Technical Report
CMU/SEI-94-TR-9
ESC-TR-94-009

Artificial Intelligence (AI) and Ada: Integrating AI with Mainstream Software Engineering

Jorge L. Díaz-Herrera

September 1994

Technical Report
CMU/SEI-94-TR-9
ESC-TR-94-009
September 1994

Artificial Intelligence (AI) and Ada: Integrating AI with Mainstream Software Engineering

Jorge L. Díaz-Herrera

Academic Education Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8222 or 1-800 225-3842.]

Table of Contents

List of Figures

Artificial Intelligence (AI) and Ada
Integrating AI with Mainstream Software Engineering

Executive Summary

Artificial Intelligence (AI) components will play a major role in the next generation of embedded applications "... AI enables new and improved DoD mission functions by insertion of automated cognitive capabilities, e.g., smart weapons, smart sensors, or automated planning functions."– DoD Software Technology Strategy, 1991.

Artificial Intelligence with Ada is a reality! Ada is an excellent technology for the needed software engineering approach to integrate AI applications with more conventional software systems. In this report we describe Ada support for AI techniques and present empirical evidence accumulated over the last six to seven years showing the successful implementation of large-scale AI systems in Ada. An assessment of these experiences shows that many of the problems and challenges facing AI software development shops are fundamentally the same as those faced by the traditional software engineer. An important conclusion is that scalability from the requirements analysis and the design prototypes to a full-scale system is an ill-defined area for AI applications.

The report discusses issues covering the complete life cycle including the challenges of engineering AI software and the difficulties with real-time AI technology. Many challenges facing the AI community are equally shared by the Ada community. These include domain-specific technical issues; liability issues associated with building trusted systems; the difficulties with requirements analysis and with design methods that adequately encompass rapid prototyping; as well as testing, validation and verification techniques, and maintainability of long-lived systems. Specific problems with AI technology to scale up for the engineering of high-quality production systems are presented within the framework of modern software engineering technology. We address the unique issues brought about by the integration of AI components into more conventional embedded systems.

We present an overview and a general assessment of AI applications that have been developed with modern software engineering languages, and in particular, we discuss experiences in AI development using Ada. The report also summarizes survey results from the Association for Computing Machinery (ACM) Special Interest Group for Ada (SIGAda) AI Working Group (AIWG), highlighting lessons learned and sample applications. We describe the use of Ada as a specification language to define the functional behavior of an executive to coordinate the activities of the AI components; we show how this specification can be used to define and validate the interfaces between independent AI components. With these techniques, each of the independent AI components of a system can evolve through a typical AI exploratory development while the integrity of the complete system is maintained. We present detailed descriptions in Ada of the basic AI programming techniques and approaches. These include dynamic

data structures, object-oriented and frame-based programming, and non-procedural approaches such as functional and logic programming.

An interesting observation is that a large percentage of AI code is procedural by nature and that better productivity rates are achieved by using Ada; also, efficiency is much better when compared to traditional AI languages. Although we show favorable results from these multi-year studies, a total integration of AI with mainstream software engineering remains difficult at least at the conceptual level. There are some impediments to a completely satisfactory solution, but only a few restrictions are more intrinsically related to the current Ada standard (Ada83); these, however, are being dealt with in the next language revision known as Ada9X.

It is generally agreed that embedded on-board software implies real-time operations. The definition of *real time*, however, can vary widely. The less procedural the processing path, the more difficult it becomes to predetermine the flow of control and guarantee a response time. We also illustrate specific aspects of real-time AI computing, showing what can be achieved today and what remains to be solved.

Artificial Intelligence (AI) and Ada
Integrating AI with Mainstream Software Engineering

Abstract: In this report we discuss in detail pragmatic problems posed by the integration of AI with conventional software engineering, and within the framework of current Ada technology. A major objective of this work has been to begin to bridge the gap between the Ada and AI software cultures. The report summarizes survey results from the Association for Computing Machinery (ACM) Special Interest Group for Ada (SIGAda) AI Working Group (AIWG), highlighting lessons learned and sample applications. An interesting observation is that a large percentage of AI code is procedural by nature and that better productivity rates are achieved by using Ada; also, efficiency is much better when compared to traditional AI languages. Although we show favorable results on the use of Ada technology for the implementation of AI software, a total integration remains difficult at the conceptual level. There are some impediments to a completely satisfactory solution, but only a few restrictions are more intrinsically related to the current Ada standard (Ada83); these, however, are being dealt with in the next revision known as Ada9X.

1 Introduction

Recently, the use of Artificial Intelligence (AI) techniques in real-time control applications has emerged. The successful use of isolated AI-based tools in recent conflicts, such as Desert Storm, has prompted the development of new “intelligent” functionality to support several of the primary software technology strategy themes of the DoD. AI technology is showing to be of value in decision support systems, situation assessment (for example, target classification and early warning systems), by allowing the manipulation of vast amounts of information generated by modern sensors and intelligence systems. Most of these AI-based applications are being implemented directly or indirectly in CommonLISP. The continued use of CommonLISP will retard Ada’s final acceptance in this “specialized” programming area.

The possibility of using Ada, and especially Ada9X, for AI applications provides a unique opportunity to support AI software development while addressing software engineering concerns to at least maintain consistent quality control and promote integration with large-scale projects. This is of paramount importance for large, expensive, and often life-critical embedded systems.

AI with Ada is a reality! Ada is an excellent technology for the needed software engineering approach to integrate AI applications with more conventional software systems. At the very least, Ada can be used once the “what” is well understood; this allows for software engineering techniques to be applied to AI software for the deployment of critical systems. Ada provides the impetus for a team-oriented development of the AI components of a system. The language also has features for “augmenting” the language with capabilities for symbolic manipulation. *We do not pretend that Ada replaces traditional AI languages such as LISP and PROLOG, but suggest that it can be used for a significant percentage of the applications deployed.*

Our interest on this work, and a major objective of this report, is to begin to bridge the gap between the Ada and AI software cultures, a need that has already been acknowledged. “If Ada and AI are to coexist, an important element will be the ability for software engineers to understand knowledge engineering problems (and embedded AI challenges), and for AI specialists to take into account the requirements imposed by large, long-lived projects [Collard 88].” There is a clear need for this type of discussion. As Bernaras put it, “It has become traditional for researchers in AI to believe that software engineering has little or nothing to offer them in their work [Bernaras 90].” We have taken the initiative to start to change this view.

Our work is motivated by a number of factors as follows. First and foremost, developing “intelligent” systems by using typical AI technology is becoming a critical bottleneck in applying knowledge-based techniques in embedded real-time computing. It is indeed envisioned that AI components will play a major role in complex, large, distributed military systems [DoD 92]. These “intelligent” systems have to deal with additional factors related to higher quality software for the insertion of knowledge-based components into embedded applications. *Can the AI technology and tools scale up? Are they up to the challenge of engineering issues such as integration, verification and validation, real-time performance, and life-cycle maintenance?*

Second, there is the need to use knowledge-based techniques for control systems that cannot be completely modeled mathematically. The increased complexity of control systems makes them very difficult, if not impossible, to be analyzed by traditional linear system theory and control methods. They are in fact very good candidates for knowledge-based qualitative control. In such systems, quantitative as well as qualitative information (for example, experience of process operators and control systems designers) is integrated with time-varying information such as that obtained directly from sensors. These reasoning systems use symbolic computation techniques to enable computers to effectively use explicit knowledge representations to discover, infer, or synthesize “facts” about the external world and respond accordingly. These knowledge-based problem-solving techniques have to deal with additional factors related to more stringent real-time constraints, such as non-static data and time-critical responses. The less procedural the processing path, the more difficult it becomes to predetermine the flow of control and guarantee a response time. *Can knowledge-based technology integrate with more traditional notions of deadlines and real-time performance?*

Third, although this next generation of embedded systems possesses some unique special-purpose requirements, these systems must be integrated with existing conventional software performing conventional real-time tasks, such as sensor reading and data fusion, control functions, actuator feedback, etc. Designers of these AI-embedded applications face the same problems and challenges as traditional real-time software engineers. Most importantly, such systems are best engineered using a language like Ada, since the language directly addresses specific aspects in the construction of large, embedded, real-time systems. Actually, it appears that newer dialects of AI languages, like LISP, continually incorporate software engineering features from basically procedural languages like Ada. An important issue is whether Ada technology is suitable for AI applications. *How does Ada match up with specific computational requirements for next-generation embedded AI systems? What has Ada to of-*

fer to the general field of artificial intelligence?

In answering these questions, we must look at the problem space and at the two solution spaces from the point of view of both integrating AI with mainstream software engineering and implementing AI applications using the Ada language.

In what follows we look at the integration of AI and software engineering with Ada. In Section 2, we investigate the relevance of AI approaches to modern automated systems, we lay out practical and conceptual integration issues, and we enumerate some remaining questions. In Section 3, we discuss the use of software engineering technology, especially Ada, to tackle basic AI programming requirements. In Section 4, we present a summary of a survey conducted by the SIGAda AIWG. Finally, in Section 4, we present an overview of perceived deficiencies and proposed approaches pointing the way for a more complete solution using Ada9X.

2 Implications of AI for Embedded Systems

It is evident that the successful use of isolated AI-based tools in recent conflicts such as Desert Storm has prompted the development of new “intelligent” functionality to support several of the primary software technology strategic themes of the DoD [DoD 92]. Indeed, the dynamic growth of the field of AI and the tremendous progress in hardware technology makes it inevitable that AI components will soon play a major role in real-time embedded applications if they are not doing so already.

Of particular importance is the study of intelligent agents (IA) [Krijgsman 90]. These are knowledge-based reasoning systems that interact with a dynamic environment. Their actions are performed within an appropriate temporal scheme, with a high degree of adaptability to unanticipated data and conditional requirements. That is, to satisfactorily select and schedule the set of operations appropriate for a given situation, these systems must be capable of introspection. This type of application contains the most AI-oriented requirements which, when compounded with stringent performance requirements, represent a large portion of the relevant aspects of the problem space. Hence, to obtain a list of requirements from the problem space, we analyze intelligent control applications in this report.

An example is the use of knowledge-based technology for control systems that cannot be completely modeled mathematically. The increased complexity of modern control systems makes them very difficult, if not impossible, to analyze by traditional linear control systems theory. Symbolic computing allows the use of knowledge representations to discover, infer, or synthesize information about the external environment. This information is used as the basis of computational introspection. In terms of embedded systems, this gives rise to the possibility of developing systems that can “learn” from their environment, and that can “change” their own control programs to adapt to new situations; for example, these features are required to operate autonomous devices.

2.1 Prototypical Applications

There are several kinds of systems that have one or more of these characteristics. Among them we can mention intelligent control systems, planning systems, and simulation systems. For each one of these categories, we present how blackboard architectures can be useful for its implementation and provide references of its application.

2.1.1 Intelligent Control Systems

At each point in a problem-solving process, an intelligent control system has to determine the next action to perform. Each action can generate or modify the solution elements that would again require a decision to steer or control the intelligent system toward its final goal.

A typical application of AI-based control is in command, control, communication, and intelligence (C^3I) systems. We can identify several C^3I subsystems. See Figure 2-1. One such sys-

tem is TSATT (Time Sensitive Attack of Terrestrial Targets) [Díaz-Herrera 92a]. The mission of the TSATT is to function as the principal agency of the Tactical Air Control System for the decentralized execution of attacks against selected time-sensitive ground targets. The TSATT is responsible for the management and control of assigned air assets engaged in the detection, classification, attack, and destruction of selected time-sensitive targets on a time-urgent basis. New sensors such as Joint Surveillance and Target Attack System (JSTARS) provide accurate real-time information on enemy targets. In the past the Tactical Air Control System has operated on a 24-hour cycle, planning and tasking attacks on enemy targets through the Air Tasking Order (ATO), which is generated daily. TSATT is designed to deal with a set of assets allocated through the ATO and assigned to time-sensitive targets. The TSATT uses real-time sensor data to allocate these assets to the highest priority targets.

C^3I systems in the not-too-distant future will require the ability to process messages quickly and efficiently from multiple sources, and fuse and correlate that data into a coherent picture of the battlefield [Miles 92].

Another C^3I subapplication is that of automated message handling [Díaz-Herrera 92b]. In an environment in which all textual information received must be reviewed in terms of its relevance and grouped according to a mission's tasks, it is imperative that such processing be done rapidly and accurately. An automated text (message) processing system (ATPS) is intended primarily to assist in the classification of textual information, and hence to disseminate it to the correct users of such information. It is important that the correct group get the right information at the same time that the correct information gets sent to the right groups; this processing is principally based on the establishment of predefined categories, or profiles, which characterize a domain. This can be implemented as a distributed system of high-powered workstations and servers, providing information-gathering, dissemination, processing, and presentation services to a number of users. This setup provides a "universal terminal" interface to several other systems, but more importantly, it represents a quick and accurate facility for processing textual messages uniformly.

This kind of application performs a major role in the outcome of important military decisions, especially in an environment in which multiple sources of tasking are compounded with the use of human-oriented media (such as telephone conversations and verbal orders). Of paramount importance is a rapid and precise response to incoming messages. The growing volume of textual data from incoming messages and from database queries is already becoming overwhelming. The use of human operators manually processing textual information—producing, summarizing and routing messages—is slow, and leads to inconsistencies such as different interpretations of similar messages at different times. Errors can have a catastrophic impact during a crisis, especially when information given to commanders is inaccurate or irrelevant. The availability of this information in machine-readable form makes it more amenable to automatic processing.

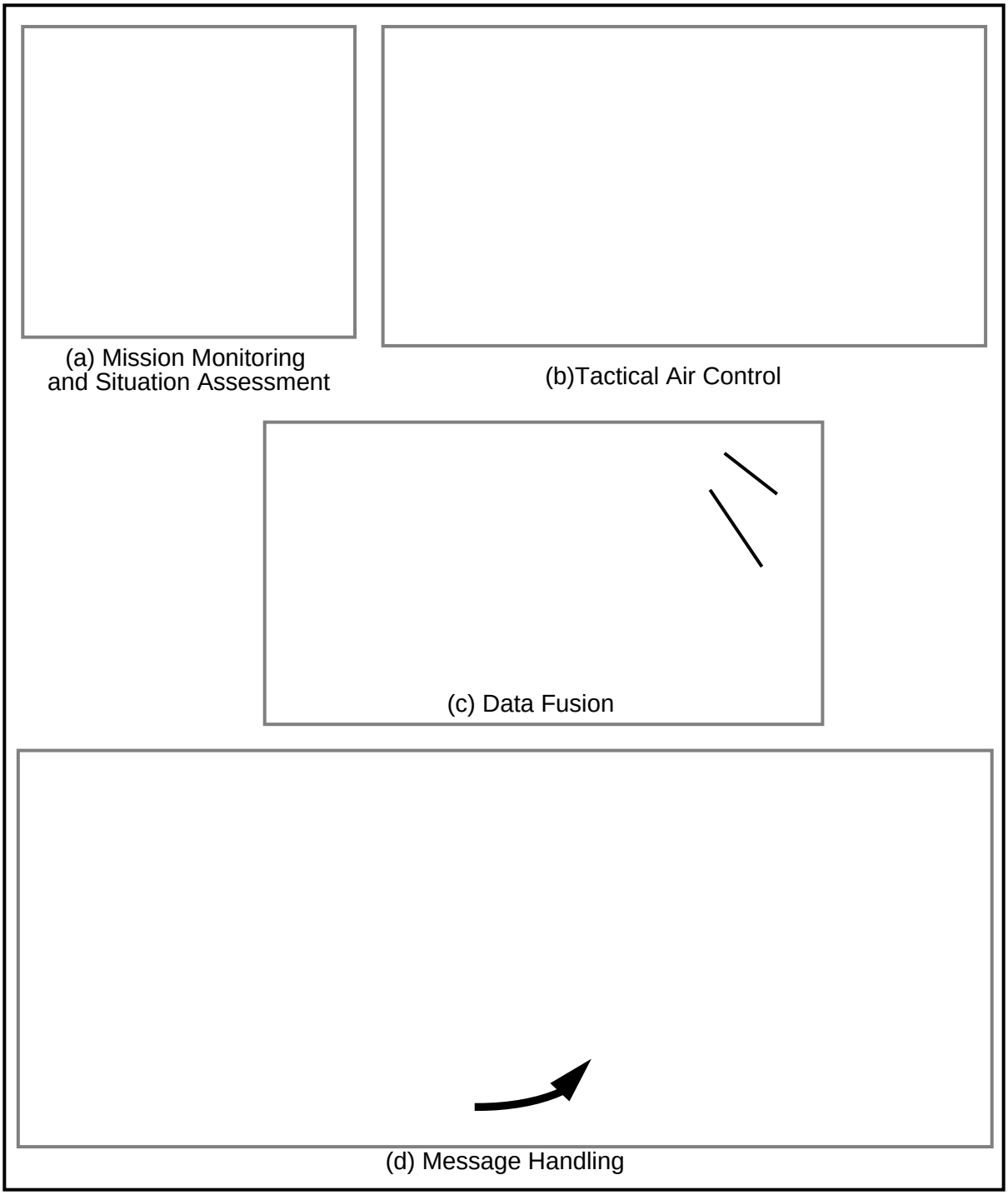

Figure 2-1 Intelligent Command and Control Applications

The thrust of AI-based approaches is to obtain an explicit representation of the meaning of the text being categorized. The latter, if successful, resolves ambiguities inherent in the use of words by contextual inferencing, and offers the possibility of obtaining very high accuracy. Natural language-oriented techniques discover knowledge (and thus categorization) from linguis-

tic knowledge and the use of dictionaries. More recently, practical natural language processing has resulted in systems with substantially higher accuracy, using more modest resources and within reasonable response times [Díaz-Herrera 92b].

As an example of intelligent control we looked at “BSEToll” [MacLean 88]. The development of an expert systems tool based on the blackboard architecture is described in this paper. The applicability of a blackboard shell to real-time control problems is demonstrated through the design and simulation of a problem of single-junction street traffic control.

The architecture of this system is a typical BB. The knowledge Sources (KSs) take data from one or more BB units and in the process of execution create or modify BB units. They may also read from an input file or write to an output file. The control unit chooses the invocable KS with the highest priority number for execution. This priority number is determined as a function of the user-defined static priority of the corresponding KS and the number or cycles elapsed since the KS was triggered.

The physical representation of the blackboard is a single PROLOG database. All the data structures are implemented in PROLOG as attribute-value pairs. The entries on the BB are called blackboard units. These units carry an identifier called level, which allows the user of the expert system shell to structure knowledge at different conceptual levels. Also, these units have an associated certainty factor that reflects the degree of belief that the fact they represent is true.

The proposed shell allows real-time solutions to control problems by admitting dynamic control in its control strategy and allowing a KS to be incorporated for keeping track of clock readings. The KSs may use different methods of reasoning to arrive at their respective information contributions. No particular inferencing technique is imposed on the KS. In the traffic application presented here, some KSs use a backward chaining inference engine while others use predicates in a simple forward chaining sequence. Each KS is a collection of PROLOG predicates, executed by a call to a main predicate. As we said previously, the BB stores the knowledge in different levels (seven in total). These levels also distinguish between data coming from an input source and data going to an output device. A backward chaining inference engine is included in the software to facilitate the writing of KSs. For KSs reading or writing out data, simple PROLOG rules are used. The data for the simulation of the system was generated by the GenSim program. GenSim contains PROLOG rules and uses a random number generator to convert user specifications into simulated data.

Pang [Pang 89] has also proposed a blackboard architecture as a framework for developing intelligent control systems. As an example of this architecture, an autonomous mobile robot is described. This architecture is adopted because it provides a good framework in which the different subsystems can be allocated to different KSs, providing good modularity. Each subsystem would be dealing with a different aspect of the problem and could be modified without interfering with the others. In the same way, new KSs could be added without major problems.

A BB is used to store the knowledge accessible to all the KSs. The objects in the BB may be

stored as a list of attribute values. Also, each KS has its own private data structure. It contains information necessary for a correct run of the KS. In the control blackboard (needed to achieve the behavioral goals of the system) there are control data objects, used by the control unit to select which of the potentially executable KSs to execute.

Since many different sources of knowledge may be needed, Pang suggests that each KS may have a different knowledge representation or different inferencing technique. Therefore, the algorithms would be determined by each one of the modules composing the system according to its characteristics and needs. Also, Pang claims that blackboard architectures are well suited for parallel or distributed processing because the assessment of the KSs for the selection and scheduling can be performed concurrently.

An experimental system has been implemented using Quintus PROLOG. For numeric computation, procedural routines can be called from Quintus PROLOG. The system, installed in an autonomous mobile robot of the Pattern Analysis and Machine Intelligence Laboratory of the University of Waterloo, runs over three 68020 processors on top of the Harmony operating system. Pang recognizes that, although the blackboard architecture provides generality, modularity, easy modification, and extensions, there is a high associated cost in computational resources and information storage.

Another example of a software architecture for intelligent monitors is that of Trellis [Factor 89]. Trellis has been applied to the development of an experimental intelligent cardiovascular monitor (ICM) that is intended for eventual use in a cardiac intensive care unit.

The proposed architecture is the process trellis. The process trellis is a hierarchical network of logically disjoint decision processes. Each process is responsible for a semantically meaningful piece of the solution; the division into processes should mirror the domain's logical structure. Processes run continuously, concurrently with all other processes. It is an essential attribute of the architecture that processes are logical black boxes (they can incorporate any kind of logic that seems appropriate).

Every process in the trellis hierarchy has a set of inferiors and a set of superiors. Control flows upward when a process' inferior produces a new state, and it flows downward when a process queries an inferior. Information flows upward when a process' inferior produces a new state, and it flows downward when a process' context produces a new state. The parallelism of the trellis process enables the application to attain improved performance when run on multiple CPUs.

A process trellis shell implements all interactions between processes. The shell maps the trellis processes to the CPUs of a parallel computer. In addition, the shell provides both a graphics and a menu interface to allow interactive invocation of probes and trellis program debugging. This graphical interface was implemented using X11R3. The process trellis shell runs with only trivial changes on the encore Multimax, the Sequent Symmetry, and the Intel iPSC2 parallel computers.

Trellis processes are written in procedural languages and can contain arbitrary code. The runtime portion of the trellis shell is implemented in the parallel language Linda, which extends C with a small number of primitives to create processes and to enable them to communicate through a logically shared associative memory. The trellis shell contains 11,500 lines of documented code, 7,500 of which are in the runtime kernel. The current version of the ICM system contains 64 processes comprising roughly 27,000 lines of code, excluding the trellis shell.

The process trellis is presented as a valid alternative to the blackboard architecture. While blackboard models are characterized here as inherently sequential, trellis processes are presented as very suitable to parallel processing. Because the trellis shell implements all the parallelism, an individual with no detailed knowledge of parallel programming can create a trellis program.

2.1.2 Planning Systems

A good planning system must be able to evaluate the relative importance of the information available, choose between conflicting goals, and reach conclusions in spite of incomplete information. Also, especially in real-time planning systems, it should be possible to determine the action to take based not only on the available information, but within the existing time limits. Depending on these limits, a more careful plan can be deployed if more time is available. Independent and concurrent processes can achieve the goal of making the best plan within the existing constraints in the system.

Multilevel planning systems have been applied to address some of the real-time aspects of planning for threat response [Petterson 88]. Planning maneuvers for combat aircraft can occur at different levels, where the choice of level is determined by the time available for planning. The Adaptive Planning for Threat Response [Hayslip 89] is an excellent example. The main task of the system is to plan maneuver decisions for air-to-air combat at low altitude over hilly terrain.

Real-time planning applications require a method for organizing and distributing knowledge that facilitates reasoning. The original Rapid Expert Assessment to Counter Threats (REACT) is another example of a DAI multilevel planning system. To address the problem of real-time planning in REACT, coupled systems, concurrency, multiple knowledge representations, and an architecture in which specific reasoning about time is localized to a specific module are used. The solution adopted to resolve this problem made use of *nested blackboards* to be able to have concurrently operating planners, each operating with a different level of knowledge detail and at different time scales. Ideally the processes will run concurrently, so if a higher level decision becomes available, the lower level behavior can be subsumed, and if higher level decisions are not ready in time, the lower level is instituted.

The REACT knowledge representation hierarchy consists of three different levels:

1. *Preprocessed maps*. This is a structure composed of reduced terrain patterns resembling checkerboards transformed into arrays of numbers. Associated

maneuver choices are prestored for various relative opponent placements over the map.

2. *Terrain descriptions in terms of tactical terrain primitives* providing a symbolic description of the layout of the land. At this level reasoning about the terrain can be accomplished by inference rules.

3. *Numerical models, Defense Mapping Agency elevation maps*. Unreduced world representation. This is the layer where the numerical models live.

A prototype system was developed on a Symbolics 3675 computer using the Georgia Tech Generic Expert System Tool (GEST) with specific modules written in LISP and numerical routines implemented in procedural languages on a Data General MV/1000. Because of its modular design, the REACT system allows modifications and/or extensions without too much effort. Also, because the problem of determining the time available for planning is relegated to a higher level expert, the competing planners do not have to reason explicitly about time. On the actual implementation, the concurrency of the planning modules is simulated.

2.1.3 Simulation Systems

Simulation systems are usually used to obtain predictive information that would be costly or impracticable to obtain with real devices. The information gained from simulation experiments contributes to decisions about the real system modeled by the simulation. Because the simulation model captures the change in the status of the system by focusing on the behavior of the individual, usually independent components, blackboard architectures are particularly adequate for implementing these kind of systems.

Parks and colleagues [Parks 90] describe a blackboard/knowledge-based system for manufacturing scheduling and control. The system uses a real-time global systems simulator (GSS) that provides real-time decision support, analysis, and implementation options, with a distributed environment to provide hierarchical decision-support facilities. The main task of this system is to schedule manufacturing operations based on real-time simulation data. The architecture of the system is the typical one from a blackboard system. This architecture was chosen because it provided the management with a tool to effectively monitor shop floor progress and analyze its impact on the master manufacturing plan while providing the shop floor personnel with support for short-term operation.

The representation methodology should be able to support general, procedural, and constraint knowledge. The representation must also permit interfaces with existing data models and application programs. Knowledge is represented as rules, frames, or logic assertions. To avoid a storage-intensive system, they used currently existing databases. Through prototyping, the architecture is refined one work center simulator at a time. The complete system will then be built in an IBM 3090-600 network to provide both remote and distributed access. The approach is generic and can be used in other factory environments as well as in other environments in which simulation is needed. The system is adaptable to modifications and expansion owing to its structuring.

2.2 Integrating AI and Software Engineering

The integration of matured AI methods and techniques with conventional software engineering remains difficult and poses both implementation problems and conceptual problems. In this report we are mainly concerned with implementation problems. These include, more specifically, two aspects. First, there is component-level interoperability; that is, the use of existing AI software and its knowledge bases with other conventional components. The second is referred to as *AI components reengineering*—the process of restructuring existing matured AI components using software engineering practices to enable effective enhancement, adaptation, and maintenance through their continued use. These issues represent software engineering challenges that span the complete software life cycle and software engineering languages such as Ada.

2.2.1 Conceptual Problems: Real-Time AI?

Researchers in this area do not consider real-time constraints in the same perspective or with the same focus as traditional software engineers. On the one hand, typical real-time approaches depend on a serialized algorithm based on predefined timing considerations, where control flow is synchronized with a real-time clock and is predictable because of the tightly coupled nature of the computations being performed. On the other hand, it has been noted that intelligent control systems are almost always in a state of computational overload, and thus cannot in general perform all potential operations in a timely fashion. This precludes the use of traditional static scheduling strategies [Hayes-Roth 90]. The objective of intelligent control systems is to seek satisfactory performance for a range of AI tasks, and real-time performance is only one of several objectives. In AI-based control mechanisms, deadlines are approached gradually; the need for a response progressively decreases with time, rather than at once, as is the case for hard deadlines. Achieving a correct real-time AI-based solution amounts to selecting the right set of actions, in response to stimuli, at the right time. Recent work on reflective systems is addressing these issues [Stankovic 93]. According to Stankovic, “Reflection is defined as the process of reasoning about and acting upon the system itself.”

In AI-based control systems, quantitative as well as qualitative information (for example, experience of process operators and control systems designers) is integrated with time-varying information obtained directly from sensors. Information can be tactical; managerial; related to scheduling, operations, or control; etc. Because of new data coming in from sensors or changes in the state of the system, data in these systems is not durable and decays in validity.

Although little work has been reported in the literature in this research area, preliminary investigations have identified the following related approaches:

- Analytic decision techniques [Horvitz 87]
- Progressive deepening [Winston 84]
- Progressive reasoning (subset of progressive deepening) [Wright 86]
- Time-constrained inference [Sorrells 85]

- Reasoning with incomplete information and resource constraints [Michalski 86]

There are two basic approaches. First, in a cognitive-servo approach [Park 93], control activities are performed directly by the problem-solving components of the application. Symbolic processing is part of the control loop, making it easier to maintain “timely” decisions at the expense of introducing cognitive latency into the control loop. Second, in the hierarchical-controller approach [Krijgsman 90], problem solving is taken out of the basic control loop by having it modify control parameters of a conventional lower level controller. Control latency is not affected by symbolic processing.

The blackboard paradigm is the architectural design predominant for DAI [Corkill 90]. The blackboard architecture is a collection of modules with special attention paid to information exchange. Conceptually, it is synonymous with global memory. The basic blackboard architecture consists of a shared data region called the *blackboard* (BB), a set of independent *knowledge sources* (KSs), and a control unit called the *scheduler*.

A blackboard architecture is particularly appropriate for applications requiring multilevel reasoning or flexible control of problem solving. This paradigm is a powerful tool for expert systems integration and a good model of real-time problem solving. It also provides an excellent integration framework for combining diverse problem-solving techniques. It can deal with large amounts of diverse, incomplete, and even erroneous knowledge to solve problems.

Because of their nature, blackboard architectures are specially useful for systems with the following characteristics:

- Several or many different sources from which to deduce a following state of the system from the current one. These sources can also be from different types.
- A single state space shared for the system components, to store not only the state of the system but also intermediate and partial results.
- Need for different behavior according to the state of the system or environment.
- Need for parallel processing (different things must be accomplished by different components of the system at the same time).
- Need for different levels of precision, both in data and in processing, generally based on time constraints.

2.2.2 Implementation Problems

Our investigation indicates that AI technology already exists and has been applied in the realm of expert systems for fault diagnosis, event tracking in a command and control environment, and text processing in general. Typically a developer builds a rule base specifying categories and concepts to be handled. Interfaces to generic input/output processing facilities can be readily provided and typically require traditional programming skill.

We have also noticed that although the technology does exist for building intelligent systems with an adequate level of accuracy, and several systems are commercially available, current implementations are not sufficiently flexible to rapidly adapt to different domains of discourse.

They depend on the availability of specialized domain knowledge, which makes these systems less adaptable and also more difficult to maintain.

Very few applications have progressed beyond the prototype stage for everyday use in a real-time embedded setup, and only a handful of existing AI software systems show any of the acceptable engineering attributes. These systems are not written according to modern software engineering practice and software development standards, and at the moment are unreliable and difficult to debug.

Software portability is another very important issue. In general, portability depends on the availability of standard languages and standard software practices. The use of common languages and standard software development practices is also of crucial importance in software engineering to maintain consistent quality control. Lack of standardization is detrimental to large-scale projects for which the significant use of AI techniques is expected. Commonality and portability are issues of minor significance in the AI community. In fact, the sheer number of programming paradigms may prevent the development of a standard.

Another problem is the capacity of such a technology to satisfy real-time performance requirements, in terms of both execution speed (response time) and coverage (size of knowledge base). AI techniques are generally very computationally expensive. The challenge for intelligent processing is to extract useful information with high degrees of accuracy as opposed to finding the most efficient and fastest algorithms. For example, it is not desirable to have the wrong messages delivered fast!

All in all, the state of the practice of AI is characterized by a variety of concepts, techniques, and linguistic tools independently developed in research laboratories, some of which are slowly finding their way into industrial and government shops. AI software technology is not in place for the development of well-engineered AI applications. Currently ad hoc techniques are used, and more often than not, it is the responsibility of the programmer (by patching) to make the system “work.” This approach is inadequate for a number of reasons:

1. Ad hoc methods cannot be exported to other projects.
2. Performance becomes brittle with changes in the specification or in the environment.
3. Hand-tuning is time-consuming.

Modern software engineering activities work together to achieve preset software engineering goals that, if attained, will lessen the effects of the so-called software crisis by providing high-quality software at greater levels of productivity. These goals include, among others, maintainability, correctness, reusability, testability, reliability, and portability of software systems. A number of software engineering principles can be identified that contribute to the achievement of these goals. These principles involve the use of modular decomposition, encapsulation, step-wise refinement, information hiding, abstractions, stylistic conventions, etc., during software development.

We refer the reader to the general software engineering literature for a complete discussion

of these goals and principles. However, we would like to highlight life-cycle requirements, portability and adaptability, maintainability, and performance as being of paramount importance. We discuss each of these in turn.

2.2.2.1 Life-Cycle Requirements

AI software technology is at odds with software engineering practice. A basic deficiency of current practice is the lack of discipline similar to that found in software engineering for large embedded systems. AI tools provide poor or no support for team-oriented programming-in-the-large software engineering. Many existing tools can be difficult to integrate with traditional software engineering environments, and their use may add unnecessary "software baggage" because of a lack of modular design.

What are the AI design milestones? How are knowledge engineering specifications written? How are knowledge engineering designs written? How are knowledge bases maintained?

The AI life cycle is characterized by an evolutionary development from prototypes. Ideally, prototype knowledge would be transferred to full-scale development. However, many AI applications do not progress beyond the prototype stage for everyday use. Only a handful of existing AI software systems shows any of the acceptable high-quality engineering attributes. Currently ad hoc techniques are used, and more often than not, it is the responsibility of the AI programmer (by patching) to make the system "work." This approach is of course inadequate because

- Ad hoc methods cannot be exported to other projects and will not scale up.
- Performance is brittle with changes in the specifications or the environment.
- Hand-tuning is time-consuming since the programmer is performing an unconstrained search through the space of possible modifications to identify a subset that meets all constraints.

Until one or more AI applications have been prototyped and developed, AI application requirements are difficult, if not impossible, to define. Since there are only a few well-defined requirements, and requirements are intrinsically vague, AI requirements analysis requires prototyping and/or rapid prototyping. These prototypes are used as specifications. The design of an AI application is also accomplished with iterative prototypes.

There is a mixture of procedural processing (80-85%) and non-procedural code, i.e., rules, in AI applications. Classic implementation characteristics of traditional AI programs include late binding; large address spaces; extensive use of shared memory for communication; extremely large programs; very dynamic problem spaces; and complex and non-homogeneous data structures.

Test, verification, and validation are difficult and ill-defined for AI applications.

System integration is also difficult. Embedded AI faces difficult integration problems. For example, there are requirements that integration must not destabilize existing software and must not exceed stringent requirements such as the limits for data bus loads, computational re-

sources, weight, power consumption, etc. A set of reusable components, readily available in conventional software engineering environments, does not exist.

2.2.2.2 Portability and Adaptability

Portability refers to the level of effort, and in fact to the possibility of transporting a system from one environment to another (with hardware/operating system differences). This is important because large systems outlive the computing environments (especially hardware) for which they were initially developed. Portability is supported by modular decomposition and information hiding, as well as by standard coding practices to isolate machine dependencies. At the source level, it is achieved by using standard languages.

AI applications are notoriously non-portable. This is primarily because of their dependency on specialized development environments. Multiple paradigms, although important for research, can be a handicap for large-scale development; this constrains the range of possible solutions, and recommendations cannot be made to accommodate specific AI paradigms. Lack of standardization can only be detrimental to large-scale projects for which a significant use of AI techniques is expected.

The capability to adapt the categorization algorithm to a wide range of domain-specific profiles, from simple to complex and from general to specific, is also desirable. Software engineering principles of abstraction and modularity play an important role here. Knowledge bases are difficult to integrate with conventional databases that are large-scale and interoperable with exiting commercial products. Variations in input text format may impede the use of a system across dissimilar platforms. Data interoperability depends on standard protocols and system software interfaces.

2.2.2.3 Maintenance

Because few large-scale systems have been fielded, little maintenance experience is available for AI applications. If large-scale AI applications are to be manageable, knowledge maintenance for a long life cycle is an area of critical concern.

Maintenance of large systems covers both enhancements and corrective tasks, which may better be termed *continued development*. It usually involves many incremental changes to the operational system. Simple changes often trigger modifications in many other modules, sometimes at great expense, and additional design effort is needed to minimize this ripple effect. Software engineering principles directly supporting this goal are modular decomposition, information hiding, and abstractions.

2.2.2.4 Performance

Performance requirements are those issues that determine system usability. This is generally measured in terms of two factors: accuracy and resource utilization.

Accuracy refers to the level of success in profiling information against a set of preestablished

categories. Accuracy is commonly measured in terms of levels of precision and recall. High levels of recall imply that less false negative mappings were made, whereas high levels of precision indicate that less false positive mappings occurred. Precision is actually a measure of correctness—to what extent a user obtained correct messages and not incorrect ones. Recall serves to measure completeness—whether or not all corresponding messages were disseminated to a respective user.

Two commonly used measures of computing *resource utilization* are storage space used and execution speed. Space is relatively important if large quantities of information must be processed. For many embedded applications, timeliness (more than simply speed) is a crucial requirement. For example, the timely distribution of information may be important for making mission-critical decisions.

AI languages are NOT suitable for real-time programming. Real-time performance is also poor for typical interpreted AI languages. Typical AI languages are at a higher level than more traditional procedural languages, but they are not suitable for real-time embedded programming. These languages do not have adequate constructs to handle timing and synchronization constraints, and parallel and distributed processing; they are weak in handling numeric types and strong typing in general. Even extended, newer versions of LISP (e.g., CLOS) do not address the issues of real-time computing. Procedural languages require fewer system resources than their non-procedural counterparts.

Considerations on the ramifications of using exponential time algorithms as well as reliability and fault-tolerance concerns are generally lacking in AI approaches to real-time computing. Inheritance, especially dynamic polymorphism, is not good for real time because of its exponential nature and runtime support requirements. An intrinsic concern is the fact that AI algorithms' performance degrades exponentially, and worst-case analyses would probably result in totally unacceptable response times. Multiple inheritance and garbage collection further exacerbate the problem.

2.2.3 Functional Capabilities

AI has been viewed as an approach to problem solving that uses a physical symbol system as opposed to a numerical system [Newell 72]. Newell describes a physical symbol system as one consisting of a set of entities, called *symbols*, which are physical patterns that can occur as components of another type of entity called an *expression* (or *symbol structure*). Thus a symbol structure is composed of a number of instances (or tokens) of symbols related in some physical way (such as one token next to another). At any time the system will contain a collection of these symbol structures. Besides these structures, the system also contains a collection of processes that operate on expressions to produce other expressions: creation, modification, reproduction, and destruction. A physical symbol system is a machine that produces through time an evolving collection of symbol structures. Such a system exists in a world of objects wider than just these symbolic expressions themselves. Such a system has the necessary and sufficient means for general intelligent action.

Because AI programmers are geared primarily to symbolic processing rather than numeric computing, special AI languages have been developed.

What kinds of problems can be solved with AI that cannot be solved by traditional numeric methods? To answer this question, problems have been divided into two broad categories, namely *easy problems (P)* and *hard problems (NP)*. Easy problems can be solved in polynomial time on a deterministic machine, and the solution time is linearly bound to the size of the input data. That is, given a problem with x amount of data to process, the time required to complete the process can be expressed as |x| raised to some fixed power.

Hard problems can be solved in polynomial time on a non-deterministic machine (these machines do not exist!). The solution time increases exponentially relative to the size of the input data. The solution time is related to the amount of data according to n raised to the x power, where n is fixed and x is the amount of data. Combinatorial time explosion makes such problems intractable on conventional deterministic computers (parallelism does not help here, since they also grow linearly). If an approximation can be used, symbolic (i.e., AI-based) solutions would have to be considered.

Any successful software technology for a "plausible" solution to intelligent embedded systems must be able to deal with the following (possibly contrasting) requirements or capabilities.

- *Symbolic processing.* A physical symbol system, as opposed to a numerical system, is used for problem solving.

  This requires flexibility in the data structures and control flow functions of the program. For example, conventional programming determines all permitted relations among data a priori, whereas AI programming determines some of the permitted relations among data statically, but the AI programs themselves determine those that were not explicit in the program.

- *Adaptive behavior.* The system can "learn" from its environment and change its "programmed behavior" dynamically to respond appropriately to new situations.

- *Introspection.* The system must be able to balance its own computational load based on dynamic scheduling requirements and reactive actions. This involves tradeoffs between complex, often conflicting mission goals.

- *Non-monotonic reasoning.* Incoming data does not remain constant, and later facts or conclusions can invalidate earlier conclusions.

- *Temporal reasoning*. Temporal relationships, including past, present, and future, between events and activities (for example, in the context of planning) are important for arriving at a correct solution when making a best guess given a deadline.

  The best solution that comes too late is worse than a weaker solution on time. A solution must be found quickly, but it must also be a good solution.

- *Predictability and Reliability.* Uncertainty in both data and the deductive engine are intrinsic features of AI intelligent control.

  The system must also be capable of continuous operation, which implies close control of garbage collection. This reflects the capability of the system to keep

operating despite the presence of faults, which also implies uncertain or incomplete data (noise) from faulty sensors. This is specially important for embedded applications that must be reliable, robust, and fault tolerant.

- *External interfaces.* Asynchronous events and relationships between logical and physical objects to the space they inhabit must be maintained.

  Embedded AI applications must process data from external autonomous agents representing sensors, other non-AI components, other AI components, and sometimes human operators.

Developing intelligent systems using typical AI technology may impose a critical bottleneck in applying knowledge-based techniques in embedded real-time computing. The current technology base to support integration of AI with mainstream real-time software engineering is basically inadequate. Conventional AI technology generally lacks facilities for interoperating with non-AI computer software, for dealing with real-time settings, or for integrating several independent AI applications into a cohesive system.

3 Implementing AI using Ada

In this section we discuss engineering issues and software challenges that span the complete software life cycle of developing AI applications with Ada. These include domain-specific technical issues such as

- The difficulties with (real-time) AI programming.
- The difficulties with requirements analysis, and with design methods that adequately encompass rapid prototyping, testing, validation and verification techniques, and maintainability of long-lived systems.
- The implementation of AI applications using Ada.

In this section we also present an overview of AI applications that have been developed with the Ada programming language. These applications are sufficiently large and mature to provide valuable insights into the experience of AI with Ada. Many of these applications are expert systems. We provide a general assessment of the current state of expert systems development and discuss the experiences of developers of Ada expert systems.

There is no question that deployable AI-based systems must be explainable and understandable. This is not easy; consider for example chess-playing programs. What do they do? How do they do it? And why do they have to do what they do to play chess?

Is AI programming very different? Studies have shown that at least 90% of the code of the most advanced AI systems are procedural in nature! One important fact is that AI research is fundamentally about prototyping systems. Abstraction mechanisms and objects are important for AI software development, particularly persistence and dynamic typing! In terms of the computational concepts they employ and depend upon, they all focus on the manipulation of symbolic information. Most development is based on two well-founded computational models: functional (or applicative) programming and relational (or logic) programming.

Why do we want to program AI applications using conventional procedural languages? How do these features help or hinder AI software development? Modern procedural languages, like Ada, possess many features expressly designed to support system integration and large-scale software development. AI technology is quite poor in these two areas. In addition, orders of magnitude in performance improvement can readily be achieved. Experience has shown that performance is improved by using the approach proposed by Ada, such as programmer-controlled memory management (“manual” memory management rather than automatic garbage) and strong typing (automatic type checking rather than programmer imposed checking). Typing is an area in which most procedural languages can offer automatic runtime improvements. Strong typing allows checking to be done at compile time, thus avoiding many runtime checks. And the system is not performing automatic type conversions! Inheritance (and particularly multiple inheritance) is another source of inefficiencies, particularly when combined with dynamic binding and dynamic typing.

In what follows we contrast these two very different software development approaches: con-

ventional software development and AI software development.

Embedded, real-time AI systems must also be well engineered, following established practices of real-time software engineering. Large potential cost savings and risk reduction would be possible through the development of architectures, interfaces, and components that can be effectively reused and reengineered for multiple uses. We have studied the suitability of the Ada technology for the development of AI applications and have found that although there are some impediments to a proper solution to the kinds of problems typically addressed in this domain, current Ada (Ada83) is indeed able to support most of these AI techniques [Diaz-Herrera 93]. It is also important to point out that these problems are much less severe in Ada than when using less powerful languages, such as C or Pascal. Furthermore, any remaining shortcomings are being addressed by the proposed changes to the language, collectively known as Ada9X.

3.1 Conventional Software Development with Ada

Software development, the process of going from system conception to deployment, is the essential component that controls the successful deployment of an application. In many instances it even defines the application. For several years there has been considerable general discontent with the process of designing and producing software and the quality of the products delivered. This, the so-called software crisis, is one of the most important problems currently facing the software community at large.

A response to this crisis has been the emergence of the discipline called *software engineering*. Modern software engineering activities work in concert toward the achievement of preset software engineering goals that, if attained, will lessen the effects of the software crisis. A number of software engineering principles have been identified that contribute to the achievement of these goals.

3.1.1 Software Development Process

The conventional software engineering approach is characterized by a well-defined development process that, together with a variety of methods and supporting tools, specify a software life cycle for producing a deployable system. The process refers to a set of activities and products that, in general, specify a transformation of problem requirements into software structures.

A widespread process model is the well-known *waterfall model,* illustrated in Figure 3-1a. This model, which evolved over the past few decades, prescribes a serial sequence of activities for a progressive software production. In addition, a set of review points or managerial milestones is also defined for checking the results of the activities to verify their consistency and to validate the products against the original requirements.

The basic waterfall model suffers from a lack of early feedback on the products (especially with respect to executable functionality), and an unrealistic separation between requirements

specification, design, and coding. These deficiencies are primarily due to the strict sequential nature of the activities and a potentially long period before any system functionality can be experienced. The introduction of prototypes in the early activities alleviates these problems. Rapid prototypes are generally of two kinds, namely *throwaway* and *evolutionary* prototypes.

A more formal model incorporating this notion of prototypes is the *spiral* model [Boehm 88]. This approach calls for an iterative development cycle (see Figure 3-1b), whereby rapid prototypes (p1) are used in the early stages of development, followed by a series of evolutionary prototypes (p2, p3) that, when incorporated into incremental "builds," converge into an operational prototype and finally into a deployable system. The spiral model provides considerable flexibility and is in fact a generalization of the waterfall model and incremental development.

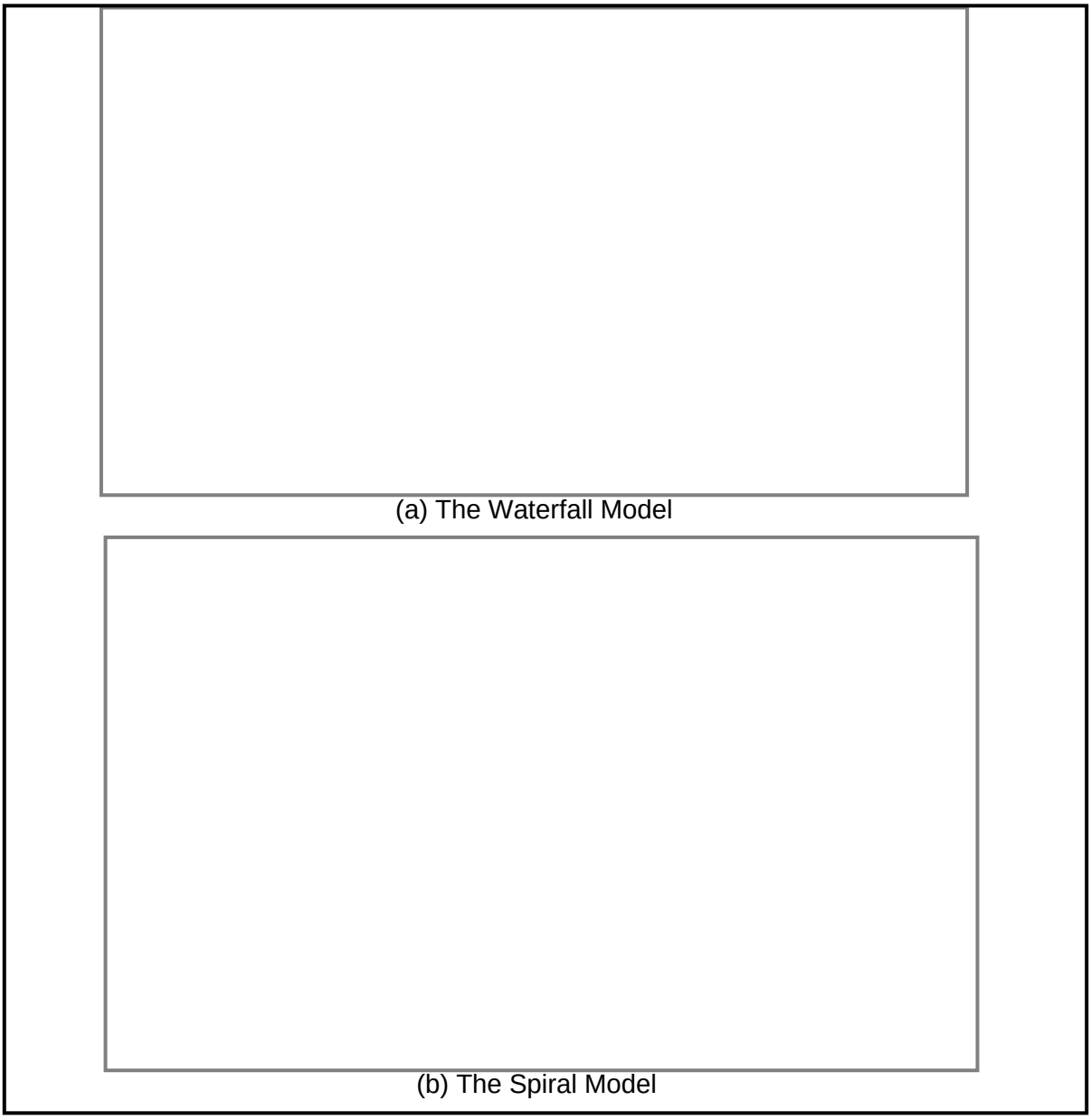

Figure 3-1 Conventional Software Development Process Models

3.1.2 The Ada Language

The state-of-the-art software engineering language is, in many respects, Ada. The language directly addresses specific issues in the construction of large embedded systems. The solution to these problems is of paramount importance to the industrial base in general, and to the military establishment in particular. Both are inextricably tied to the software element of computer systems. Ada represents the cornerstone of the U.S. Department of Defense software initiative in its latest attempt to focus all software development on a single standard language. Ada, unlike earlier programming languages, embodies a collection of current knowledge of

software engineering and a modern view of the process of developing large programs, thus incorporating specific constructs directly supporting these software engineering principles.

The development of Ada was also motivated by the software crisis. Ada is much more than another programming language; it is a robust and proven technology specially designed to support well-engineered software. The language supports modern software engineering, risk reduction, and several development paradigms, including object-oriented programming. Initially intended for embedded real-time systems, it meets a wide spectrum of needs and has proven to be suitable in many dissimilar application areas ranging from commercial data processing through artificial intelligence.

Although software engineering principles and activities are generally language independent, Ada is becoming the first widely available standard language especially designed to directly support these principles and activities. An Ada compiler must comply with the ANSI-MIL-STD-1815A standard, which must be implemented in its entirety, and nothing can be implemented that is not in the standard. The language validation and certification process tests implementation conformance to the Ada standard, not performance. Validation also identifies behavior that is implementation dependent. A compiler must correctly process the entire validation suite and demonstrate conformity to the standard by either meeting the *pass* criteria given to each test or showing inapplicability to the implementation.[1]

The basic focus is on the enhancement of the environment in which software is developed and "maintained"; the overall goal is to substantially reduce the cost of developing large software systems. Among the design goals of the language, the concern for programming as a human activity takes paramount importance. For example, program readability is a much more prominent design goal than program writing; strong typing and programmer-controlled runtime conditions directly support reliability; good control and encapsulating structures together with powerful data abstraction facilities support the goal for modifiability.

Productivity is enhanced by

- The direct support of parallel development (top-down/bottom-up incremental integration) through language features.
- Compiler-enforced separate compilation and module obsolescence control.
- Facilities to interface with other languages.
- Features directly supporting software reusability.

Other weighty goals are those of portability and transportability of source Ada programs. Machine-dependent features are clearly marked in Ada itself, as are programmer-defined physical representations of objects and disciplined (and quite elegant) access to low-level hardware features. Object representations include storage layout, addresses, and literal values. I/O operations are not an intrinsic part of the language, but are defined as standard library units whose implementation dependencies are clearly marked in an appendix of the reference man-

1. Ada9X is destined to become the first international standard object-oriented language.

ual. This also supports portability.

The language, in effect, provides a “software bus” further supported by an also standard programming support environment, an area in which the language is unique. A flexible tool interface, known as the common APSE (Ada programming support environment) interface set (CAIS) [Oberndorf 86], has been approved as an accompanying standard.

3.1.2.1 What is Ada like?

Ada is a language with considerable expressive power that includes facilities offered by several more classical languages as well as features found only in specialized languages, and in this sense it is a multi-dimensional language. It is sequential or block-structured like Pascal, hierarchical or module-structured like Modula-2, concurrent or process-structured like CSP (communicating sequential processes), and low-level for machine-oriented programming. It is important to emphasize that the language is not a bundled collection of heterogenous features coming from these different languages, but instead, an orthogonal and homogeneous language providing important features found in these other languages. The following is an outline of the language’s main concepts.

Program Units. In Ada, a software system is organized architecturally as a hierarchical collection of program units. Program units are the basic tools for manipulating program text and for the control of visibility. Each unit is defined, and provided, in two parts, namely a specification and a body. Entities declared in the specification are visible (i.e., exported) to other units, whereas entities local to the unit's body are invisible (i.e., non-exported) outside the unit. These import/export (I/E) aspects of units have an effect at both compile-time and at runtime. Compile-time I/E controls visibility, whereas at runtime it controls existence.

The language defines three kinds of program units, namely subprograms, packages, and tasks, in order to allow the specification of distinct approaches to the dynamic behavior of the software system. *Subprograms* provide procedural abstractions, in the form of procedures and functions, which define operations applicable to objects, passed as parameters. Subprograms enforce traditional block-structured scope rules; a subprogram interface simply specifies the names, types, passing modes and default values of parameters (if any). They can be *overloaded* (more than one subprogram with the same name in the same scope if their profile is sufficiently different in at least one argument), and can be used to specify programmer-defined operators.

Packages serve as organizational units providing a higher-level abstraction mechanism characterized by the exported entities (they can export objects, types, and even other units.) Packages, obviously a key concept in Ada, provide means for the separate control of visibility and existence. Unlike subprograms, packages do not really exist at runtime! They provide encapsulating environments affecting visibility (a compile-time phenomenon).

Tasks are like specialized packages in that they encapsulate concurrent sequential processes (a “program” executing sequential statements activated implicitly) with an interface specifying

a message-based communications protocol (entries); tasks provide a mixture of procedural and abstraction capabilities.

Incremental software development is directly supported by separate compilation facilities of library units. Bottom-up development is supported by the notion of library units. A *library unit* is simply the stand-alone compilation of a subprogram or package specification. Their corresponding bodies, or secondary units, may also be provided in separate compilations. The body of a (local, i.e., non-library) program unit may also be compiled separately as a subunit, in which case, a “body stub” is placed in the enclosing unit at the place where the actual body would normally occur. In this way, development takes place top-down.

3.2 AI Software Development with Ada

In this section we present

- Detailed descriptions of how to implement in Ada the basic AI programming techniques and approaches.
- Empirical evidence accumulated by the Special Interest Group for Ada (SIGAda) AI Working Group and the AI and Ada Conference Series (AIDA) of six conferences showing the successful implementation of large-scale AI systems in Ada.

We also illustrate specific aspects of real-time AI computing, showing what can be achieved today and what remains to be solved.

AI software development tends to have a short development cycle and rarely involves teams of programmers. These systems differ from non-AI systems in several significant ways; typically, they are built without a well-understood specification through a process known as “exploratory development and transformational implementation” [Agresti 86]. See Figure 3-2a.

Support for software engineering is being recognized as an important criterion when selecting an implementation technology. NASA [Gilstrap 91] has put together a risk-based methodology for the development of expert systems (ESDM) that incorporates the concerns of managers and developers. Since expert systems requirements are not known at the project outset, the methodology focuses on driving out and validating requirements.

The ESDM life-cycle model is based on the spiral model, discussed earlier, formalized into five stages. See Figure 3-2b. The main goal of each iteration is to add knowledge about what the human expert does and what the requirements should be.

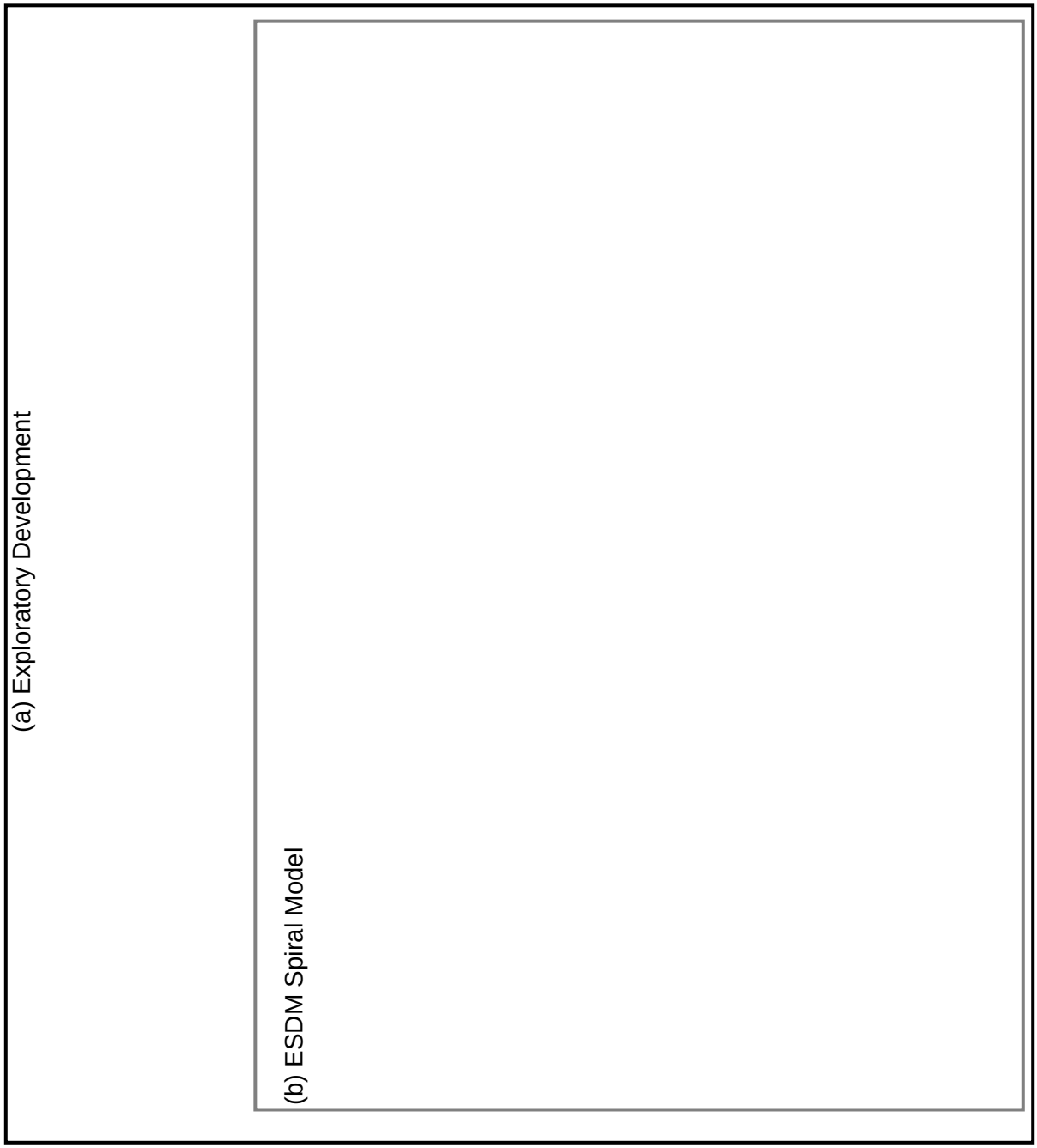

Figure 3-2 AI Software Development Process Model

The five stages are

1. *Feasibility*: to demonstrate that the key functions of a manual system can be automated.
2. *Research:* to demonstrate that sufficient functionality can be implemented to produce a useful expert system.
3. *Field:* to demonstrate that an automated expert system can be built to function in a realistic setting.

4. *Production:* to demonstrate that a robust and reliable version of the system is feasible.
5. *Operational*: to demonstrate that the risks of both construction and use of the expert system are acceptable.

The important thing is that breaking up the total development task into stages[2] helps the manager of an AI project because each stage has a specific set of objectives that can be monitored and evaluated effectively.

AI has a wealth of languages, techniques, and methods of its own that have evolved since the early 1960s. These include dynamic data structures, object-oriented and frame-based programming, and non-procedural approaches such as functional and relational programming.

3.2.1 AI Basic Techniques

Basic AI techniques include dynamic data structures, object-oriented and frame-based programming, and non-procedural approaches such as functional and relational programming. As we describe later, these are all supported by Ada concepts. With these techniques, each of the independent AI components of a system can evolve through a typical AI exploratory development while the integrity of the complete system is maintained.

Studies have shown that Ada outperforms interpreted LISP by executing orders of magnitude faster [Miles 89].

Ada can also be effectively used for system integration. For example

- To develop and verify well-defined interfaces for the exchange of information between the independent AI components of a system.
- To specify functional behavior and implement an executive that controls and coordinates the activities of the varied (AI/Ada, non-AI/Ada, and AI/non-Ada) components of an application.

In what follows we describe this Ada83 support for AI techniques. We defer any discussion on the new, more powerful features of Ada9X for later.

3.2.1.1 Frames

From a more pragmatic point of view, these techniques are intrinsically symbolic rather than primarily numeric. Many of these symbolic techniques are generalized in the notion of *frames,* which are flexible data structures treating data as objects. Frames form the basis for knowledge representation techniques. The idea of keeping together facts about a “concept” and linking these representations is of paramount importance for developing AI systems. The frame, first introduced by Minsky [Minsky 74], was defined for such purposes. Frames are used to represent knowledge in terms of semantic networks as combinations of data structures with procedural attachments leading to “intelligent behavior” when used appropriately.

---

2. These could be fewer or more than the ones listed above according to the specific project needs.

Frames' slots can be used not only to keep attribute values but also to connect frames in a number of ways. In this way, frames provide an organizational scheme for knowledge bases. Such a system of connected frames is known as a *semantic network*. An early example of using Ada to implement semantic nets is described by Scheidt [Scheidt 86].

Many AI techniques are generalized in the notion of frames. Frames are used to implement adaptive programming styles such as functional programming, logic programming, and object-oriented programming. Frames are also used to implement object-oriented programming by combining inheritance relations with message passing. In this way they can be made to represent many more of the details of knowledge relevant to the problem without losing organizational effectiveness. These techniques require fundamental features such as dynamic memory allocation, late binding, and relaxed typing that together with rule-based systems and constraints allow the building of blackboard architectures and planning systems. Of particular importance is the capability to create frames arbitrarily composed of each other and with slots of any type. Frames are thus very flexible heterogeneous data structures. See Figure 3-3.

Frame structures have been successfully implemented in Ada [Walters 87]. Ada provides specific language mechanisms, known as *private types*, which unify the representation and the operations of programmer-defined data types. In this way, a frame data structure is represented by a package encapsulating a "slot" record definition and associated set of subprograms for creating/accessing/setting "slots" and their corresponding attributes.

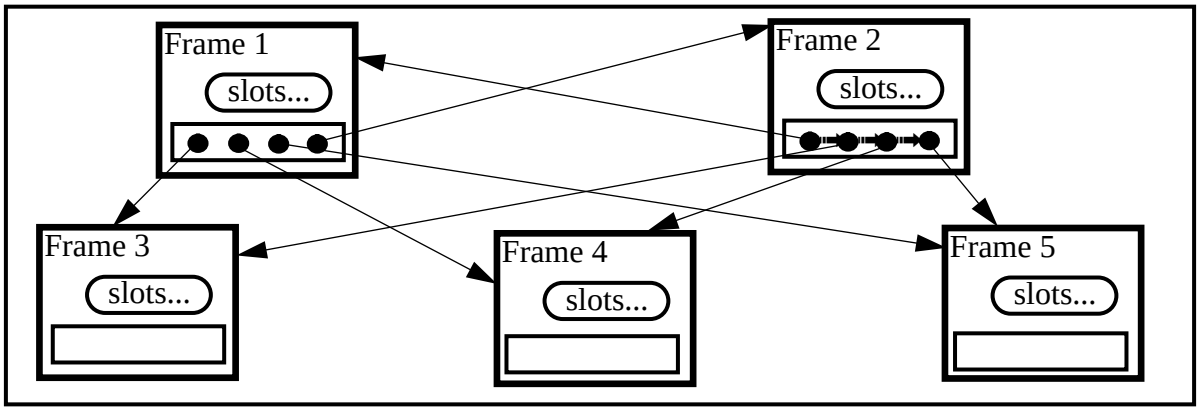

Figure 3-3 Frame Structures

3.2.1.2 Lists

A frame is fundamentally a list of lists. The capability of dynamically creating/deleting data structures is, obviously, at the heart of any proposed linguistic solution. Dynamic data structuring in the form of *list manipulation* is done in Ada basically as it is done in LISP by using access types encapsulated with a set of associated subprograms in a package.

The properties of pure LISP are (see Figure 3-4)

- The basic primitives CAR, CDR, CONS, EQ, and ATOM.
- The control structures using COND, recursion, and functional composition.

- List structures containing only atoms and sublists.
- A means of function definition.

LISP focuses on the description of values that functions produce, not on steps performed to convert an input into an output. The basic LISP data object is the atom, a named identifier whose type and contents are determined at runtime (late binding). LISP lists are singly linked lists, in which each element contains two pointers, namely CAR (points to the atom associated with that list element) and CDR (points to the rest of the list). The function CONS take two operands and produces a new list. EQ and ATOM are predicates returning T (true) or nil (false).

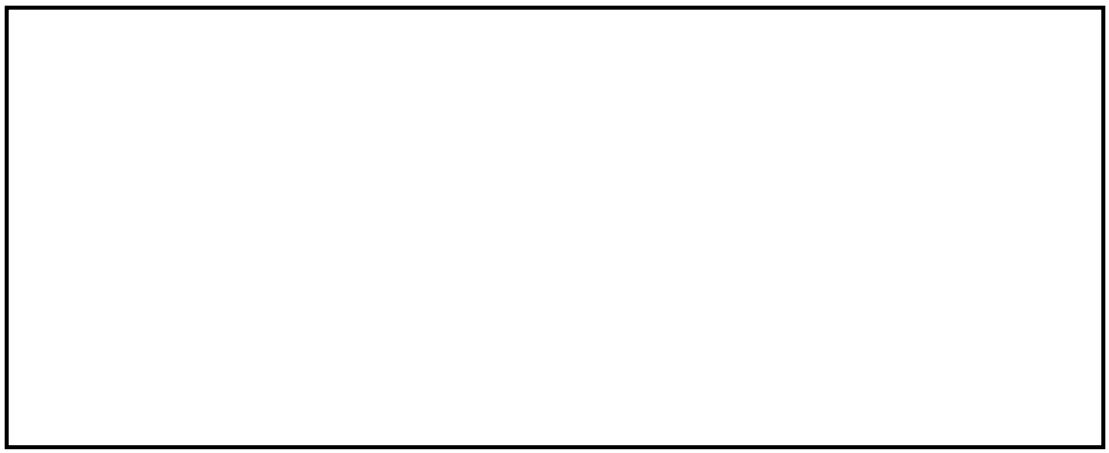

Figure 3-4 Pure LISP Basic Elements

Objects in Ada are strongly typed; thus, before an operation is performed, its operands are checked, at compile time, to ensure that they correspond to the appropriate type. This prevents nonsensical operations, but more importantly it allows overloading resolution (see below). The facility of generic formal type parameters in Ada, allows the creation of “type-less” algorithms since the generic specification is defined for no specific type, but for a family of related types. This delays the actual association of a specific type until the generic instantiation.

The use of Ada objects referred to by access types is much safer than similar concepts in other languages. When using access types, objects are continuously taken off free memory. The problem with dangling references does not surface with Ada access types, since the accessed objects form a *collection* whose scope is that of the access type. This collection disappears when the scope is exited, and by then any corresponding access object would also disappear; so there would be no possibility for an access object to point to or to reference a nonexisting object. Ada access types are also strongly typed; the objects accessed by a given access type are all of a given specified type (the collection’s type). Ada programmers can directly control the size of the space used for each specific collection.

When a dynamic Ada object is inaccessible (because no other objects refer to it either directly or indirectly), the storage that it occupies may be reclaimed by an automatic garbage collector. For Ada, automatic garbage collection is not required; however, the Ada standard does not preclude it either, and an implementation is free to provide it. Alternatively, implementors of

packages providing dynamic data structures may provide in the package body a uniform garbage collection mechanism by explicitly deallocating no longer needed objects and keeping track of the space thus made available [Yen 90]. The Ada feature supporting programmer-controlled storage allocation is the generic UNCHECKED_DEALLOCATION library procedure. This is the preferred approach since not only is it more implementation-independent, but it also gives better timing control in a real-time situation.

3.2.1.3 Objects

Objects are an important development that stem from frame structures to impose encapsulation.

The most basic tenets of the object-oriented paradigm are encapsulation, hiding, polymorphism, and inheritance.

Ada packages represent the chief mechanism for encapsulation and hiding. The binding of underlying data with an associated collection of subprograms is called *encapsulation*. Inaccessibility to the internal structure imposed by the separation of the specification and its body is called *hiding*. Ada packages are object-oriented in the sense that they can export definitional means for creating instances of objects and a set of subprograms that operate on those objects, while keeping the objects' state variables totally hidden in the package body. A single object can be implemented in Ada as a package. Methods are implemented as Ada operators, subprograms, and built-in attributes. The capability to instantiate multiple objects can be obtained in a number of ways. The simplest approach would be to make the single-object package a generic package. Alternatively, a package exporting a type and a set of related subprograms implements the idea of a class, and is further supported by strong typing; in this sense, objects of a given type can only respond to operations defined for that type. Object instances are created by Ada object declarations from the given type.

Overloading refers to the ability of a subprogram identifier (or operator) to denote several methods simultaneously (within the same lexical scope), each specifying distinct actual operations. This notion has also been termed as ad hoc polymorphism.

Procedural abstractions that operate uniformly and unambiguously on values of "different" types are said to be (parametric) polymorphic abstractions. This is done in Ada by defining these abstractions in terms of generic formal types.

Class specialization can be done by adaptation from parameterized generics components or by inheritance. In its simplest form, inheritance is available in Ada in the form of subtypes that inherit their properties from their base types. A more powerful form of inheritance in Ada is that provided by derived types, another form of defining a new type whose definition is inherited from that of an existing "parent" type. The definition may add constraints to the properties inherited. The derived type can change base type attributes by applying representation clauses. It can also add operations to those inherited or provide new methods for any of the inherited operations (by using overloading). This is also known as inclusion polymorphism.

An important consideration not supported by Ada in a straightforward manner is the capability to extend the data structure of the inherited type by adding new fields. This, however, is fully supported in Ada9X as described later.

There are several factors other than data structures that characterize AI programming. These include symbolic manipulation, pattern matching, procedural attachment, and adaptive control flow computations, as well as interpreted and declarative languages, and environments with higher level debugging facilities.

3.2.2 Functional Programming

A *function* is a computation that returns a value when supplied with the appropriate input. Since its original definition by McCarthy in the late 1950s, LISP has been heavily enhanced and modified into many dialects [Lifschitz 91]. The result is that the current CommonLISP is a massive language by any standard, but it has solved, to a large degree, the problem that resulted from the proliferation of incompatible dialects.

Facilities for LISP-like *pattern-directed computation* on list structures have been developed in Ada [Reeker 87]. The operation that drives a pattern-directed computation is that of finding a pattern in the data, and is generally identified with the processing of character strings. A pattern determines the structure of the string to which it is matched.

Adaptive control flow refers to the capability of defining algorithms recursively in terms of themselves, and the *dynamic* association of executable code with procedural (or functional) abstractions. Ada provides very good facilities for recursive programming. Both procedures and functions can be called recursively. Furthermore, because of the separation between a subprogram declaration and its body, mutually recursive subprograms are possible. Functions are also capable of returning any complex data structure.

In a frame system, a slot may be provided with default value and/or with information related to that slot. This attached information can take several forms. It may be a constraint that must be satisfied by the filled-in value for the slot, or it may be a *procedural attachment* (also known as a “demon”) used to determine the value for the slot or triggered after a value is filled in for the slot. This attachment of information to slots provides great flexibility. In this way, programs are treated as data. The attachment of data to slots also enables the creation of self-generative code—a technique used in LISP and other interpretative languages in which a function or program segment is developed at runtime and structured around applying functions to linked lists of arguments that may themselves be functions. This means that programs can be modified as data, and data structures can be constructed and directly executed as programs. LISP permits this type of programming by requiring an interpreter.

This issue is important, since *functional programming* is a pervasive style found in LISP. Ada does not offer this capability. Ada does not allow the dynamic association of names with executable statements.[3] However, since it is always possible to write an Ada program that interprets an array of symbolic instructions, this is a non-issue. Furthermore, self-modifying

programs are mathematically undecided and may have unpredictable results, thus precluding verification and validation requirements! They therefore should be avoided. It is also true that the great majority of AI applications do not include self-modifiable code.

Mimicking functional structures in Ada is beneficial. A limited, more controlled, similar effect can be achieved using generic formal subprogram parameters. This allows the association of different actual subprograms for each different generic instantiation; the corresponding unit bodies must be available at link time at the latest. In this way, function-forming operations (a function that takes another function as input) can be provided in Ada by using generic functions with other functions as generic parameters.

There have been several automatic translation systems from LISP into Ada; Baker provides a good discussion of the topic [Baker 90]. Strong typing has been one of the major problems to be solved in such translations. The use of predefined library packages makes the translation more straightforward, and in actuality lessens any concerns about “loosing” the LISP environment.

3.2.3 Logic Programming

Relational languages such as PROLOG are generalizations of the functional language model and were originally developed using functional languages. These languages free programmers from the need to specify “how” even more than functional languages do. They began in theorem-proving research. In the late 1960s, the unification algorithm and resolution principle were developed, and the control structure of a program was merged with the operations of logic manipulation.

Another important tool is the idea of programming based on primitives for defining facts and rules, known as *logic programming*. Although rules can be represented using a frame structure, special languages and environments have been developed, notably PROLOG. This formalism is based on Horn-clause logic. Rules are statements of knowledge in IF-THEN like constructs with antecedent and consequent; when the antecedent is satisfied, the consequent is enacted or inferred. The knowledge base is a collection of rules.

An inference engine is a control mechanism that applies rules to data received (from external source or internally generated) (See Figure 3-5). As a rule is conditional with an antecedent and a consequent, the interpretation of a rule is that if the antecedent is satisfied, the consequent is enacted if it is an action, or inferred if it is a conclusion. The basic elements are the unification component and the working memory. The unification (or resolution) algorithm can be thought of as a pattern matcher that identifies all of the rules or conditions that are satisfied by a given piece of available data. The consequent of each satisfied antecedent is then enacted. If the consequent generates new data, the data is stored in the working memory for unifi-

3. Actually, this is only possible if the language involved is the “native language” of the underlying computational engine, whether it be hardware or software. Of course, if there is an Ada machine that directly executes Ada code, either virtually or physically, dynamically definable functions could be possible in Ada.

cation during the next cycle. If the consequent is an inference, this forms an output of the inference engine. The processing continues until the unification algorithm is not able to identify any antecedents that are satisfied by the available data.

Figure 3-5 A Simple Inference Engine

Variations include forward and backward chaining. The IF-THEN rule format is very common. Systems using this format are known as forward chaining because they proceed from the antecedent to the consequent. PROLOG (and other pure Horn-clause logic) are termed backward chaining because the rules are written in consequent-antecedent form and because unification proceeds by applying data to the consequent.

These rule “interpreters” are purely procedural programs, and several have been written entirely in Ada [Lander 86, Bobbie 87]. Other systems supporting logic programming have been designed and implemented in Ada, notably ALLAN [Ice 87] and PROVER [Burback 87] and several expert systems tools [De Feyter 88, Wright 86, Martin 89], some of which are commercially available such as ART-Ada [Charniak 87], CHRONOS [Collard 88], and CLIPS/Ada [NASA 89]. Some of the implementations using tasks allow for concurrent queries to be handled graciously by the Ada runtime kernel. They also permit the association of timing constraints, and thus control of the use of resources spent in solving the queries [Kilpeläinen 89]

The RTEX system [De Feyter 88], an industry-oriented system for developing embedded real-time expert systems, is particularly interesting. The system integrates advanced software engineering and real-time concepts with object-oriented data-driven concurrent programming, real-time inferencing, symbolic matching, and signal understanding.

Finally, the idea of providing development environments for Ada that have the same flexibility as those found in AI shops has been suggested, and such environments have been developed [Lee 90, Martin 89]. The capability of merging Ada technology with AI has been shown [Fornarino 89]. Rules are translated to an intermediate form from which actual Ada or LISP code can be generated.

3.2.4 Blackboard Architectures

The blackboard architecture is a collection of modules with special attention paid to information exchange. Conceptually, it is synonymous with global memory. The basic blackboard ar-

chitecture consists of a shared data region called the blackboard (BB), a set of independent knowledge sources (KSs) or experts, and a control unit called the *scheduler* (see Figure 3-6).

The BB is actually a very flexible shared data structure, the sole repository for global data, and the only communication path between the components. It is the part of the blackboard system that is used for storing knowledge accessible to all the KSs. It is a global structure used to organize the problem-solving data and to handle communications between the KSs. The BB contains synchronization primitives to schedule the KSs. The objects that are placed on the BB could be input data, partial results, hypothesis, alternatives, and the final solution. Interaction among the KSs is carried out through the BB. A BB can be partitioned into an unlimited number of sub-blackboards. That is, a BB can be divided into several BB levels, each one corresponding to different aspects of the solution process.

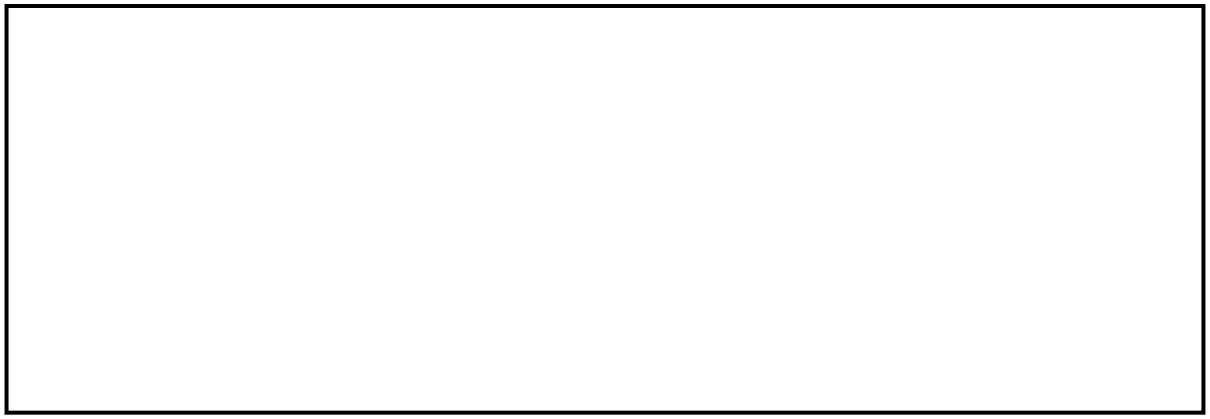

Figure 3-6 The Blackboard Architecture

The KSs are the self-selecting modules of domain knowledge. Each KS is separately compiled and possibly programmed independently with its own execution thread with unknown identity. They are processes that retrieve/deposit data from/to the BB. Their execution can be data driven, i.e., the KS waits for critical data to appear on the BB. KSs can be added freely, and failure of a KS does not invalidate design or integrity of the system. Each KS can be viewed as an independent program specializing in processing a certain type of information or knowledge of a narrower domain. KSs can allow different kinds of knowledge representation and different inferencing techniques. Each KS should have the ability to assess itself on whether it should contribute to the problem solving in any instance of the problem-solving process. The action part of a KS performs the actual problem solving and produces changes to the BB.

The BB scheduler serves as a control unit: when an event triggers some KSs that need the newly posted information, testing procedures should be performed on those KSs to determine whether they are eligible for execution. When a triggered KS is chosen to be executed, it adds or modifies information on the BB, which in turn triggers a new event. The cycle then starts again. Hence, the task of the control unit is to identify the set of permissible next computations, select one among them, and then execute it.

Frames, together with rule-based systems and constraints, are sufficient to build blackboard

architectures. Conceptually, parallel implementation on shared memory fits and can be distributed. From a pragmatic viewpoint, the BB is an object accessed through a BB manager that schedules data transfers and removes data that is no longer needed. Knowledge organization and resource contention are potential bottlenecks.

Blackboard systems have been implemented in Ada [Stockman 88]. Frame structures are used to implemented needs functionality; Ada tasking can be used to allow several interacting BBs to coexist. Current Ada limitations prevent the implementation of procedural attachments; it is important to notice that this can be circumvented by calling a routine in another language to perform the actual dispatching, using the ‘ADDRESS attribute of the subprogram to be actually executed. This solution is far from appropriate since it increases complexity, and an all-Ada workaround is preferred which can be achieved by using record variants and case statements.

It is also important to mention that the new proposed revision of Ada83, the Ada9X version, has direct mechanisms for achieving runtime dispatching.

3.3 SIGAda AIWG Survey

The ACM SIGAda AI Working Group (AIWG) was formed as a response to an increased awareness of the role played by AI in complex systems, especially in the areas of command and control. Results from an earlier survey conducted by the AIWG are summarized next [Johns 92a]. Of the 34 surveys responses, 17 provided data about the nature of their AI applications written in Ada, as summarized in Figure 3-7.

Figure 3-7 AIWG Applications Survey Summary

Notice that source code is measured in thousands of lines, whereas object code is measured in millions of bytes. The average size of applications was computed to be 85.7 KSLOC, ranging from 1 KSLOC to 1,000 KSLOC. Code production rates were computed from the data pro-

vided in the surveys producing an average of 6.263 KSLOC/person-year! Insufficient data was available to work with the object sizes for the applications (only seven respondents included object sizes).

Products were categorized as commercial, public domain, or internal use; most applications are commercially available (40%), whereas the smallest number of them were in public domain (2%) (See Figure 3-8).

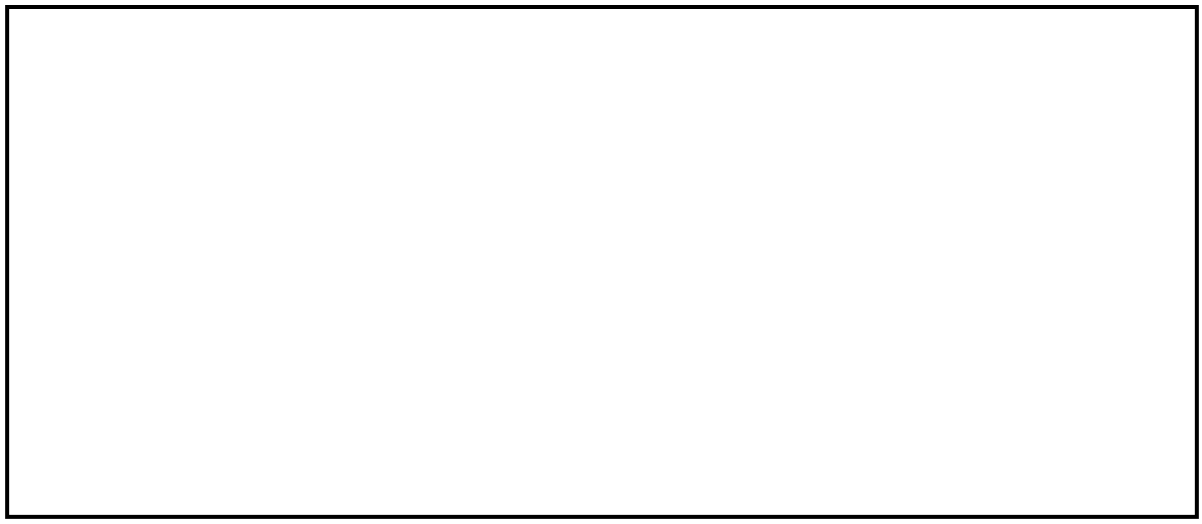

Figure 3-8 AI-Ada Product Availability

There were seven types of applications listed, namely expert system, fault diagnosis, planning system, blackboard architecture, natural language processing, programming tool, and others. Respondents to the survey indicated a mixture of the types for their applications. Details are shown in Figure 3-9. Expert systems are the most common type of applications.

Figure 3-9 AI-Ada Product Availability

3.4 AIWG 1992 Workshop

Figure 3-10 highlights the thousands of lines of source code (KSLOC) and knowledge-based system (KBS) rules in the AI with Ada systems described by the AIWG 1992 workshop participants [Johns 92b]. The Data Fusion Technology Demonstrator System (DFTDS) by the Defence Research Agency (DRA) in the UK, Boeing's Ada Real Time Inference Engine (ARTIE) Automated Sensor Manager (SM), ARTIE Search Area Planner (SP), ARTIE Tactical Cockpit Mission Manager (MM), and Training Control and Evaluation (TC&E) by Bolt, Baranek, and Newman (BBN) are a sampling of applications that are larger than most of those documented in the 1991 AIWG applications survey.

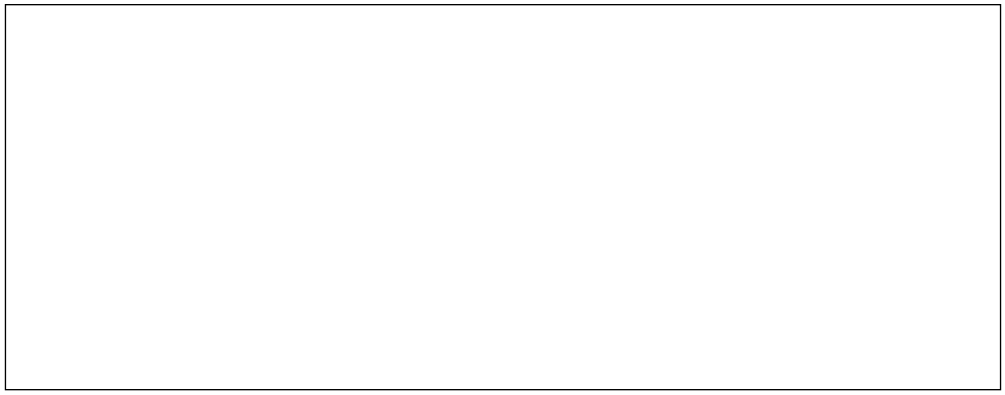

Figure 3-10 Representative AI Applications Written in Ada

3.4.1 The Data Fusion Technology Demonstrator System (DFTDS) Project

The United Kingdom's DRA has implemented a large-scale real-time KBS for shipborne command and control. DFTDS has been implemented with 220 KSLOC, of which 50 KSLOC implements KBSs for time-critical functions such as data fusion and situation assessment. DFTDS is a true KBS in Ada application as the rules are coded directly in Ada to achieve the runtime efficiency required by command and control applications, which must process data from radar, sonar, navigation, electronic support measures (ESM), and other sensors.

The authors find that "Ada as a language has been found to be quite simple to use for a large real-time KBS and in some respects, notably runtime efficiency, abstraction, exception handling, strict typing, has distinct advantages over AI toolkits and expert system shells for engineered real-time applications." Prototyping to assess the potential of implementing time-critical functions such as those required for data fusion, situation assessment, planning, and reaction was also carried out.

- **AI Topics:** blackboard architecture, expert system, data fusion, situation analysis, planning
- **Domain Area:** command and control
- **Language Interfaces:** unknown
- **Project Status:** undergoing sea trials and evaluation

- **Size of Ada Source Code:** 220 KSLOC
- **Number of Rules in Knowledge-Based System:** 510
- **Design/Development Methodology:** iterative prototyping
- **Hardware Platforms:** MicroVAX 3800

The DRA has a three-year program to study the validation and verification (V&V) of safety-critical KBSs with the use of domain-dependent virtual machines. In the first year, a data fusion language (DFL) was identified and formally defined and the requirements for a virtual machine to support the DFL were established. A virtual machine to support the DFL was designed and the design verified in the second year. A virtual machine prototype to support the DFL will be implemented in the third year. At the present time (year two), the virtual machine design is being verified. The approach is being tested with a subset of the Ada KBS that is currently installed on the Royal Naval Frigate HMS Marlborough and described in the associated article "The Data Fusion Technology Demonstrator System (DFTDS) Project" [Johns 92b].

3.4.2 Embedded Real-Time Reasoning Concepts and Applications

Boeing has successfully implemented several intelligent real-time embedded avionics systems with the Ada programming language. These systems include a Tactical Cockpit Mission Manager (40 KSLOC of Ada, 250 rules), a Search Area Planner Tool (45 KSLOC of Ada, 300 rules), and an Automated Sensor Manager (153 KSLOC of Ada, 1181 rules). Notice that these applications are operational AI with Ada applications totaling 238 KSLOC that implement 1,731 rules of complex reasoning-based avionics systems [Johns 92b].

Boeing developed these applications with an innovative approach that combines the "engineering" rigor of conventional software development with AI rapid prototyping techniques. Ada Real-Time Inference Engine (ARTIE) is a tool that provides an interpretive development environment and a small, fast embedded inference engine for runtime performance. ARTIE offers an innovative approach to rapidly prototyping embedded real-time reasoning software with an interpretative development mode and an automatic code generator for generating embedded real-time code for execution on the target platform. Workshop participants were in mutual agreement that interpretive tools such as ARTIE are a valuable asset for real-time embedded systems prototyping, development, and debugging.

- **AI Topics:** cooperating expert systems, forward chaining, iteratively recursive inferencing
- **Domain Area:** intelligent avionics systems
- **Language Interfaces:** Pascal inference engine, developed in 1987
- **Project Status:** test and evaluation
- **Size of Ada Source Code:** ARTIE is 25 KSLOC with associated tools, and the embedded inference engine is less than 3 KSLOC; avionics reasoning-based applications total 238 KSLOC.
- **Number of Rules in Knowledge-Based System(s):** 1731 rules in 3 Ada applications

- **Design/Development Methodology:** iterative prototyping
- **Hardware Platforms:** Apollo 3000, 4000, and 590; Sun 2/60, 3/60, and 4/60; Digital Equipment MicroVAX and VAX; Silicon Graphics IRIS

3.4.3 Training Control & Evaluation (TC&E)

BBN has successfully used an iterative prototyping approach to perform requirements analysis, design, and full-scale engineering development (FSED) for a 70 KSLOC AI application. This application, Training Control & Evaluation (TC&E), is a valuable example of the success that can be achieved by rapid prototyping AI applications with Ada. One of the most difficult tasks in a rapid prototyping requirements analysis activity is the transition of the legacy prototypes and knowledge to the FSED. The TC&E experience offers some valuable insight into a successful transition of prototypes from the requirements analysis to an FSED for a DoD-STD-2167A project.

- **AI Topics:** blackboard architecture, goal setting, path following, progress tracking, obstacle avoidance, target selection
- **Domain Area:** training, simulation, and modeling
- **Language Interfaces:** C and 4GL
- **Project Status:** fielded
- **Size of Ada Source Code:** 70 KSLOC
- **Design/Development Methodology:** iterative, rapid prototyping
- **Hardware Platforms:** Sun

4 Conclusions

Concomitant with the dynamic growth and increased popularity of knowledge-based systems and the progress seen in hardware technology is the vision that AI components will play a major role in complex, large, distributed, and embedded systems. It is indeed envisioned that AI components will play a major role in complex, large, long-lived military systems [DoD 92]; it is thus desirable to be able to use existing matured AI technology.

This next generation of systems must be well engineered following established software engineering practices. It is well known that most AI software development takes place in research laboratories, and systems seldom pass the prototype stage. This next generation of systems must also be able to work together with more conventional software components performing traditional real-time tasks. Designers of these AI embedded applications face the same sort of problems and challenges as traditional software engineers. These systems are best engineered using a language like Ada, since the language addresses specific aspects in the construction of large embedded systems.

There is no question that Ada features more than meet the basic AI requirements expressed in this report. The language has clear and modern control and data structuring mechanisms and powerful abstraction facilities, with comprehensive support for modularity. Furthermore, it has been shown that runtime performance is much better in Ada than in current interpreted typical AI languages. A basic roadblock is that traditional software technology lacks the powerful development environments in use for AI software practice. The few “paradigmatic” inconveniences are secondary to the language’s final success and are related to using a new language to tackle old problems. These problems are in fact much less severe in Ada than when using more traditional languages such as C or Pascal. A few restrictions are more intrinsically related to the current language design (Ada 83); fortunately, several of the proposed changes for the next language revision, collectively known as the Ada9X Project, solve these shortcomings and provide a more complete solution.

A strategy that may prove to be feasible is to develop implementation guidelines for the development of a library of basic AI components, *written in Ada* and using current standard software components [e.g., Motif, Graphics Kernel System (GKS), etc.], to make them available, integratable, and reusable in more conventional software systems. This not only satisfies the need above, but it also paves the way for future automatic support for reengineering (and reverse engineering) of existing AI software. The latter are of paramount importance in the light of the current thrust on megaprogramming and software reuse.

Typical components will include things like frames together with rule-based systems and constraints, to build blackboard architectures and planning systems.

The use of Ada is justified by its sound support for software engineering practice, its recently confirmed strong support for DoD-wide needs, and the access it allows to a relatively large tool base [e.g., the Software Technology for Adaptable, Reliable Systems (STARS) software re-

pository, trained personnel, and a number of standard bindings such as Portable Operating System Interface Standard (POSIX), X-Windows, Structured Query Language (SQL), GKS, etc.].

Advantages of Ada for embedded AI:

- It is a natural choice for embedded AI applications.
- It promotes software engineering even within a prototyping approach (for example, the designer can develop an architecture that encompasses legacy subsystems and new subsystems; it allows subsystems to grow and evolve in a typical evolutionary AI fashion, etc.).
- It is a standardized language (the designer can design well-behaved and well-defined functions and interfaces; it promotes portability—the designer can design for multiple target platforms, etc.).
- Its compilers have matured to the point that Ada is as efficient or more efficient than assembly languages on target embedded processors.

Can AI technology and tools scale up to the challenge of engineering issues such as integration, verification and validation, real-time performance, and life-cycle maintenance?

We have investigated problem-solving architectures for real-time AI systems and identified several remaining engineering problems. *Developing "intelligent" systems by using typical AI technology is becoming a critical bottleneck* in applying knowledge-based techniques in embedded real-time computing. This technology generally lacks facilities for integration with traditional computer software and systems and for dealing with embedded real-time settings. Furthermore, it does not seem capable of scaling up to large software systems, primarily because of the lack of standardization acceptance in the community.

All in all, AI technology generally lacks facilities for the insertion of knowledge-based components into embedded real-time applications. Traditional AI approaches have not considered the ramifications of using exponential time AI algorithms in real-time systems, nor have they addressed reliability and fault-tolerance concerns. In addition, current intelligent control systems would almost always be in a state of computational overload, and thus *cannot in general perform all potential operations in a timely fashion,* thus precluding the use of traditional static scheduling strategies typical of today's real-time technology.

4.1 Ada83 Limitations

In this section we present a summary of the remaining shortcomings of Ada83. This is important since there will be people writing AI software until Ada9X, which is tackling most of these deficiencies, is available.

Of all the AI-unique programming requirements, only two are partially met by Ada83. These are

- Object-oriented programming (full inheritance)
- "Self-modifying" code (adaptive programming).

Self-modifying code is mathematically undecided and hence better avoided for production systems. It can always be achieved by interpreted languages.

4.2 Ada9x Solutions

From the list in the previous section, only object-oriented inheritance remained an impediment for a complete Ada solution to AI problems. This requirements is more than satisfied by the object-oriented features of Ada9X.

In addition, Ada9X provides the following enhancements:

- List manipulation and semantic nets; Ada9X supports “heterogeneous” linked structures (access T’CLASS).
- Frames; Ada9X provides better solutions with class-wide programming and dynamic polymorphism (tagged types).
- Memory management; enhanced in Ada9X with hierarchical access collection pools for type classes and automatic initialization/finalization (protected types).
- Pattern-directed computation and procedural attachment; greatly enhanced in Ada9X by access-to-subprograms types and more flexible strings and arrays.
- Object-oriented programming; full support for inheritance achieved by derivation and type extension. Multiple inheritance can be done in a number of ways, but the best way is to use access discriminants. Abstract types are intrinsic in Ada9X.
- Blackboard architectures; better supported in Ada9X by the use of protected records, asynchronous transfer of control, procedural attachments, heterogeneous data structures, and general object-oriented programming.

The strong typing philosophy of Ada coupled with runtime polymorphism and type classes provides a very robust set of tools for programming AI applications.

Because of the evolutionary nature of AI applications, software engineering is a challenge for the AI community. Software engineering is one of the strengths and advantages offered by a properly managed Ada environment.

Many of the issues and problems faced by the AI with Ada community are the same problems faced by all AI researchers and developers; however these are worse for AI because of the exponential nature of the typical algorithms. We should work closely with the AI community to concentrate our combined efforts on solving our common problems rather than focusing on the perceived differences between the AI and Ada communities.

4.3 Further Work

There are several objectives of our future effort; some of them are

- To complete a survey of maturing AI technology.
- To generalize an AI framework or architecture supporting the integration of AI components with more traditional embedded systems (with reuse in mind).

- To propose the standardization of organized implementation guidelines and language bindings for basic AI modules supporting the architecture.
- To implement a substantial application in Ada9X.
- To define a standard language interface between Ada9X and CommonLISP.

The proposed integration framework is primarily driven by the discovery of standard development models of information agents, standard integrated object-oriented databases, “correspondence points” between Ada and typical AI languages, and component composition technology necessary to enable AI and software engineering components to interoperate smoothly (this include things like intelligent agent architectures and generic architectures for KB systems).

The implementation guidelines and language bindings for the identified framework would include AI planning modules, AI basic-fundamental modules (such as frame structures, lists, pattern matching and functional programming facilities, semantic nets, etc.), KB modules (such as inference engines and BBs), automatic garbage collection Ada issues, generic packages implementing abstract data types, etc.

The product could clearly form part of a handbook for practitioners; it also provides educational technology stimulating better software engineering practice, supported by Ada, for AI.

We propose to select representative AI methods/algorithms to study their performance requirements and to propose a model of “adaptive” real-time behavior as a set of requirements to study the suitability of rate monotonic analyses.

Close collaboration between people with backgrounds in AI, software engineering, real-time programming, and embedded domain software (e.g., operational flight) is essential for the successful completion of these projects.

Integration of AI with conventional software remains difficult. There are some “difficult” things to do, such as multiple inheritance and mutually dependent classes, and some “impossible” things to do such as treating programs as data.

Conceptual problems are best exemplified by the inability of AI approaches to respond rapidly and predictably to fast-changing data for controlling complex systems. Current AI computational models are not based on a good model of behavior, from a resource utilization point of view, for a number of knowledge-based techniques such as searching, rule-based reasoning, semantic nets, etc.

Traditional AI approaches have not considered the ramifications of using exponential time AI algorithms in real-time systems, nor have they addressed reliability and fault-tolerance concerns. Inheritance, especially multiple inheritance, is not good for real time because of its exponential nature and runtime support requirements. An intrinsic concern is the fact that AI algorithms’ performance degrades exponentially and worst-case analyses would probably result in totally unacceptable response times. Multiple inheritance and garbage collection further exacerbate the problem.

The RETE algorithm, an efficient, widely used technique to determine which member(s) of an inference engine's rule base will execute in response to a dynamic set of data (see CLIPS/Ada), has recently come under scrutiny with regard to deterministic real-time performance.

During the last decade, the emergence of AI techniques in control has become evident. Traditionally, however, these knowledge-based problem-solving techniques have been applied in domains in which the data is static and no time-critical responses are required. An assessment of the experiences of using Ada to develop AI applications shows that many of the problems and challenges facing the Ada developer are equally applicable to all AI development environments.

A complete solution for all these problems is not found in commercial software. Only a few shells offer 'real-time-like' capabilities. Reasoning methods are sensitive to quality (utility) and efficiency (resource utilization) tradeoffs.

All in all, AI technology generally lacks facilities for the insertion of knowledge-based components into embedded real-time applications. The depth of these problems must also be studied carefully.

A promising approach is to find "plausible" solutions to the following (contrasting) requirements:

- Non-monotonic: incoming data does not remain constant and later facts or conclusions can invalidate earlier conclusions.
- Temporal: temporal relationships (past, present and future) between events and activities (e.g., in the context of planning).
- Predictable: uncertainty in both data and the deductive engine.
- Asynchronous events: relationship between physical objects to the space they inhabit.
- Interfacing external software: conventional software components performing traditional real-time tasks such as sensor reading and data fusion, control and scheduling functions, and actuator feedback processing.

Acknowledgments

I would like to thank Janet Johns from MITRE Corporation and Jay Strosnider from the Department of Electrical and Computer Engineering, Carnegie Mellon University, for many useful comments on earlier drafts of this report. This work would have not been possible without the encouragement and active support of Dan Roy at the Software Engineering Institute. I would also like to thank Bill Pollak for editorial assistance.

Bibliography

[Adkins 86] Adkins, M.M. “Flexible Data and Control Structures in Ada,” 9.1 - 9.17. Díaz-Herrera, J.L. & Humberger, H. (eds.), *Proceedings of AIDA ‘86.* Fairfax, VA: George Mason University, Nov. 1986.

[Agresti 86] Agresti, W. (ed.). *New Paradigms for Software Development*, IEEE Tutorial. Washington, DC: IEEE Computer Society Press, 1986.

[Ayel 88] Ayel, J. “A Supervision System in Computer Integrated Manufacturing,” 239-246. *Proceedings of the 3rd International Conference in Artificial Intelligence*, Varna, Bulgaria. New York: North-Holland, 1988.

[Baker 90] Baker, H.G. “The Automatic Translation of LISP Applications into Ada,” 633-639. *Proceedings of the 8th Annual Conference on Ada Technology,* Atlanta, GA. Fort Monmouth, NJ: U.S. Army Communications-Electronics Command, 1990.

[Bernaras 90] Bernaras, A. & Smithers, T. “On the Application of Software Engineering Techniques in Artificial Intelligence Research,” 561-169. *Proceedings of the 3rd International Workshop on Software Engineering and its Applications.* Toulouse, 3-7 Dec., 1990. Edinburgh: University of Edinburgh, Department of Artificial Intelligence.

[Bhugra 85] Bhugra, P., et al. *Comparisons Between Ada and LISP* (RSD-TR-9-85). Ann Arbor, MI: University of Michigan, College of Engineering, Robot Systems Division, 1985.

[Bobbie 87] Bobbie, P.O. “Ada-PROLOG: An Ada System for Parallel Interpretation of PROLOG Programs,” 102-123. Díaz-Herrera, J.L. & Humberger, H. (eds.), *Proceedings of AIDA ‘87.* Fairfax, VA: George Mason University, Oct. 1987.

[Bobrow 86] Bobrow, D.G. et el. “CommonLoops: Merging Common Lisp and Object-Oriented Programming,” 17-29. *Proceedings of the ACM OPSULA ‘86*. Washington, DC: The Brookings Institute.

[Boehm 88] Boehm, B.W. “A Spiral Model of Software Development and Enhancement.” *IEEE Computer 21*, 5 (May 1988): 61-72.

[Bonasso 90] Bonasso, P. (ed.). *Practical Artificial Intelligence: Techniques and Applications in Government Systems.* Bedford, MA: The MITRE Corp, 1990.

[Booch 86] Booch, G. “Object-Oriented Development.” *IEEE Transactions on Software Engineering SE-12*, 2 (Feb. 1986): 211-221.

[Brachman 83] Brachman, R.J. “What IS-A Is and Isn’t: An Analysis of Taxonomic Links in Semantic Networks.” *IEEE Computer 16*, 10 (October 1983): pp. 30-36.

[Burback 87] Burback, R. “PROVER: A First-Order Logic System in Ada,” 166-190. Díaz-Herrera, J.L. & Humberger, H. (eds.), *Proceedings of AIDA ‘87.* Fairfax, VA: George Mason University, Oct. 1987.

[Burns 81] Burns, A. & Davies, G. *Pascal_FC: A Language for Teaching Concurrent Programming*. Bradford, UK: Schools of Studies in Computing, University of Bradford, UK, 1981.

[Charette 86] Charette, R.N. *Software Engineering Environments: Concepts and Technology.* New York: McGraw-Hill, 1986.

[Charniak 87] Charniak, E. & McDemott, D. *Introduction to Artificial Intelligence.* Reading, MA: Addison-Wesley, 1987

[Collard 88] Collard P., et al. “Knowledge-Based Systems and Ada.” *Ada Letters 8*, 6 (Nov./Dec. 1988): 72-81.

[Corel 85] Corel Software Corporation. *Object-Logo 1.5.* Cambridge, MA: 1985.

[Corkill 88] Corkill, D. & Gallagher, K. “Tuning a Blackboard-Based Application: A Case Study Using GBB.” *Proceedings of AAAI ‘88*, vol. 1, 671-676. Cambridge, MA: Department of Computer and Information Science, University of Massachusetts, 1988.

[Corkill 90] Corkill, D. “Blackboard Architectures and Control Applications,” 36-38. *Proceedings of the IEEE International Symposium on Intelligent Control*, Philadelphia, PA, 5-7 Sept., 1990.

[Dahl 69] Dahl, O-J; Nygaard, K.; & Myhrhaug, B. *The SIMULA 67 Common Base Language* (Pub. S-22). Oslo, Norway: Norwegian Computing Center, 1969.

[Dahl 72] Dahl, O-J; Dijkstra, E.W.; & Hoare, C.A.R. *Structured Programming.* New York: Academic Press, 1972.

[De Feyter 88] DeFeyter, A.R. “RTEX: An Industrial Real-Time Expert System Shell,” 6.1 - 6.22. Díaz-Herrera, J.L. & Humberger, H. (eds.), *Proceedings of AIDA ‘88.* Fairfax, VA: George Mason University, Nov. 1988.

[Diaz 90] Diaz, A.C., et al. “A Prototype Blackboard Shell Using CLIPS.” *Applications of Artificial Intelligence in Engineering V*, vol. 2 (1990): 67-77.

[Díaz-Herrera 86] Díaz-Herrera, J.L.; Gonzalez, C; & Wang, P. “The Development of a Flight Control System in Ada,” 95-102. *Proceedings of WADAS ‘86*. Washington, DC: March 1986.

[Díaz-Herrera 87] Díaz-Herrera, J.L. “The Ada-AI Interface,” *67-72. Proceedings of the National Computer Conference*. Chicago, IL: June 1987.

[Díaz-Herrera 92a] Díaz-Herrera, J. L. & Stewart, C. *Experiences with the Evolutionary Spiral Software Development Process.* Fairfax, VA: Center of Excellence in C^3I, George Mason University, Jan. 1992. (Special project report.)

[Díaz-Herrera 92b] Díaz-Herrera, J.L., et al. *Intelligent Message Tracking: A Feasibility Study and Risk Analysis of the Use of Artificial Intelligence Technology in Crisis Action Tracking in the Crisis Management ADP System (CMAS)*. Fairfax, VA: Center of Excellence in C^3I, George Mason University, Jan. 1992. (Special project report.)

[Díaz-Herrera 93] Díaz-Herrera, J.L. “Artificial Intelligence and Ada,” pp.1-48. Kent, A. & Williams, J. (eds.). *Encyclopedia of Computer Science and Technology,* vol. 27, ch. 12. New York: Marcel Dekker, Inc., 1993.

[DoD 87] U.S. Department of Defense. *Use of Ada in Weapon Systems* (Directive No. 3405). March 2, 1987.

[DoD 92] U.S. Department of Defense. *Department of Defense Software Technology Strategy* (draft). December 1992.

[Dodhiawala 89] Dodhiawala, R., et al. “Real-Time AI Systems: A Definition and an Architecture,” 256-261. *IJCAI ‘89*, vol. 1. Santa Clara, CA: FMC Corp. Technology Center, 1989.

[EIA 85] Electronic Industries Association. *The DoD Computing Activities and Programs: 1985 Specific Market Survey.* Washington, DC: 1985.

[Factor 89] Factor, M. & Gelernter, D.H. “The Process Trellis: A Software Architecture for Intelligent Monitor,” 174-181. *IEEE International Workshop on Tools for Artificial Intelligence*, Fairfax, VA, 1989.

[Fonash 83] Fonash, P. “Ada—Program Overview.” *Signal 37*, 11 (July 1983): 27-31.

[Fornarino 89] Fornarino, C. & Neveu, B. “Ada and LeLisp: A Marriage of Convenience for AI.” Díaz-Herrera, J.L. & Zytkow, J. (eds.), *Proceedings of AIDA ‘89.* Fairfax, VA: George Mason University, Nov. 1989.

[Gilstrap 91] Gilstrap, L. *Expert System Development Methodology Reference Manual* (DSTL-90-006). Greenbelt, Maryland 20771: NASA/Goddard Space Flight Center, 1991.

[Goldberg 76] Goldberg, A. & Kay, A. (eds.). *Smalltalk-72 Instructional Manual* (PARC technical report). Palo Alto, CA: Xerox Corp. March 1976.

[Haihong 90] Haihong, D., et al. “A Framework for Real-Time Processing,” 179-185. *Proceedings of the European Conference on Artificial Intelligence*, Stockholm, Sweden. London, UK: 1990.

[Hayes-Roth 90] Hayes-Roth, B. “Architectural Foundations for Real-Time Performance in Intelligent Agents.” *The Journal of Real-Time Systems 2* (1990): 99-125.

[Hayslip 89] Hayslip, I.C. & Rosenking, J.P. “Adaptive Planning for Threat Response,” 1031-1041. *Proceedings of SPIE 1095*, pt. 2, Applications of Artificial Intelligence VII. Orlando, FL: 28-30 March, 1989.

[Henderson 80] Henderson, P. *Functional Programming: Application and Implementation.* Englewood Cliffs, NJ: Prentice-Hall International, 1980.

[Horvitz 87] Horvitz, E.J. *Reasoning About Beliefs and Actions Under Computational Resource Constraints* (KSL-87-29). Stanford, CA: Stanford University, Department of Computer Science, 1987.

[Ice 87] Ice, S., et al. “Raising ALLAN: Ada Logic-Based Language,” 155-165. Díaz-Herrera, J.L. & Humberger, H. (eds.), *Proceedings of AIDA ‘87.* Fairfax, VA: George Mason University, Oct. 1987.

[Jackson 83] Jackson, M. *System Development*. Englewood Cliffs, NJ: Prentice-Hall International, 1983.

[Johnson 89] Johnson, D., et al. “Real-Time Blackboards for Sensor Fusion,” 61-72. *Proceedings of SPIE 1100*, Sensor Fusion II. Orlando, FL: 28-29 March, 1989.

[Johns 92a] Johns, J.F. *1991 Annual Report for the ACM SIGAda AIWG* (Document M92B0000056). Bedford, MA: MITRE Corp., June 1992.

[Johns 92b] Johns, J.F. 1992 *Summer SIGAda AIWG Workshop* (Document M92B0000103). Bedford, MA: MITRE Corp., September 1992.

[Kara 91] Kara M., et al. *A Blackboard-Based Framework for the Development of Cooperating Schedulers* (Report 91.34). Leeds, UK: University of Leeds, School of Computer Studies, 1991.

[Kilpeläinen 89] Kilpeläinen, P., et al. “Prolog in Ada: An Implementation and an Embedding,” 96-107. Díaz-Herrera, J.L. & Zytkow, J. (eds.), *Proceedings of AIDA ‘89.* Fairfax, VA: George Mason University, Nov. 1989.

[Kovarik 88] Kovarik, V.J. & Nies, S. “Supporting Object-Oriented Programming with Ada: Extending the Paradigm,” 131-136. Díaz-Herrera, J.L. & Humberger, H. (eds.), *Proceedings of AIDA ‘88.* Fairfax, VA: George Mason University, Nov. 1988.

[Kowalski 79] Kowalski, R. *Logic for Problem Solving.* New York: American Elsevier, 1979.

[Krijgsman 90] Krijgsman, A.J.; Verbruggen, H.B.; Brujin, P.M.; & Holweg, E.G.M. “DICE: A Real-Time Intelligent Control Environment,” 61-65. *Proceedings of the 1990 European Simulation Symposium.* Ghent, Belgium: 8-10 Nov. 1990.

[Lander 86] Lander, L., et al. “The Use of Ada in Expert Systems,” 7.1 - 7.12. Díaz-Herrera, J.L. & Humberger, H. (eds.), *Proceedings of AIDA ‘86.* Fairfax, VA: George Mason University, Nov. 1986.

[Larner 90] Larner, D.L. “A Distributed, Operating System Based, Blackboard Architecture for Real-Time Control,” 99-108. *Proceedings of the Third International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems,* vol. 1. Charleston, SC: 1990.

[Lee 90] Lee, S.D. “Toward the Efficient Implementation of Expert Systems in Ada,” 571-580. *Proceedings of the TRI-Ada Conference.* New York, NY: ACM/SIGAda, December 1990.

[Lifschitz 91] Lifschitz, V. (ed.). *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John MacCarthy.* Boston: Academic Press, 1991.

[Liskov 74] Liskov, B. & Zilles, S. “Programming with Abstract Data Types.” *SIGPLAN Notices 9*, 4 (1974): 50-59.

[MacLean 88] MacLean, J., et al. “BESTool: A Blackboard Shell and its Application to a Traffic Control Problem,” 389-393. *IEEE International Symposium on Intelligent Control.* Arlington, VA: 1988.

[Marshall 90] Marshall, C.B. “The Application of Real-Time AI Techniques to Pulse Train De-interleaving,” 350-355. *Proceedings of the Military Microwave Conference*, London, UK. Tunbridge Wells, UK: 1990.

[Martin 89] Martin, J.L. “A Development Tool for Real-Time Expert Systems.” *Alsynews 3*, 1 (Waltham, MA, Alsys, March 1989): 13-16.

[Meeson 84] Meeson, R.N. “Function-Level Programming in Ada,” 128-132. *Conference on Ada Applications and Environments.* Silver Spring, MD: IEEE Computer Society Press, 1984.

[Meyer 87] Meyer, B. “EIFFEL: Programming for Reusability and Extendibility” *SIGPLAN Notices 22*, 2 (Feb 1987): 85-94.

[Michalski 86] Michalski, R. & Winston, P. “Variable Precision Logic.” *Artificial Intelligence 29*, 2 (North-Holland, 1986): 121-146.

[MIL-STD-1638A] *Common Ada Programming Support Environment (APSE) Interface Set* (CAIS-A). Washington, DC: U.S. Department of Defense, April 1990.

[MIL-STD-1815A] *The Reference Manual for the Ada Programming Language* (ANSI/MIL-STD-1815A). Washington, DC: US Department of Defense, February 1983.

[MIL-STD-2167A] *Software Development.* Washington, DC: U.S. Department of Defense, February 1983.

[Miles 89] Miles, J.; Daniel, J.W.; & Mulvaney, D.J. “Real-Time Performance Comparison of a Knowledge-Based Data Fusion System Using MUSE, ART and Ada,” 247-259. *International Workshop on Expert Systems and Their Applications*. Avignon, France: Nanterre, Hauts-de-Seine, EC2, June 1989.

[Miles 92] Miles, J. “The Data Fusion Technology Demonstrator System (DFTDS) Project,” B.1-B.10. Johns, J.F. (ed.), *AIWG 1992 Summer Workshop* (M 93B0000072). Bedford, MA: MITRE Corp., 1993.

[Mills 86] Mills, H.D. “Structured Programming: Retrospect and Prospect.” *IEEE Software 3*, 6 (Nov. 1986): 58-66.

[Minsky 74] Minsky, M. *A Framework for Representing Knowledge* (AI Memo 306). MIT AI Laboratory, 1974.

[Mizuno 83] Mizuno, Y. “Software Quality Improvement.” *IEEE Computer 15,* 3 (March 1983): 66-72.

[Moon 86] Moon, D. “Object-Oriented Programming with Flavors.” *SIGPLAN Notices 21,* 11 (November 1986): 1-8.

[NASA 89] NASA Johnson Space Center. Artificial Intelligence section, *CLIPS Version 4.3 Reference Manual*. 1989.

[Newell 72] Newell, A. & Simon, H.A. *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall, 1972.

[Oberndorf 86] Oberndorf, P.A. *Common Ada Programming Support Environment (APSE) Interface Set (CAIS)*. San Diego, CA: U.S. Department of Defense, U.S. Naval Systems Center, October 9, 1986.

[Pang 89] Pang, G.K.H. “An Architecture for Expert System Based Feedback Control.” *Automatica 25,* 6 (November 1989): 813-827.

[Park 93] Park, Y-T. *Blackboard Scheduler Control Knowledge for Heuristic Classification: Representation and Inference* (Report No. UIUCDCS-R-93-1788). Department of Computer Science, University of Illinois at Urbana-Champaign, January 1993.

[Parks 90] Parks, et al. “A Blackboard/Knowledge-Based Systems Approach to Manufacturing Scheduling and Control.” *Computers & Industrial Engineering 19*, 1-4 (March 1990): 72-76.

[Parnas 72] Parnas, D. “On the Criteria to Be Used in Decomposing Systems into Modules.” *Communications of the ACM 15*, 12 (1972): 1053-1058.

[Petterson 88] Petterson, G. & Strönberg, D. “System Architecture for a Real-Time Planning Agent in an Autonomous Air Craft,” 132-139. Ras & Saitta (eds.), *Methodologies for Intelligent Systems,* vol. 3. New York, NY: Elsevier Science Pub. Co., 1988.

[Probert 82] Probert, T.H. *Ada Validation Organization: Policies and Procedures* (MTR-82W00103). McLean, VA: MITRE Corp., June, 1982.

[Quillian 68] Quillian, M.R. “Semantic Memory,” 216-270. Minsky, M. (ed.), *Semantic Information Processing.* Cambridge, MA: MIT Press, 1968.

[Ramamorthy 87] Ramamorthy, C.V.; Shekhar, S.; & Garg, V. “Software Development Support for AI Programs.” *IEEE Computer 20*, 1 (Jan. 1987): 30-40.

[Reeker 87] Reeker, L.H. & Wauchope, K. “Pattern-Directed Processing in Ada,” 49-56. *Second International Conference on Ada Applications and Environment*, Miami FL, April 1987.

[Ruoff 91] Ruoff, K. “Integration of Artificial Intelligence Technologies to Support Command, Control, Communication and Intelligence Systems” (AIAA-91-3708-CP), 36-39. *8th AIAA Computing in Aerospace Conference.* Washington, D.C: AIAA, 1991.

[Sellers 91] Sellers, S., et al. “The Migration of Artificial Intelligence Onboard Spacecraft” (AIAA-91-3711-CP), 54-61. *8th AIAA Computing in Aerospace Conference.* Washington, D.C: AIAA, 1991.

[Schaffert 86] Schaffert C., et al. “An Introduction to Trellis/Owl.” *SIGPLAN Notices 21*, 11 (1986): 9-16.

[Scheidt 86] Scheidt, D.; Preston, D.; & Armstrong, M. “Implementing Semantic Networks in Ada,” 8.1-8.6. Díaz-Herrera, J.L. & Humberger, H. (eds.), *Proceedings of AIDA ‘86.* Fairfax, VA: George Mason University, Nov. 1986.

[Shriver 87] Shriver, B. & Wegner, P. (eds.). *Research Directions in Object-Oriented Programming*. Cambridge, MA: The MIT Press, 1987.

[Sorrells 85] Sorrells, M.E. “A Time-Constrained Inference Strategy for Real-Time Expert Systems,” 1336-1341. *IEEE Proceedings of the National Aerospace and Electronics Conference*. New York, NY: IEEE, 1985.

[Stankovic 88] Stankovic, J.A. “Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems.” *IEEE Computer 21*, 10 (Oct. 1988): 10-19.

[Stankovic 93] Stankovic, J.A. *Reflective Real-Time Systems.* Amherst, MA: Department of Computer Science, University of Massachusetts, June 28, 1993.

[Stockman 88] Stockman, S. “ABLE: An Ada-Based Blackboard System,” 18.1-18.9. Díaz-Herrera, J.L. & Humberger, H. (eds.), *Proceedings of AIDA ‘88.* Fairfax, VA: George Mason University, Nov. 1988.

[Stroustrup 86] Stroustrup, B. *The C++ Programming Language.* Reading, MA.: Addison-Wesley 1986.

[Tanimoto 90] Tanimoto, S. *The Elements of Artificial Intelligence.* Rockville, MD: Computer Science Press, 1990.

[Tesler 85] Tesler, L. *Object Pascal Report*. Cupertino, CA: Apple Computer, 1985.

[Thornbrugh 87] Thornbrugh, A.L. “Organizing Multiple Expert Systems: A Blackboard-Based Executive Application,” 145-155. *Knowledge-Based Systems for Engineering: Classification, Education, and Control.* Southampton, UK: Computational Mechanics Publication, 1987.

[Vepa 91] Vepa, R. “The Architecture of Knowledge Bases Required for Implementing Expert Control Systems,” Number 091. *IEEE Colloquium on Knowledge-Based Control,* London, UK, Sept. 1-4, 1991.

[Walters 87] Walters, M.D. & Martz, S.M. *Frame-Based Knowledge Representation in Ada.* Wichita, KS: Boeing Military Airplanes, March 1987.

[Warnier 76] Warnier, Jean-Dominique. *Logical Construction of Programs.* New York, NY: Von Nostrand Reinhold Co., 1976.

[Welsh 87] Welsh, J., et al. *The Ada Language and Methodology.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

[Winston 84] Winston P.H. *Artificial Intelligence,* 2nd edition, 129-131. Reading, MA: Addison-Wesley, 1984.

[Woods 91a] Woods, D. “Advanced Automation Integration” (AIAA-91-3707-CP). *8th AIAA Computing in Aerospace Conference.* Washington, D.C: AIAA, 1991.

[Woods 91b] Woods, D. “Space Station Freedom: Artificial Intelligence, Engineering & Integration” (AIAA-91-3852-CP), 217-225. *8th AIAA Computing in Aerospace Conference.* Washington, D.C: AIAA, 1991.

[Wright 84] Wright, J.M.; Fox, M.S.; & Adam, D. *SRL/1.5 User's Manual.* Pittsburgh, PA: Carnegie Mellon University, Robotics Institute, 1984.

[Wright 86] Wright, M., et al. “An Expert System for Real-Time Control.” *IEEE Software 3*, 2 (March 1986): 16-24.

[Wright 89] Wright, P.A. “Ada Real-time Inference Engine - ARTIE,” 83-93. Díaz-Herrera, J.L. & Zytkow, J. (eds.), *Proceedings of AIDA ‘89.* Fairfax, VA: George Mason University, Nov. 1989.

[Yen 90] Yen, M. “Using a Dynamic Memory Management Package to Facilitate Building Lisp-like Data Structures in Ada,” 85-100. Baldo, J.; Díaz-Herrera, J.L.; & Littman, D. (eds.). *Proceedings of AIDA ‘90.* Fairfax, VA: George Mason University, Nov. 1990.

[Zave 89] Zave, P. “A Compositional Approach to Multi-Paradigm Programming.” *IEEE Software 6,* 5 (Sept. 1989): 15-24.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-94-TR-09

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESC-TR-94-009

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (city, state, and zip code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (city, state, and zip code)
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESC/ENS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962890C0003

8c. ADDRESS (city, state, and zip code))
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63756E	N/A	N/A	N/A

11. TITLE (Include Security Classification)
Artificial Intelligence (AI) and Ada: Integrating AI with Mainstream Software Engineering

12. PERSONAL AUTHOR(S)
Jorge L. Diaz-Herrera

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (year, month, day)	15. PAGE COUNT
Final	FROM TO	September 1994	55 pp.

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
Ada
artificial intelligence
distributed agents
artificial intelligence development
embedded artificial intelligence
software engineering for artificial intelligence

19. ABSTRACT (continue on reverse if necessary and identify by block number)

In this report we discuss in detail pragmatic problems posed by the integration of AI with conventional software engineering, and within the framework of current Ada technology. A major objective of this work has been to begin to bridge the gap between the Ada and AI software cultures. The report summarizes survey results from the Association for Computing Machinery (ACM) Special Interest Group for Ada (SIGAda) AI Working Group (AIWG), highlighting lessons learned and sample applications. An interesting observation is that a large percentage of AI code is procedural by nature and that better productivity rates are achieved by using Ada; also, efficiency is much better when com-

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
Thomas R. Miller, Lt Col, USAF

22b. TELEPHONE NUMBER (include area code)
(412) 268-7631

22c. OFFICE SYMBOL
ESC/ENS (SEI)

pared to traditional AI languages. Although we show favorable results on the use of Ada technology for the implementation of AI software, a total integration remains difficult at the conceptual level. There are some impediments to a completely satisfactory solution, but only a few restrictions are more intrinsically related to the current Ada standard (Ada83); these, however, are being dealt with in the next revision known as Ada9X.