.

1. Abstract

Expert system technology seems to be the most promising type of AI application for Ada. An expert system implemented with an expert system shell provides a highly structured approach that fits well with the structured approach found in Ada systems. The current commercial expert system shells use Lisp. In this highly structured situation a shell could be built that used Ada just as well.

On the other hand, if it is necessary to deal with some AI problems that are not suited to expert systems, the use of Ada becomes more problematical. Ada was not designed as an AI development language, and it is not suited to that. It is possible that an application developed in, say, Common Lisp could be translated to Ada for actual use in a particular application, but this could be difficult. Some standard Ada packages could be developed to make such a translation easier.

If the most general AI programs need to be dealt with, a Common Lisp system integrated with the Ada environment is probably necessary. Aside from problems with language features, Ada, by itself, is not well suited to the prototyping and incremental development that is well supported by Lisp.

2. Is Ada Suitable for AI Development?

In order to answer this question we must look at what is required for developing AI applications.

2.1 AI Development Requirements

Two key phrases that describe AI development are:

- rapid prototyping
- iterative feedback development

AI systems are generally developed incrementally, where at each stage the current behavior is observed in order to determine exactly what the next stage should be. This requires great flexibility and is best supported by a language that allows either incremental compilation or interpretation. More specifically, AI development often requires heuristic search techniques that must be developed

*Ada is a registered trademark of the U.S. Department of Defense (AJPO)

on-the-fly to match the particular problem at hand.

This type of development further requires flexible dynamically changing data structures without strong typing. Any use of data declarations must be either very limited or automated in some way. Otherwise, the overhead of constantly modifying these declarations becomes unacceptable and what is worse, error prone. This data problem has been dealt with very successfully in Prolog and Sail by the use of an associative data base. This allows data access without explicit knowledge of the surrounding structure.

In larger AI systems the concepts of data abstraction or object oriented programming are used extensively. See [7] for some background on object oriented programming in AI. The motivations for their use in AI is the same as that elsewhere: use higher level concepts and hide implementation details in order to make development, modification, and maintenance easier. Object oriented programs have algorithms and data that are very closely coupled. In AI applications this coupling can be very dynamic, having procedure values mixed in with the data the procedure is going to use. This makes it very easy to create very powerful parametrized transformations. How a large data structure is transformed can often be determined by values within itself. Much of the current uses of data abstractions in AI code written in Lisp is somewhat unstructured. This is partly since Lisp does not support data abstraction as a language feature. However, data abstraction use is becoming more formalized via the increased use of expert system shells (see below).

Most AI applications require some type of general value or attribute evaluation and propagation mechanism. A simple example of this is the parameters and variables in a Prolog program. The order in which these attributes are evaluated and propagated is dynamically determined; thus it is impossible to predict their storage requirements or lifetimes. This requires a very general storage management system with garbage collection. Data on a stack will in general have the wrong lifetime and data on a heap without garbage collection will overflow during most AI applications.

2.2 Ada Features Favorable to AI Development

Ada is a modern programming language providing clear and up-to-date control and data structuring facilities. Thus it should be very good at providing programming support for well understood and highly structured programming tasks.

Compared with other languages of its type Ada also provides a great deal of leverage in dealing with data abstraction and certain types of variability. The key features that support this are overloading and generics. Packages with overloading and generics provide a very powerful data abstraction mechanism. Such features allow what appears to be one procedure to deal with a number of different data types.

Another Ada strong point is its comprehensive support for modularity. The package concept is a very useful way of organizing a data abstraction. With the cross checking provided by the compiler it is very easy to divide a large task into modular pieces that can be developed independently and reliably.

Ada is highly suited to any task that is highly structured, has a relatively static behavior, and has a close correlation between control structure and storage lifetimes. There are probably some AI applications that fit these requirements.

The package concept allows the construction of what are the equivalent of Ada language extensions – in terms of data abstractions. This means that predefined library packages could be constructed to model the following:

- Lisp list-processing language features, see e.g. [3]
- associative database language features, see e.g. [4]

Such features would go a long way in allowing reasonable AI programming in Ada. However, there are potentially serious problems in implementing these packages appropriately (see below).

2.3 AI Problem Areas for Ada

Ada is unsuitable for dealing with the variety of problems and approaches arising in AI research applications.

2.3.1 Compilation

For the most part Ada requires compilation. For the purposes of AI development the lack of a reasonably fast interpreter or incremental compilation system is a very serious problem. Dynamic debugging in this environment is often used to determine the next stage of development. Without a fast interpreter it is very difficult to get an appropriately dynamic debugging system. In AI development, incomplete programs are often run with values supplied through the debugger when missing sections are reached.

The strong typing and the large declaration overhead add a very high cost to the iterative feedback loop used for AI development.

There are further problems caused by the use of a language that requires compilation when a large system is under development; that is *recompilation.* A small change in one part of a large system may (and often does) force recompilation and modification in most modules of the system. In a Lisp development environment the use of an interpreter eliminates the need for recompilation and the flexible and general data structures eliminate the need for rewriting data declarations.

To be fair it should be noted that Ada is much better than most other languages (like Pascal or C) in this area. Ada provides for modular consistency in a large system with both recompilation analysis and intermodule type checking. And further, Ada's support for data abstraction, even though somewhat static, allows for limiting the global effect of local changes.

It should be noted here that Ada systems that support incremental compilation are just starting to become available; see e.g. [2]. Such a system could go a long way toward alleviating these development problems.

2.3.2 Storage Management

As mentioned above under AI requirements, AI applications evaluate and propagate values or attributes in a very complex and often unpredictable manner. In any case, it is rare that the lifetime of these attributes follows the control structure of the program. This requires a system of managing memory independent of the stack mechanism. Direct user control of such a system (e.g. explicit FREE) is out of the question because of the certainty of error. In any real AI application it is also not practical to simply avoid deallocation; no matter how much memory is available it will be used up. This means there must be a sophisticated memory management system with garbage collection. This provides correct reclamation of storage when data lifetime is over. It is unlikely that Ada systems will provide such a feature because its high overhead conflicts with real time requirements. However, it should be pointed out that the Ada definition does not preclude garbage collection, see section 4.8 of the Ada reference Manual. This is a feature that could be associated with a pragma.

User defined garbage collection would require the creation of a storage exception that, when raised, would call a user subprogram to deal with it. This subprogram would need to use unsafe practices to do low-level heap manipulation and bookkeeping. Ada does not have the language features to allow higher-level control of storage for garbage collection. This is due to problems with its data abstraction capability which is discussed in the next section.

2.3.3 Existing AI Packages

One further problem with Ada, especially for near term use, is the lack of existing AI packages. There are, of course, many existing AI packages written in LISP.

2.4 Is Ada Suitable for AI Re-implementation?

If we assume some AI system has already been developed in an existing AI language, then we could consider translating it to Ada. This would avoid the problems mentioned above with the AI development cycle. Further, this approach has been used in a number of AI applications. There is a hazard here, however, since it may not be possible or practical to translate all AI systems to Ada. Translation problems can be mitigated by using AI-language coding standards to limit hard-to-translate features and usages. However, hard to translate features and usages are legitimate and necessary for some applications. Translation problems are likely to arise in two areas:

- data abstraction usage
- garbage collection

Garbage collection was discussed in the previous section. In general, a Lisp program using the full data lifetime capability will not be translatable to Ada.

Ada does not have a true data abstraction facility. Even though Lisp does not support data abstraction as a language feature per se, its flexibility allows the user to define and use powerful data abstractions. Ada supports encapsulated data types via the PACKAGE feature, but does not provide explicit abstract type construction features. This will create translation problems. Missing functions or features include:

- *updating structures within the package to reflect the instantiation of an object:*
  Ada does allow auxiliary structures within a package but there is no automatic way to coordinate it with object creation. Such use is necessary, for instance, to do storage allocation with garbage collection.
- *type instantiation parameters or run-time type attributes:*
  For instance, a user cannot create a string type with string-length as a type-attribute.
- *initialization and finalization of an object:*
  These are necessary when data types interact with their type context. For example, in the case of garbage collection, it is necessary to record information both when an object is allocated and when it is de-allocated. Ada only allows a limited form of initialization; i.e. when the data representation

is a record structure. However, there is no way to do finalization.

For more details on the abstract type problems of Ada see the SRI analysis of Ada for AI uses, [5].

2.5 Expert System Shells in Ada

Expert systems are best built using a *shell*, like ART (automated reasoning tool) or KEE (knowledge engineering environment). These shells have their own syntax and provide a disciplined and highly structured way of building expert systems. The shell provides not only the inference mechanism for the expert system but also the modular and hierarchical organization. This area provides the most promise for the use of Ada.

The shell structure can be used to limit the complexity of features used and their interaction. Further, the shell can generate a large number of type declaration or long select statements where this would be impossible by hand. This is often what is necessary to cope with strong typing.

In an expert system, general attribute propagation among rules requires garbage collection. However, the problems with the data abstractions in Ada can be dealt with if, for instance, explicit subprogram calls are inserted at key points in the Ada program to coordinate allocation and de-allocation. It is not feasible to have such calls inserted by a user, but they can be inserted reliably by the shell.

The modular and hierarchical aspects of shells are well supported by Ada. On a large system this will support team development well. However, as mentioned above it is necessary to have version control and recompilation analysis when using a compilable language. Languages such as Pascal or C would have very serious drawbacks in this environment. Fortunately, Ada is designed to support consistent separate compilation so it is very well suited to this task. However, during development the compilation costs could become very high.

3. Mixed Environments of Ada and AI Language

If Ada is only well suited to use with expert system shells, as described above, then other use of AI must use existing methods. Currently the accepted approach to dealing with the most general AI programs is the use of Common Lisp. Common Lisp is becoming the standard AI programming language in the U.S. Prolog is not yet a major force, although developments in this area should be watched, especially in light of Japanese efforts. The way to solve this dilemma is to integrate a Common Lisp system with the Ada environment. For proper integration such a Lisp system would need to be supplied by the same

vendor that supplied the Ada system. In this way Ada can be integrated with AI Language tools and support. They can use shared list-processing and database packages and have the ability to call each other.

3.1 Impact on Development Tools

As long as Lisp components are under the same configuration management system, there should be no real problems. A Lisp system may require some of its own special tools, but these should not interact with the other tools.

3.2 Interfaces and Characteristics

The interface between Lisp and Ada is potentially complex. This can be made simpler by sharing standard packages (see below). However, in this case the only good solution is to require appropriate integration.

3.3 Operational Concepts

The biggest problem area in a mixed system is probably garbage collection. As described above, hand generated Ada is not designed to deal with this well. The only safe solution to this is to limit the actual AI work to the Lisp components. One can restrict the Ada components from allocation and de-allocation, unless they are correctly generated by, for instance, an expert system shell.

The storage management problem could be much simpler if it were possible to build packages that could deal with their own storage management without extra user calls. There appear to be only two ways to do this:

- low-level unsafe programming practices within the package
- language extensions to extend the data abstraction capabilities of Ada

Neither is particularly desirable.

3.4 Standard Packages

Standard packages that would be desirable for AI applications in Ada include:

- List Processing Predefined Package Library
- Associative Database Predefined Package Library

4. Conclusion

Because of its design goals Ada has some limitations in comparison with very powerful AI languages like Common Lisp. Except in very special applications, translation from Lisp to Ada is not feasible. Further, the modes of AI development are poorly supported by the Ada system. Ada is not well suited to the prototyping and incremental development required for AI work. Real promise, however comes in the area of expert system shells. The shells can be used to generate consistent Ada code that could not be generated by hand and further can generate complex constructs to bypass language feature mismatch problems. This should not be too surprising since the shell can use compiler implementation techniques used in Lisp.

If serious AI application beyond expert systems is anticipated, a mixed environment would be necessary. A language like Lisp provides the right language features along with support for AI style development. The most reasonable choice would be integrating Common Lisp in the Ada environment. However, access to some Lisp features from Ada would need to be restricted to ensure system reliability.

5. Bibliography

[1] Appelbe, W. F., "Abstract data types in Ada," Journal of Pascal, Ada and Modula-2, 3(1), 1984, pp. 26-29, 36.

[2] Crowe, M., D. Machay, M. Hughes, C. Nicol, "An interactive Ada compiler," Ada UK News, 6(4), Oct. 1985, pp. 47-50.

[3] Olgivie, J., "Using variant records: some basic Lisp functions in Modula-2," Journal of Pascal, Ada and Modula-2, 4(2), 1985, pp. 15-20.

[4] Poutanen, O., K-M. Varanki, T. Valimaki, "Notes on building a relational database management system in Ada," Ada in Use, Conf. Proc., Paris, May 1985, pp. 14-24.

[5] Schwartz, R.L. and P.M. Melliar-Smith, "On the suitability of Ada for artificial intelligence applications," Project 1019, July 1980, SRI International.

[6] Schwartz, R.L. and P.M. Melliar-Smith, "The finalization operation for abstract data types," Proc. of the 5th Int. Conf. on Software Engineering, March 1981, pp. 273-282.

[7] Stefik, M. and D. Bobrow, "Object oriented programming," AI Magazine, VI(4), 1986, pp. 40-62.