IN-61-CR
46434
p. 44

ART-Ada Design Project - Phase II
Final Report

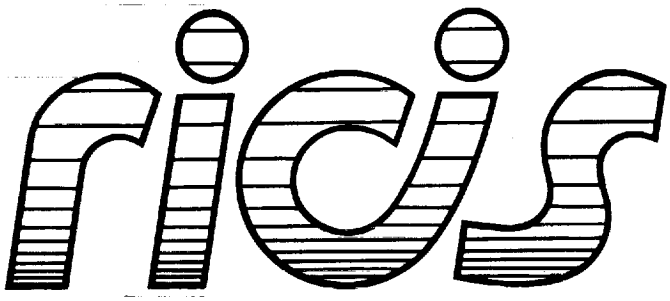
S. Danial Lee
Bradley P. Allen

Inference Corporation

February 1990

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

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division



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

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

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

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

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

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

ART/Ada Design Project - Phase II
Final Report

Preface

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

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

ART-Ada Design Project - Phase II

Final Report

Subcontract 015
RICIS Research Activity SE.19
NASA Cooperative Agreement NCC-9-16

S. Daniel Lee
Bradley P. Allen

Inference Corporation

February 1990

Table of Contents

List of Figures

List of Tables

Abstract

Interest in deploying expert systems in Ada has increased. This report describes an Ada-based expert system tool called ART-Ada, which was built to support research into the language and methodological issues of expert systems in Ada. ART-Ada allows applications of an existing expert system tool called ART-IM (Automated Reasoning Tool for Information Management) to be deployed in various Ada environments. ART-IM, a C-based expert system tool, is used to generate Ada source code which is compiled and linked with an Ada-based inference engine to produce an Ada executable image. The future research directions call for improved support for real-time embedded and distributed expert systems. ART-Ada is being used to implement several expert systems for NASA's Space Station Freedom Program and the U.S. Air Force.

1. Introduction

1.1 Motivation

The Department of Defense mandate to standardize on Ada as the language for software systems development has resulted in increased interest from developers of large-scale Ada systems in making expert systems technology readily available in Ada environments. Two examples of Ada applications that can benefit from the use of expert systems are monitoring and control systems and decision support systems. Monitoring and control systems demand real-time performance, small execution images, tight integration with other applications, and predictable demands on processor resources; decision support systems have somewhat less stringent requirements.

An example project that exhibits the need for both of these types of systems is NASA's Space Station Freedom. Monitoring and control systems that will perform fault detection, isolation and reconfiguration for various on-board systems are expected to be developed and deployed on the station either in its initial operating configuration or as the station evolves; decision support systems that will provide assistance in activities such as crew-time scheduling and failure mode analysis are also under consideration. These systems will be expected to run reliably on a standard data processor, currently envisioned to be an 80386-based workstation. The Station is typical of the large Ada software development projects that will require expert systems in the 1990's.

Another large-scale application that can benefit from Ada-based expert system tool technology is the Pilot's Associate (PA) expert system project for military combat aircraft [13]. Funded by the Defense Advanced Research Projects Agency (DARPA) as part of its Strategic Computing Program, the PA project attempts to automate the cockpit of military combat aircraft using Artificial Intelligence (AI) techniques. A Lisp-based expert system tool, ART (Automated Reasoning Tool), was used to implement one of the two prototypes built during Phase I. An Ada-based expert system tool can provide a migration path to deploy the prototype on an on-board computer because Ada cross-compilers are readily available to run Ada programs on most embedded processors used for avionics.

1.2 Project Background and Status

Inference has been involved with Ada-based expert systems research since 1986. Initial work centered around a specification for an Ada-based expert system tool. The result of this research activity is summarized in [15]. In 1988, the ART-Ada Design Project was initiated to design and implement an Ada-based expert system tool. At the end of Phase I of this project, a working prototype was successfully demonstrated. This research activity is reported in [17] and [22]. In 1989, during the ART-Ada Design Project - Phase II, the Phase I prototype was extended and refined so that it could be

released to beta sites. At the end of 1989, ART-Ada was released to beta sites as ART-Ada 2.0 Beta on the VAX/VMS and Sun/Unix platforms [18]. In 1990, eight beta sites, four NASA sites and four Air Force sites, will be evaluating ART-Ada 2.0 for eight months by developing expert systems and deploying them in Ada environments.

The objectives of the ART-Ada Design Project were two fold:

1. to determine the feasibility of providing a hybrid expert system tool such as ART in Ada, and

2. to develop a strategy for Ada integration and deployment of such a tool.

Both of these objectives were met successfully when ART-Ada 2.0 beta was released to the beta sites. Ada compiler problems and Ada language issues encountered during this project are documented in this report. During the evaluation period, the following objectives will be important:

1. to evaluate any bugs or performance problems, and

2. to determine any issues related to particular embedded system environments.

1.3 Approach

Inference Corporation developed an expert system tool called ART (Automated Reasoning Tool) that has been commercially available for several years [16]. ART is written in Common Lisp and it supports various reasoning facilities such as rules, objects, truth maintenance, hypothetical reasoning and object-oriented programming. In 1988, Inference introduced another expert system tool called ART-IM (Automated Reasoning Tool for Information Management), which is also commercially available [19]. ART-IM is written in C and it supports a major subset of ART's reasoning facilities including rules, objects, truth maintenance and object-oriented programming. ART-IM consists of

- a runtime kernel,

- a C deployment compiler, and

- an interactive development environment.

ART-IM's kernel supports the following features:

- a forward-chaining production rule system based on the Rete algorithm [9],

- an object system,

- object-oriented programming,
- a justification-based truth maintenance system (JTMS), and
- explanation generation utilities.

ART-IM supports deployment of applications in C using a C deployment compiler that converts an application into C data structure definitions in the form of either C source code or object code. ART-IM's interactive development environment includes a graphical user interface that allows browsing and debugging of the knowledge base and an integrated editor that offers incremental compilation. ART-IM is available for MVS, VMS, Unix, MS-DOS, and OS/2 environments.

Our approach in designing an Ada-based expert system tool was to use the architecture of proven expert system tools: ART and ART-IM. Both ART and ART-IM have been successfully used to develop many applications which are in daily use today [7], [23], [24]. ART-IM was selected as a baseline system because C is much closer to Ada. While ART-IM's inference engine was reimplemented in Ada, ART-IM's front-end (its parser/analyzer and graphical user interface) was reused as the ART-Ada development environment. The ART-IM kernel was enhanced to generate Ada source code that would be used to initialize Ada data structures equivalent to ART-IM's internal C data structures, and also to interface with user-written Ada code. This approach allows the user to take full advantage of the interactive development environment developed originally for ART-IM. Once the development is complete, the application is automatically converted to Ada source code. It is, then, compiled and linked with the Ada runtime kernel, which is an Ada-based inference engine.

2. ART-Ada: An Ada-based Expert System Tool

2.1 Overall Architecture

ART-Ada is designed to be used by knowledge engineers who may not be familiar with Ada. With minimum knowledge about Ada, they can still develop a knowledge base in a high-level language whose syntax most resembles that of Common Lisp. When the knowledge base is completed, Ada source code can be generated automatically by simply "pressing a button".

When this automatically generated Ada code is compiled and linked with the Ada library of the ART-Ada runtime kernel, an Ada executable image is produced. ART-Ada also provides extensive capabilities for Ada integration so that the knowledge base can be embedded in an Ada environment. It would be best if the knowledge engineer developing the knowledge base works with an Ada programmer who serves as a system integrator. ART-Ada would be most useful for those who must deploy in Ada environments (because of the Ada mandate) expert system applications already developed using tools that do not support Ada deployment.

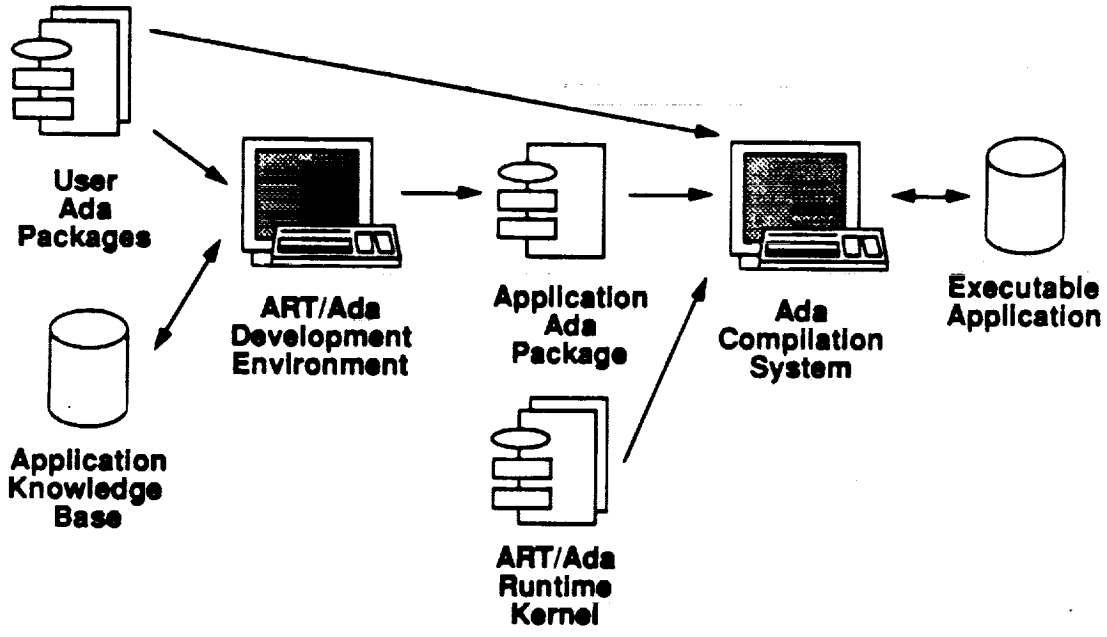


Figure 2-1: Overall Architecture of ART-Ada

The overall architecture of ART-Ada is depicted in figure 2-1. The knowledge base is developed and debugged using an interactive user interface that supports three main features; a command loop similar to the Lisp eval loop, a graphical user interface for knowledge base browsing and debugging, and an integrated editor for incremental compilation of the knowledge base. Any user-written Ada code can be integrated into the knowledge base by either calling it from a rule or invoking it as a method for object-oriented programming.

Once the knowledge base is fully debugged, it can be automatically converted into an Ada package for deployment. The ART-Ada runtime kernel is an Ada library, which is in essence an Ada-based inference engine. An Ada executable image is produced when the machine-generated Ada code and any user-written Ada code, if any, are compiled and linked with the Ada library.

2.2 Knowledge Representation

ART-Ada's key feature is the integration of rule-based representation and object-based (frame-based) representation. It supports three different programming methodologies:

- Rule-based Programming -- Rules opportunistically react to changes in the surrounding database. Rules can fire (execute) in an order based largely on the dynamic ordering of those changes. Rules cannot call other rules, and hence must communicate indirectly by making changes to the database which will, in turn, stimulate other rules.

- Object-Oriented Programming -- The fundamental unit of ART-Ada's object-oriented programming is the *object*, represented by a *schema*. Control is managed by sending *messages* to *objects* (schemas). The object reacts to the message by searching within itself for a *method* appropriate to that message. If an object does not have a method for the received message, it searches to see if it has inherited any appropriate methods from its parents. Once a method has been found, the object carries out the actions associated with the method.

- Procedural Programming -- ART-Ada's procedural language supports function calling, iteration (for, while) and conditionals (if, and, not). There are more than two hundred functions available in the procedural language.

ART-Ada's rule system is based on the optimized Rete pattern-matching algorithm [9]. Unlike OPS5, ART-Ada rules can pattern-match on objects called *schemas* as well as on lists called *facts*. *Facts* are similar to Lisp lists and do not support any inheritance. *Schemas* are similar to CLOS (Common Lisp Object System) objects; they are organized as attribute-value pairs and support *inheritance* through the

is-a (subclass) and *instance-of* (member) relations. In the following example, *mammal* and *dog* are schemas while (animal-found dog) is a fact. *Mammal* is a class and *dog* is a subclass of the class *mammal*; they are linked with an *is-a* link. On the other hand, *fido* is a member of classes *dog* and *mammal*; it is linked to the class *dog* through an *instance-of* link. The significance of the relations *is-a* and *instance-of* is that the attribute-value pairs gets inherited either from a class to a subclass or from a class to a member. In the following example, *fido* will inherit attributes (eats meat), (socialization pack), (locomotion-mechanism run), and (instance-of mammal) from *dog*; it will also inherit (feeds-offspring milk) and (skin-covering hair) from *mammals*. As shown in the rule *determine-if-dog* that matches on both a schema pattern (schema ?animal (...)) and a fact pattern (classify-animal ?animal), the ART-Ada rules can match with schemas as well as facts. In order to optimize performance, ART-Ada uses two separate pattern matchers: one for schemas and one for facts.

```
(defschema mammal
   (feeds-offspring milk)
   (skin-covering hair))

(defschema dog
   (is-a mammal)
   (eats meat)
   (socialization pack)
   (locomotion-mechanism run))

(defschema fido
   (instance-of dog)
   (owned-by John))

(defrule determine-if-dog
   "Determine if subject is a dog."
   (classify-animal ?animal)
   (schema ?animal
      (is-a mammal)
      (socialization pack)
      (eats meat))
   =>
   (assert (schema ?animal
              (is-a dog)))
   (assert (animal-found dog)))
```

When an expert system deduces a conclusion (e.g. to diagnose faults in an electric circuit), it is often required to answer a question like "why?". This capability is called *explanation*. In ART-Ada, an explanation capability can be implemented using the *justification system*. When enabled, the justification system can provide a listing of the rules and data objects which were responsible for creating a particular fact or schema. By embedding features of the justification system in an application, the expert system can trace the steps leading to a particular conclusion. The justification system is also a powerful debugging tool when used during the development of an expert system.

Should an application exhibit unexpected behavior during development, the programmer can exploit the features of the justification system to discover the source of the problem.

In the following example, if (classify-animal my-kangaroo) matches with a LHS pattern (classify-animal ?animal) where ?animal is a variable, and the rule fires to assert (schema my-kangaroo (is-a marsupial)), then we say that (classify-animal my-kangaroo) *justifies* (schema my-kangaroo (is-a marsupial)). In ART-Ada, consistency of the knowledge base is maintained by a justification-based truth maintenance system (JTMS) called Logical Dependencies. If *logical* is wrapped around (classify-animal ?animal), (schema my-kangaroo (is-a marsupial)) is not only *justified by* but also *logically dependent* on (classify-animal my-kangaroo); when (classify-animal my-kangaroo) is retracted from the knowledge base, (schema my-kangaroo (is-a marsupial)) is also retracted, and therefore consistency of the knowledge base is maintained automatically.

```
(defrule determine-if-marsupial
   "Determine if subject is marsupial."
   (logical (classify-animal ?animal))
   (schema ?animal
      (is-a mammal)
      (carries-offspring pouch))
   =>
   (assert (schema ?animal
             (is-a marsupial))))
```

In ART-Ada, object-oriented programming can be used with rule-based programming to take advantage of both paradigms. In the following example, the rule *print-out-object* is used to sent the *print* message to all objects that are instances of *object*. When an object *my-triangle* matches with the rule *print-out-object*, an inherited method *print-triangle* will be invoked. Methods can be defined either in ART-Ada's procedural language using *def-art-fun* which is similar to the Lisp defun, or directly in Ada using *def-user-fun* which will be discussed later.

```
;;; define print methods using def-art-fun

(def-art-fun print-unknown (?schema ?x ?y)
  (printout t t "print unknown " ?schema " " ?x " " ?y))

(def-art-fun print-circle (?schema ?x ?y)
  (printout t t "print circle " ?schema " " ?x " " ?y))

(def-art-fun print-triangle (?schema ?x ?y)
  (printout t t "print triangle " ?schema " " ?x " " ?y))

;;; define objects

(defschema object
  (print print-unknown))

(defschema circle
  (is-a object)
  (print print-circle))

(defschema triangle
  (is-a object)
  (print print-triangle))

(defschema my-triangle
  (instance-of triangle)
  (position (1 2)))

;;; define a rule that sends a print message.

(defrule print-out-object
  (schema ?object
          (instance-of object)
          (position (?x ?y)))
  =>
  (send print ?object ?x ?y))
```

2.3 Knowledge Base Debugging

ART-Ada offers three main features in the user interface called the *Studio*:*

- a command loop,
- a graphical user interface, and
- an integrated editor.

*The Sun version supports only a command loop interface while the VAX/VMS version supports all three.

ART-Ada's command loop is similar to the Lisp eval loop, in which user input is interpreted. More than two hundred functions are available in the command loop. Even Ada functions can be added to the command loop and called from the command loop.

The Studio's interactive, menu-based graphical user interface provides immediate access to the knowledge base, and lets you monitor any aspect of program development or execution via an integrated network of menus and windows.

The Studio also provides a tightly integrated interface to the GNU Emacs full-screen editor. This interface facilitates the ART-Ada program development process by providing a number of powerful capabilities, such as incremental compilation of ART-Ada code.

The ART-Ada Studio can be used to do the following:

- Develop and execute an ART-Ada application.
- Browse the knowledge base -- to examine declarative (facts/schemas) knowledge, procedural (rules) knowledge, and runtime state, such as matches and activations.
- Debug the knowledge base -- by setting breakpoints in the programs and tracing their execution.
- Develop applications incrementally -- by editing the knowledge base to change facts or rules, or to modify program interactively.
- Generate Ada source code.

The ART-Ada/VMS Studio is based on DECwindows. The Studio is also implemented using other user interface standards (e.g. PM, OSF/Motif, ISPF) on other platforms.

2.4 Ada Integration

A major feature of ART-Ada is its ability to integrate expert systems technology with Ada. ART-Ada supports three types of Ada integration:

- *Ada call-out* refers to an ability to call Ada subprograms (procedures and functions) from the knowledge base (rules and methods).
- *Ada call-in* refers to an ability to call ART-Ada's public functions from Ada.
- *Ada call-back* is a special case of Ada call-in and refers to an ability to call

ART-Ada's public functions from an Ada subprogram called from the knowledge base using Ada call-out.

Designers of expert systems will want to develop their own Ada code to provide user and system interfaces for their applications. There also may be a need to interface expert systems with other Ada applications (e.g. a signal processing application). A primary benefit of incorporating Ada code into the knowledge base is that Ada code will execute faster than similar code written in the ART-Ada procedural language. A consistent Ada call-in and call-out interface is provided for both development and deployment environments so that user-written Ada code runs without modification when it is deployed in Ada. In order to illustrate how an Ada subprogram is called from the knowledge base, let's consider the following rule:

```
(defrule distance-calculation-rule
  "calculates distance between airfield and base"
  (schema ?airfield
    (instance-of airfield)
    (lat ?lat1)
    (lon ?lon1))
  (schema ?base
    (instance-of base)
    (lat ?lat2)
    (lon ?lon2))
  =>
  (bind ?distance
    ;; call an Ada function to calculate distance
    (calculate-distance ?lat1 ?lon1 ?lat2 ?lon2))
  (assert (distance ?base ?airfield ?distance)))
```

The function, *calculate-distance*, can be implemented either in the ART-Ada procedural language or in Ada, but the Ada version would run faster. The ART-Ada construct *def-user-fun* specifies the interface between ART-Ada and Ada. It establishes an ART-Ada function name which calls out to the corresponding Ada subprogram, and it provides a description of data being passed. For example, *calculate-distance* can be specified as an Ada function as follows:

```
(def-user-fun calculate-distance
  :args ((lat1 :float)
         (lon1 :float)
         (lat2 :float)
         (lon2 :float))
  :returns :float
  :compiler :dec-ada)
```

This *def-user-fun* statement specifies that the ART-Ada function *calculate-distance* will call out to an Ada function CALCULATE_DISTANCE. There are four arguments of a type floating-point number being passed to Ada. The return value is also a

floating-point number. It also specifies the default Ada compiler for the platform (i.e. DEC Ada). The corresponding Ada code should be declared in a package called *USER* and would look like:

```
-- ART is a public package of ART-Ada.
with ART;
-- USER is a package for user's Ada code.
package USER is

  function CALCULATE_DISTANCE
    (LAT1, LON1, LAT2, LON2 : ART.FLOAT_TYPE)
       return ART.FLOAT_TYPE;

end USER;
```

Ada data types supported for the call-in and call-out interfaces are: 32 bit integer (INTEGER_TYPE), 64 bit float (FLOAT_TYPE), boolean (BOOLEAN_TYPE), string and symbol (STRING), and an abstract data type for objects in ART-Ada (ART_OBJECT). Table 2-1 summarizes the mapping between ART-Ada and Ada data types.

ART-Ada	Ada	Comments
integer	INTEGER_TYPE	32 Bits
float	FLOAT_TYPE	64 Bits
boolean	BOOLEAN_TYPE	-
string	STRING	-
symbol	STRING	-
art-object	ART_OBJECT	abstract data type

Table 2-1: Data Types for Ada Call-in/Call-out

2.5 Ada Code Generation

ART-Ada takes one or more ART-Ada source files as input and outputs Ada source files that represent a single Ada package. At any point after ART-Ada source files are loaded into ART-Ada and the knowledge base is initialized for execution, the Ada code generator may be invoked to generate Ada source code. An Ada package specification generated by ART-Ada for an example application called MY_EXPERT_SYSTEM is shown below:

```
-- generated automatically by ART-Ada
package MY_EXPERT_SYSTEM is

  -- initialize the application.
  procedure INIT;

end MY_EXPERT_SYSTEM;
```

A simple Ada main program that initializes and runs the application MY_EXPERT_SYSTEM is shown below. It is the simplest way to run an ART-Ada application in an Ada environment. It is possible, however, to embed it in a large Ada program. ART-Ada's public Ada packages, ART and SCHEMA, include a full set of Ada utilities to control and access procedurally the knowledge base from Ada. In OPS5, for example, it is hard to access working memory elements procedurally. In ART-Ada, Ada utilities are provided to access the knowledge base directly from Ada.

```
-- This is a main program written by the user.
-- ART is a public package of ART-Ada.
with ART, MY_EXPERT_SYSTEM;
procedure MAIN is
  TOTAL_RULES : ART.INTEGER_TYPE;
begin
  MY_EXPERT_SYSTEM.INIT;          -- initialize it.
  TOTAL_RULES := ART.A_RUN(-1);   -- run it.
end MAIN;
```

In addition to generating the Ada source code that initializes the knowledge base, a call-out interface module is generated as a separate procedure; it is a large case statement that contains all Ada subprograms called out to from ART-Ada. ART-Ada also generates a command file used to compile all Ada files generated by ART-Ada.

2.6 Ada Runtime Deployment

The methodology for developing an ART-Ada application defines three distinct platforms, some or all of which may be the same:

- an ART-Ada development platform with Ada call-in and call-out capability on which an application is actually developed and debugged;
- an Ada compiler platform on which either a self-targeted compiler or a cross-compiler is used to compile Ada source code; and
- a target platform on which an Ada executable image will be deployed.

The development phase would involve the development of an ART-Ada program and the Ada code that interfaces with ART-Ada, which occurs on the ART-Ada develop-

ment platform. The deployment phase would involve the compilation of the Ada code generated by ART-Ada and written by the user, which occurs on the platform where the Ada compiler runs. The generated Ada code contains application-specific code. The actual Ada-based inference engine is contained in the ART-Ada runtime kernel. The ART-Ada runtime kernel is provided as an Ada library. If the Ada compiler is a self-targeted compiler, the Ada executable image will be deployed on the same platform where the Ada compiler runs. If it is a cross-compiler, it will be deployed on the target platform (which may be an on-board computer).

The steps needed to deploy an ART-Ada application in Ada are summarized below:

1. Develop and debug an application using ART-Ada's interactive development environment. If necessary, call out to Ada using the call-in/call-out interface.

2. Generate Ada code from ART-Ada using the Ada code generator. If the Ada compiler platform is different from the ART-Ada development platform, the generated Ada code can be moved to the platform on which the Ada compiler runs as long as the ART-Ada runtime kernel is available for that platform.

3. Compile the generated Ada code and user-written Ada code using either a self-targeted compiler or a cross-compiler into an appropriate Ada library of the ART-Ada runtime kernel.

4. Create an Ada executable image by linking an Ada main program.

5. Deploy the Ada executable image on a host computer or on a target system.

2.7 Performance Benchmarks

Historically, benchmarking of software systems, and especially expert systems development tools, has been an area of great controversy. One problem is that the comparisons are often "apples and oranges" --- some tools are good for one thing, but not for others. Another problem is the effort required to make a benchmark program run on many different tools. This usually results in simple, toy problems being used as the basis of the benchmark study. Toy problems, however, seldom have the same performance characteristics as real applications. In particular, toy problems do not indicate how the tool responds to large knowledge bases. We selected benchmarks that specifically "stress test" the system in ways typical of large applications.

The tests selected were designed so they can be scaled to a variety of problem sizes. Each test was on problem sizes of 50, 100 and 200. Many pattern matching systems, including ART-Ada, perform pattern matching when the assertions or objects are placed

into the knowledge base. For this reason, the tests and times also include the time required to place the objects into the knowledge base. For applications with a large, relatively static, knowledge bases, the initial assertion time costs can be incurred during system development and are not realized by the deployed application. For ART-Ada applications, this can result in more than an order of magnitude speed improvement over the times listed here.

The tests were run on a Sun 3/260 with 16 megabytes of memory using the Verdix Ada compiler version 5.5(t). No attempt was made to optimize the compiled code by using a maximum optimizer option or by suppressing constraint checking.

2.7.1 Simple Constant Attributes

A large number of rules were generated, each with a different set of constant pattern. Exactly one assertion was made into the knowledge base that corresponds to each pattern. This is the simplest form of pattern matching, however, it is the only kind provided by many tools. The example below is the test of size 3. Tests of size 50, 100 and 200 were performed. For each case, one additional rule fires to initialize the knowledge base.

```
(defrule T1 (a 0)(b 0)(c 0) =>)
(defrule T2 (a 1)(b 1)(c 1) =>)
(defrule T3 (a 2)(b 2)(c 2) =>)

(defrule x =>
  (assert (a 0)(b 0)(c 0))
  (assert (a 1)(b 1)(c 1))
  (assert (a 2)(b 2)(c 2))
```

The results of the test are:

SIZE	RULES/SECOND	FIRED	TIME
50	94 rules/second	51	0.54 seconds
100	84 rules/second	101	1.20 seconds
200	59 rules/second	201	3.40 seconds

2.7.2 Simple Patterns with Co-occurring Variables

A single rule containing three patterns is in the knowledge base. The patterns contain a co-occurring variable (?i). A large number of knowledge base objects are created that can cause the rule to fire. Each knowledge base object paticipates in exactly one rule firing. The example below is the test of size 3. Tests of size 50, 100 and 200 were performed. For each case, one additional rule fires to initialize the knowledge base.

```
(defrule y (a ?i)(b ?i)(c ?i) =>)

(defrule x1 =>
  (assert (a 0)(b 0)(c 0))
  (assert (a 1)(b 1)(c 1))
  (assert (a 2)(b 2)(c 2)))
```

The results of the test are:

SIZE	RULES/SECOND	FIRED	TIME
50	88 rules/second	51	0.58 seconds
100	79 rules/second	101	1.28 seconds
200	56 rules/second	201	3.58 seconds

2.7.3 Objects with Co-occurring Variables

A single rule containing three patterns is in the knowledge base. The patterns contain a co-occurring variable (?i). A large number of objects are created that can cause the rule to fire. Objects occur in pairs, such that each pair causes one activation of the rule. The example below is the test of size 3. Tests of size 50, 100 and 200 were performed. For the tests of size 50 and 100, 1 additional rule fires to initialize the knowledge base. For the test of size 200, 2 initialization rules fire.

```
(defschema x (a)(b)(c))
(defschema y (d)(e)(f))

(defrule z
  (schema ?x (instance-of x)(a ?x1)(b ?x2)(c ?x3))
  (schema ?y (instance-of y)(d ?x1)(e ?x2)(f ?x3)))

(defrule T2 (declare (salience 1000)) =>
  (assert (schema T1
            (instance-of x)(a T2)(b T3)(c T4)))
  (assert (schema T2
            (instance-of y) (d T2)(e T3)(f T4)))
  (assert (schema T9
            (instance-of x)(a T10)(b T11)(c T12)))
  (assert (schema T10
            (instance-of y) (d T10)(e T11)(f T12)))
  (assert (schema T17
            (instance-of x)(a T18)(b T19)(c T20)))
  (assert (schema T18
            (instance-of y) (d T18)(e T19)(f T20))))
```

The results of the test are:

SIZE	RULES/SECOND	FIRED	TIME
50	38 rules/second	51	1.34 seconds
100	38 rules/second	101	2.64 seconds
200	37 rules/second	202	5.48 seconds

3. Discussion

3.1 Ada Issues

Our internal benchmark results show that the speed and size of ART-Ada is much better than that of a Lisp-based tool ART while it is somewhat slower and larger than a C-based tool ART-IM. While Ada compilers are improving, they still have not reached the maturity of C compilers. In fact, because of numerous bugs found in the Ada compilers used for this project, we could not make some of the obvious performance optimizations that could have made ART-Ada faster and smaller. In addition to the compiler problems, we also discovered some fundamental issues with the Ada language itself that also affected the performance of ART-Ada. Both of these issues will be discussed in the later sections. It has also been observed that both the speed and size of ART-Ada vary up to 30% depending on which Ada compiler is used. A recent paper discusses the key technical issues involved in producing high-quality Ada compilers [10]. As Ada compiler technology advances, ART-Ada's performance will improve.

3.1.1 Compiler Problems

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

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

- In ART-Ada, we reuse several Booch components [4]. These software components are used to implement data structures (e.g. linked lists and strings) and other utilities (e.g. quick sort). Most Booch components are implemented as generic packages using object-oriented design methodology. This means that a large number of subprograms are provided in each generic package, which may be instantiated multiple times. Unfortunately, one of the compilers does not support a feature called *selective linking* --- a linker feature that makes it possible to include only those subprograms actually used in the program. The underlying mechanism used by the compiler is the

Unix linker (ld), which does not support selective linking. As a result, whenever a generic package is instantiated and included using the *with* statement, all subprograms in the package will be always included in the executable image regardless of their actual usage. This will increase the size of the executable image.

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

3.1.2 Dynamic Memory Allocation

Due to the dynamic nature of expert systems, it is necessary to allocate memory dynamically at runtime in ART-Ada. In Ada, *new* is used to allocate memory and *unchecked_deallocation* is used to deallocate it. Our experiment shows that the average overhead of *new* in the Verdix compiler is about eighteen bytes, i.e. every time *new* is called, an extra eighteen bytes are wasted. This result is obtained by using a program that allocates the same data structure multiple times using *new* and measuring its process size with the Unix command "ps aux". We repeated the same experiment using several data structures of different size. According to Verdix, *new* eventually calls *malloc*. We tried similar experiments using the Sun C compiler. The average overhead of *malloc* was about eight bytes, which was significantly smaller than that of Ada. It is not clear why it is necessary to add extra ten bytes to every *malloc*. The only information needed to call *free* is the size of the memory, which can be obtained from the data type used to instantiate the generic procedure *unchecked_deallocation*. The exceptions are unconstrained arrays and variant records whose size can vary. For these data types, it would be necessary to add four bytes to store the size information. The actual measurement results are summarized in Tables 3-1 and 3-2. Units in these tables are bytes. The C and Ada program used are shown below:

```
#include <stdio.h>

main()
{
  int i;

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

```
with TEXT_IO;
procedure TEST_NEW is

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

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

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

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

Table 3-1: Overhead of Dynamic Memory Allocation using new in Ada

Item Size	Item Count	Ideal Size	Actual Size	Overhead	Overhead/Item
8	100,000	800 K	1600 K	800 K	8.0
16	100,000	1600 K	2384 K	784 K	7.84
24	100,000	2400 K	3160 K	760 K	7.60
32	100,000	3200 K	3944 K	744 K	17.44
8	50,000	400 K	816 K	416 K	8.32
16	50,000	800 K	1208 K	408 K	8.16
24	50,000	1200 K	1600 K	400 K	8.0
32	50,000	1600 K	1922 K	392 K	7.84
Average	N/A	N/A	N/A	N/A	7.9

Table 3-2: Overhead of Dynamic Memory Allocation using malloc in C

The real problem with this overhead is that in ART-Ada *new* is called very frequently to allocate relatively small blocks while in ART-IM *malloc* is called only to allocate large blocks (e.g. 100 Kbytes). In order to achieve maximum time and space efficiency, ART-IM has been optimized in ways that are not portable to Ada. For example, the type cast feature of the C language has been used both to optimize data structures and

to implement an internal memory manager. ART-IM's memory manager maintains its own free lists and handles all allocation and deallocation requests from the ART-IM kernel; it allocates large blocks of memory from the system, and then fulfills individual (relatively small) requests for storage from the large blocks. As storage is released, it is added to internally maintained free lists; the blocks themselves are never released back to the system. There are several advantages to this approach:

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

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

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

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

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

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

struct bar {
     .
     .
     .
};

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

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

There are two problems in implementing this in Ada:

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

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

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

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

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

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

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

The disadvantage of this approach is that large blocks of memory must be allocated

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

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

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

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

Various Ada language issues are being studied by several working groups for the Ada 9X standard [1], [2]. We believe that the issue of dynamic memory management and other issues discussed in this paper should also be considered for the Ada 9X standard.

3.1.3 Other Language Issues Related to ART-Ada Performance

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

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

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

- The variant record is the only Ada data type that can be used to implement C's union, but it is not as efficient nor flexible.

3.1.4 Portability

Ada is quite portable and probably more portable than C. Contrary to popular belief, however, Ada is not 100% portable.

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

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

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

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

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

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

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

In C, conditional compilation facilitated by preprocessor directives (e.g #define and

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

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

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

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

We define app as follows:

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

Then, we execute the following commands:

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

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

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

3.2 Hardware Issues

Although semiconductor technology is improving very rapidly in the commercial sector, embedded processors are still based on old technology. Modern operating system features such as virtual memory are not readily available on most on-board computers. The resource requirements on these computers such as processor speed and real memory are quite stringent. The Air Force standard avionics processor MIL-STD-1750A, for ex-

ample, supports address space of only 1 megaword. Another restriction is that its memory must be partitioned into multiple 64 Kbyte segments. The newer GVHSIC (Generic VHSIC) version will support 8 megaword, but the 64 Kbyte segment restriction will still remain. This is certainly too restrictive for Ada-based expert systems that would probably require at least one Mbyte of contiguous memory space. We believe, however, that ART-Ada can satisfy the resource requirements for the newer embedded processors such as the Intel 80386 and 80960, the Motorola 68000 and 88000, and the MIPS RISC chip. Another requirement for porting ART-Ada to an embedded system is the dynamic memory management support in the Ada runtime executive.

4. Related Work

IntelliCorp has done some research to develop a system for translating KEE applications into Ada [8]. On the surface, the main difficulties of the approach seem to be Ada integration during development and the translation of Lisp code to Ada. The advantage of ART-Ada is that Ada subprograms can be called directly from the knowledge base during development. Since the development environment of ART-Ada is written in C, Ada call-back is used to integrate Ada subprograms. Ada call-back simply means that the Ada *main* program calls the C program (ART-Ada development environment), which calls back to Ada subprograms. This is the only proper way to call Ada from another language such as C. Ada is not only a programming language but also a runtime environment. The use of the Ada *main* program ensures the proper initialization of the Ada runtime environment. The problem with KEE is that it is written in Lisp. Lisp is also a runtime environment like Ada. Therefore, it would be hard to start Lisp from the Ada main, which is the only way to call back to Ada from Lisp. In fact, the Lucid Common Lisp 3.0 used to implement KEE supports call-out to C, Fortran, and Pascal but not Ada. ART-Ada also supports Ada call-in; there are over 200 Ada subprograms that can be used to control and access the knowledge base procedurally. It would be impossible to implement Ada call-in in Lisp that allows Lisp functions to be called from Ada. When neither Ada call-out nor call-in is available, actions in the rule right-hand side (RHS) must be implemented in Lisp. The automatic translation of the Lisp code to Ada would alleviate the burden of manual translation if it is technically feasible. It might be possible to translate a small subset of Lisp to Ada automatically. Even so, the efficiency of the translated Ada code would not be as good as hand-written Ada code. This approach also excludes the use of existing Ada packages (for numerical analysis, signal processing, etc.) in the knowledge base. In ART-Ada, existing Ada packages can be easily integrated directly into the knowledge base even during development.

FLAC (Ford Lisp-Ada Connection) uses a Lisp environment on a Lisp machine to develop an expert system application and generate Ada code [21]. In FLAC, the knowledge base is specified using a graphical representation similar to that of VLSI design (e.g. OR gates and AND gates). Since FLAC's development environment is based on Lisp, it probably does not support Ada call-in/call-out. FLAC's knowledge base is pre-compiled and static, which means that objects may not be added or deleted dynamically at runtime although their values may be changed. This imposes major functionality restrictions that do not exist in ART-Ada.

CHRONOS is a commercial expert system tool written in Ada that was introduced recently. It is developed and marketed by a French company, Euristic Systems. Currently, little is published about this tool.

Another commercial tool is an object-oriented programming language called

Classic-Ada [25]. Its input language is based on Smalltalk, but it works like C++; it is a preprocessor that generates Ada source code. Unlike ART-Ada, the generated Ada code is self-sufficient; it does not require an Ada runtime kernel to compile it. Although Classic-Ada does not support rules, its object-oriented programming features are similar to that of ART-Ada.

It is reported that several logic-based tools support Prolog in Ada [5], [3], [14]. Although Prolog can be used to implement expert systems, its approach and scope are significantly different from expert system tools such as ART-Ada. These tools, therefore, are not covered in this paper.

5. Future Directions

During the next several months, various sites selected by NASA and the Air Force will be involved in the ART-Ada evaluation project. During this project, they will try to implement prototype expert systems for the Space Station Freedom Program and other Air Force applications to understand the potential uses and operational issues of ART-Ada. At the end of the project, they will write a report about their findings.

At the same time, Inference will continue its research effort in the following areas to better support real-time, embedded and distributed expert systems:

- to implement real-time enhancements for ART-Ada,
- to support ART-Ada on embedded processors, and
- to parallelize ART-Ada on a shared-memory multiprocessor.

We have identified four areas of enhancements for real-time support in ART-Ada:

- a set of tools for performance monitoring and tuning,
- temporal reasoning and trend analysis,
- kernel support for dynamic priority scheduling, and
- communications package for multiple cooperating ART-Ada processes.

The performance of an ART-Ada application varies widely depending on how it is implemented. It is often necessary to monitor activities in the pattern matcher (e.g. the number of pattern instantiations, partial matches, activations, etc.) or the execution time of a rule RHS action in order to determine areas for optimization. Performance analysis can be aided by a set of tools that graphically display information on the ART-Ada execution. It is also possible to automate manual optimization process. It has been reported that an automated tool was successfully used to optimize join ordering [20].

A real-time monitoring-and-diagnosis expert system refers to an application that monitors incoming data and performs fault diagnosis. In such a system, it is often necessary to reason about and perform statistical analysis on *temporal* data -- data that change over time. In order to avoid information overloading, several levels of abstraction should be used. Raw data are usually preprocessed to suppress noices and redundant data and stored in a ring buffer. Historical data do not participate in the pattern-matching process directly. Rather, high-level abstration acquired by applying temporal reasoning and trend analysis to the historical data is used in the knowledge base for fault diagnosis and other high-level tasks.

In recent years, several real-time architectures based on the blackboard architecture have been reported [12], [6]. In these real-time expert systems, priority can be dynamically determined based on the timing constraints and the resource requirements of a task. Currently, in ART-Ada rule priorities cannot be changed dynamically. We believe that ART-Ada can be enhanced to implement these blackboard architectures with dynamic priority scheduling in a number of ways.

Multiple cooperating applications of ART-Ada can run either as multiple processes on a single processor (e.g. Sun) or on loosely coupled multiple processors (e.g. multiple embedded processors on a bus or multiple Suns on the network). The communications package for message passing between multiple ART-Ada processes can be implemented using various interprocess communications protocols (e.g. Unix sockets, VMS mailboxes. Ada tasks). The asynchronous function facility in ART-Ada will be used to poll the message queue between rule firings. The advantage of this approach is that it does not require a parallel computer with a shared memory and is suitable for embedded systems or distributed processors on a network.

ART-Ada can be parallelized using node parallelism on a shared-memory architecture (e.g. Encore Multimax). Node parallelism exploits parallelism at the join node by calculating matches in parallel. It has been reported that this approach can speed up OPS5 programs up to 10-fold [11]. It may not be possible, however, to exploit node parallelism on an embedded system because most embedded systems do not support a shared memory architecture.

References

1. Ada Language Issues Working Group. "Ada Language Issues Working Group (ALIWG) Minutes of 17 August 1988". *Ada Letters IX*, 1 (January/February 1989).

2. Ada Runtime Environment Working Group. "Activities of the Ada Runtime Environment Working Group". *Ada Letters IX*, 5 (July/August 1989).

3. Bobbie, P.O. ADA-PROLOG: An Ada System for Parallel Interpretation of Prolog Programs. Proceedings of the Third Annual Conference on Artificial Intelligence and Ada, 1987.

4. Booch, G. *Software Components With Ada.* Benjamin/Cummings Publishing, 1987.

5. Burback, R. PROVER: A First-order Logic System in Ada. Proceedings of the Third Annual Conference on Artificial Intelligence and Ada, 1987.

6. Dodhiawala, R. et. al. Real-Time AI Systems: A Definition and An Architecture. Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI, 1989.

7. Dzierzanowski, J.M. et. al. The Authorizer's Assistant: A Knowledge-based Credit Authorization System for American Express. Proceedings of the Conference on Innovative Applications of Artificial Intelligence, AAAI, 1989.

8. Filman, R.E., Bock, C., and Feldman, R. Compiling Knowledge-Based Systems Specified in KEE to ADA. Final Report, NASA Contract NAS8-38036, IntelliCorp Inc., August, 1989.

9. Forgy, C.L. "RETE: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem". *Artificial Intelligence 19* (1982).

10. Ganapathi, M., Mendal, G.O. "Issues in Ada Compiler Technology". *Computer 22*, 2 (February 1989).

11. Gupta A. et. al. Results of Parallel Implementation of OPS5 on the Encore Multiprocessor. CMU-CS-87-146, Carnegie-Mellon University, Department of Computer Science, August, 1987.

12. Hayes-Roth, B. et. al. Intelligent Monitoring and Control. Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI, 1989.

13. Hugh, D.A. "The Future of Flying". *AI Expert 3*, 1 (January 1988).

14. Ice, S., et. al. Raising ALLAN: Ada Logic-Based Language. Proceedings of the Third Annual Conference on Artificial Intelligence and Ada, 1987.

15. Inference Corporation. Ada-ART, Specification for an Ada-based State-of-the-Art Expert System Construction Capability. Inference Corporation, August, 1987.

16. Inference Corporation. *ART Version 3.2 Reference Manual.* Inference Corporation, 1988.

17. Inference Corporation. ART/Ada Design Project - Phase I, Final Report. Inference Corporation, March, 1989.

18. Inference Corporation. *ART-Ada/VMS 2.0 Beta Reference Manual.* Inference Corporation, 1989.

19. Inference Corporation. *ART-IM/VMS 2.0 Beta Reference Manual.* Inference Corporation, 1989.

20. Ishida, T. Optimizing Rules in Production System Programs. Proceedings of the National Conference on Artificial Intelligence, AAAI, 1988.

21. Jaworski, A., LaVallee, D., Zoch, D. A Lisp-Ada Connection for Expert System Development. Proceedings of the Third Annual Conference on Artificial Intelligence and Ada, 1987.

22. Lee, S.D., Allen, B.P. Deploying Expert Systems in Ada. Proceedings of the TRI-Ada Conference, ACM, 1989.

23. Nakashima, Y, Baba, T. OHCS: Hydraulic Circuit Design Assistant. Proceedings of the Conference on Innovative Applications of Artificial Intelligence, AAAI, 1989.

24. O'Brien, J. et. al. The Ford Motor Company Direct Labor Management System. Proceedings of the Conference on Innovative Applications of Artificial Intelligence, AAAI, 1989.

25. Software Productivity Solutions, Inc. *Classic-Ada User Manual.* Software Productivity Solutions, Inc, 1988.

I. Beta Test Sites and Contacts

I.1 NASA Sites

Gary Riley — Sun 4 with Verdix
NASA/Johnson Space Center
Mail Code FM 72
Houston, TX 77058
(713)483-8073
GRILEY%nasamail@ames.arc.nasa.gov

Brandon Dewberry — Sun 3 with Verdix
NASA/Marshall Space Flight Center
Mail Code EB42
Huntsville, AL 35812
(205)544-4247
BDEWBERRY%nasamail@ames.arc.nasa.gov

Robert Jones — Sun 4 with Verdix
NASA/Lewis Research Center
Mail Code 54-8
Cleveland, OH 44135
(216)433-3457
RJONES%nasamail@ames.arc.nasa.gov

Rich Knackstedt — VAX/VMS with DEC Ada
McDonnell Douglas Space Systems Company
Space Station Division
5301 Bolsa
Mail Code A95/J845/22-3
Huntington Beach, CA 92647
(714) 896-2296
RKNACKSTEDT%nasamail@ames.arc.nasa.gov

I.2 USAF Sites

Bill Baker — VAX/VMS with DEC Ada
WRDC/TXI
WPAFB, OH 45433
(513)255-5800
BAKER%aruba.dnet@wpafb-avlab.arpa

Lori Attias — VAX/VMS with DEC Ada
McDonnell Aircraft Company

AI Center
314/105/1065205
PO Box 516
St. Louis, MO 63166
(314)232-1858
FAX: (314)232-7499

Stephen Bate
McDonnell Aircraft Company
Mail Code: 064 4244
PO Box 516
St. Louis, MO 63166
(314)232-5844
FAX: (314)777-6293

VAX/VMS with DEC Ada

Joe Hintz
Raytheon Company
Missle Systems Division
Mail Code: T3SU21
Tewksbury, MA 01876-0901
(508)858-5907
JCH@swlvx2.ray.com

VAX/VMS with DEC Ada