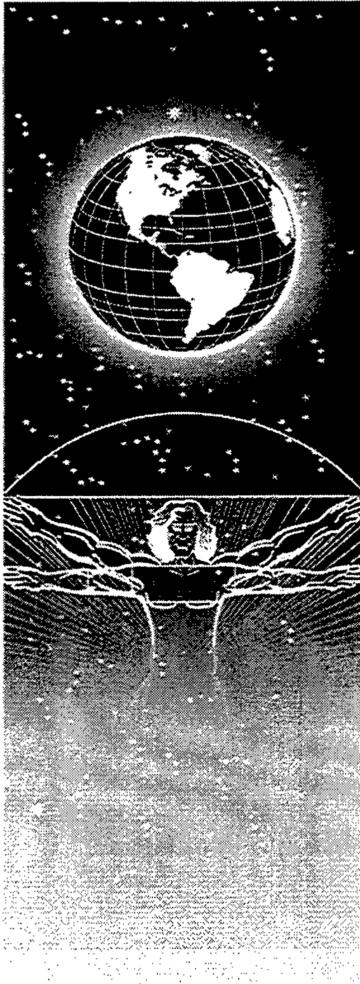
AFRL-HE-WP-TR-1999-0023



UNITED STATES AIR FORCE RESEARCH LABORATORY

The Distributed Operator Model Architecture

Stephen E. Deutsch
Nichael Cramer
George Keith
Bobbi Freeman

BBN Technologies
10 Moulton Street
Cambridge MA 02138

February 1999

Final Report for the Period November 1997 to February 1999

19990618 176

Approved for public release; distribution is unlimited.

Human Effectiveness Directorate
Deployment and Sustainment Division
Sustainment Logistics Branch
2698 G Street
Wright-Patterson AFB OH 45433-7604

DTIC QUALITY INSPECTED 4

NOTICES

When US Government drawings, specifications, or other data are used for any purpose other than a definitely related Government procurement operation, the Government thereby incurs no responsibility nor any obligation whatsoever, and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise, as in any manner, licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use or sell any patented invention that may in any way be related thereto.

Please do not request copies of this report from the Air Force Research Laboratory. Additional copies may be purchased from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Federal Government agencies and their contractors registered with Defense Technical Information Center should direct requests for copies of this report to:

Defense Technical Information Center
8725 John J. Kingman Rd., STE 0944
Ft Belvoir VA 22060-6218

DISCLAIMER

This Technical Report is published as received and has not been edited by the Air Force Research Laboratory, Human Effectiveness Directorate.

TECHNICAL REVIEW AND APPROVAL

AFRL-HE-WP-TR-1999-0023

This report has been reviewed by the Office of Public Affairs (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER

JAY KIDNEY, Lt Col, USAF, Chief
Deployment and Sustainment Division
Air Force Research Laboratory

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
February 1999

3. REPORT TYPE AND DATES COVERED
Final - November 1997 - February 1999

4. TITLE AND SUBTITLE
The Distributed Operator Model Architecture

5. FUNDING NUMBERS
C - F41624-97-D-5002
PE - 62202F
PR - 1710
TA - D0
WU - 04

6. AUTHOR(S)
Stephen E. Deutsch, Nichael Cramer, George Keith and Bobbi Freeman

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
BBN Technologies
10 Moulton Street
Cambridge MA 02138

8. PERFORMING ORGANIZATION REPORT NUMBER
BBN No. 8254

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory, Human Effectiveness Directorate
Deployment and Sustainment Division
Air Force Materiel Command
Sustainment Logistics Branch
Wright-Patterson AFB OH 45433-7604

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-HE-WP-TR-1999-0023

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Distributed-OMAR (D-OMAR) is a new implementaion of the Operator Model Architecture (OMAR) designed to operate in a distributed computing environment. In the new network-based configuration, Core-OMAR operates as a server that provides the OMAR simulator. The developer's interface operates as a client that may connect to any of the instances of Core-OMAR operating on the network. As in OMAR, D-OMAR is designed as a simulation environment in which to create human performance models. It also provides the capability to develop the complex multi-tasking agents that are needed in agent-based systems. The representation languages -- the Simple Frame Language (SFL), the SCORE procedural language, and the rule language -- are provided to facilitate model development. The developer's interface -- including the Simulation Control panel, the Concept Editor, the Procedure Browser, and the Agent and Event Timeline analysis tools -- has been re-implemented in Java. The architecture for D-OMAR was designed to enable operation in several different middle-ware environments. Currently supported middle-ware layers include Java RMI, CORBA, and the Defense Modeling and Simulation Organization's High Level Architecture. A series of Combat Air Patrol scenarios, used as test cases to support D-OMAR development, are available as demonstration scenarios.

14. SUBJECT TERMS
intelligent agents, agent-based architectures, human computer interfaces, agents, operator model architectures, OMAR

15. NUMBER OF PAGES
42

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

The work on the Distributed Operator Model Architecture was conducted under Delivery Order 13 of the Logistics Technology Research Support (LTRS) program administered under U. S. Air Force Contract Number F41624-9T-D-5002.

The authors wish to thank the Technical Monitor, Dr. Michael J. Young of Air Force Research Laboratory's Sustainment Logistics Branch, for the many important contributions that he has made to this research effort.

Table of Contents

1. **INTRODUCTION AND OVERVIEW** 1

2. **DISTRIBUTED-OMAR** 2

2.1 TRANSITIONING OMAR TO D-OMAR 3
2.2 THE D-OMAR MIDDLE-WARE LAYER 6
2.3 LINKING CORE-OMAR INTO THE DISTRIBUTED OPERATING ENVIRONMENT 7
2.3.1 The OMAR Connector and Its Connectors 8
2.3.2 The Lisp to Java Connection 9
2.4 THE EXTERNAL CONNECTOR 10
2.4.1 The Core-OMAR External Connector 11
2.4.2 The External Connector for the High Level Architecture 11
2.4.3 The External Connector for CORBA 13
2.5 THE DEVELOPER'S GUI CONNECTOR AND THE DEVELOPER'S INTERFACE 13
2.5.1 D-OMAR Developer's Interface Client-Server Operation 14
2.5.2 The Concept Editor 14
2.5.3 The Procedure Browser 16
2.5.4 The Simulator Control Panel 19
2.5.5 The Timeline and Analysis Data 19
2.6 THE APPLICATION GUI CONNECTOR 22

3. **D-OMAR AGENTS AND AGENT-BASED SYSTEMS** 22

3.1 AGENCY IN HUMAN MODELS AND IN AGENT-BASED SYSTEMS 22
3.1.1 Agency in Human Performance Models 23
3.1.2 Agency in Agent-based Systems 23
3.2 AGENTS IN D-OMAR 24
3.2.1 Agent-Agent Communication 24
3.2.2 Proactive and Reactive Behaviors 25
3.2.3 Multi-task Performance 26
3.2.4 Invoking a Goal or a Procedure on an Agent 27
3.3 D-OMAR AGENT-BASED SYSTEMS 28
3.3.1 Homogeneous and Heterogeneous Agents 28
3.3.2 Agent Access to Application Data 29
3.3.3 Agent Mobility 30

4. **THE COMBAT AIR PATROL DEMONSTRATION** 30

4.1 CAP SCENARIO AGENTS 30
4.2 CAP SCENARIO MODES OF OPERATION 32

5. **ACRONYMS** 34

6. **REFERENCES** 35

Figures

FIGURE 1. D-OMAR CLIENT-SERVER CONFIGURATION 4
FIGURE 2. D-OMAR DEVELOPER'S CONFIGURATION 5
FIGURE 3. IMPLEMENTATION OPTIONS FOR DISTRIBUTED OBJECT SERVICES 6
FIGURE 4. D-OMAR CONNECTIVITY FOR CORE-OMAR 9
FIGURE 5 A D-OMAR OPERATING AS AN HLA FEDERATE 12
FIGURE 6 D-OMAR TOOLBAR 14
FIGURE 7 CONCEPT EDITOR NETWORK 15
FIGURE 8 CONCEPT EDITOR TABLE 16
FIGURE 9 PROCEDURE BROWSER SUB-PROCEDURES VIEW 17
FIGURE 10 PROCEDURE BROWSER SIGNALS GENERATED OVERVIEW 18
FIGURE 11 SIMULATOR CONTROL PANEL 19
FIGURE 12 AGENT TIMELINE DISPLAY 21
FIGURE 13 EVENT TIMELINE DISPLAY 21
FIGURE 14 D-OMAR AND HETEROGENEOUS AGENTS 29
FIGURE 15. CAP SCENARIO RADAR SYSTEM WORKPLACE 33

THIS PAGE INTENTIONALLY LEFT BLANK

1. Introduction and Overview

Distributed-OMAR (D-OMAR) is a new implementation of the Operator Model Architecture (OMAR) developed by BBN Technologies under the Air Force Research Laboratory's Logistics Technology Research Support (LTRS) program in Delivery Order 13. The principal goal of OMAR was to provide a sophisticated simulation environment in which to design and implement human performance models. This has also been an important focus for D-OMAR. A second goal for D-OMAR was derived from the recognition that the suite of OMAR software tools used in the development of human performance models could readily be used in the development of software agents to be employed in agent-based systems.

To make D-OMAR a viable player in a distributed computing environment a new look at the OMAR architecture was required. It was also apparent that the nature of the computing environments in which D-OMAR might be called upon to play in could be quite different than in the past. D-OMAR human performance models might be players in a Defense Modeling and Simulation Organization (DMSO) High Level Architecture (HLA) simulation exercise. As an agent-based system, D-OMAR might be a player in a large-scale CORBA-based application with many other players. Moreover, one might still want to use D-OMAR in a desktop environment without the overhead of either HLA or CORBA. The D-OMAR architecture has been designed to compartmentalize the interface to the middle-ware layer so that each of these target applications areas can be accommodated.

The design of the new architecture also required taking a new look at the major software components that make up OMAR and how they should be reconfigured to operate in a distributed computing environment. One subset of the OMAR components became the basis for a simulator that acts as a server in D-OMAR. Another component cluster, the elements that support OMAR user interface development, was re-implemented as a D-OMAR client. The client takes two forms: the first to operate as a D-OMAR developer's workplace, and the second to serve as the basis for building the user interfaces for application systems.

The OMAR simulator and the representational languages—the frame language, the procedural language, and a rule language—were viewed as being essential building blocks for the new implementation of D-OMAR. Taken together, they formed the set of OMAR components that we now call Core-OMAR. D-OMAR can include any number of Core-OMAR servers with one or more operating at any given network node.

The simulator control panel, the graphical editors and browsers for the frame and procedural languages, and the data collection and analysis tools form the essential elements that have been carried forward to form the new developer's workplace. It operates as a client in D-OMAR.

Platform independence was another goal for D-OMAR. OMAR, written in Common Lisp, had previously operated on Unix workstations. The desire to operate on Windows machines brought forward the issue of the implementation language for D-OMAR. Core-OMAR, comprising the simulator and the representation language was expected to, and in

fact did, readily compile in Common Lisp on both Unix and Windows computers. The language for the D-OMAR developer's interface and that to be used for the development of application interfaces had to be addressed. The middle-ware options also impacted the language decision. Java was selected as the language that D-OMAR uses for socket-based network connections, the interface to the middle-ware network connections, and the language for the D-OMAR developer's interface. The middle-ware connection and application interface development can be supported in Java or C++.

Lastly, it was important to provide a demonstration of D-OMAR in operation. A Combat Air Patrol (CAP) scenario was developed and has served as the testing environment for much of the development of D-OMAR itself. The scenario includes aircraft models, radar models, and human performance models for the principal players in the scenario: the pilot, co-pilot, and a radar officer on the AWACS, and the pilots for the interceptor and bogey aircraft. The scenario runs with each of the D-OMAR middle-ware layers: Java RMI, HLA, and CORBA. It is also possible to have a human player stand in for the human performance model for the radar officer using the radar workplace to control the intercept of the bogey.

Chapter 2 presents the design of the architecture for D-OMAR and outlines the implementation of that design. Chapter 3 provides background on those elements of D-OMAR that will be important in using D-OMAR for the development of human performance models and for the development of agents to operate in agent-based systems. Chapter 4 describes the CAP scenario that provides a demonstration of D-OMAR operating in several of its many potential operating environments.

2. Distributed-OMAR

The design and development of Distributed-OMAR was the central undertaking of this research effort. The Operator Model Architecture (OMAR) provided the starting point for this task. OMAR had proved to be a very successful simulation environment in which to develop human performance models, workplace models at which the operator models performed their tasks, and models of the entities needed to complete the simulation environment. Using OMAR, workplace models were created that could be operated either by human performance models or by human players (Cramer, 1995). This capability provided the basis for the evaluation of human performance models by comparing their performance with that of human subjects in experiment scenarios. In a similar manner, OMAR provided the simulation base in which to examine alternative approaches to developing automation aides (e.g., MacMillan, Deutsch, & Young, 1997) as components in complex systems.

Several factors motivated the decision to design and build a distributed version of OMAR. The most important was the understanding that OMAR embodied software capabilities that had the potential to make important contributions in two very broad and important application areas. Human performance models exhibiting skilled behaviors are essential in building real-time training environments like those currently under development by the Air Force and the other services. They are needed to drive simulation entity behaviors both at the individual platform engagement level and at command and control levels. They may be used to supplement friendly forces and to provide realistic

adversarial forces. They may be used semi-autonomously, that is, under the management of human controllers, or their development may be extended to the point where they may be reliably used as autonomous agents.

Air Force system development based on agent-based systems was viewed as a second application area in which a distributed version of OMAR could be employed. The concepts for agency implemented in human performance model development are readily adaptable for use as the basis for the design of computer-based agents. The multiple task, communication, and collaboration skills employed in human performance models are closely related to those necessary in the development of computer-based agents. These skills may be employed to guide the computer side of interactions with human users and the agents may also be tasked as subordinates by human operators to provide critical supporting services.

OMAR was originally conceived of as an "open system." That is, the option of connecting it into a broader computing environment was an important requirement met by the original architecture. The OMAR simulator has been linked to aircraft models provided by the NASA Ames Research Center (Deutsch, Cramer, & Clements, 1996) and to the University of Pennsylvania's anthropomorphic model, Jack (Deutsch, MacMillan, Cramer, & Chopra, 1997). In the first case, OMAR human performance models for commercial aircraft flight crews operated as the crew for the NASA aircraft simulator, while in the latter case, Jack provided an anthropomorphic form for an OMAR aircraft maintenance crew person. To address the challenges presented by the Air Force application areas highlighted above, the open system capabilities of OMAR had to be updated and formalized to operate appropriately in the distributed computing environments that will characterize these applications. OMAR-based systems would have to inter-operate with non-OMAR systems and large-scale applications would require an OMAR capable of operating at multiple sites supported by appropriate communication services.

Lastly, OMAR was conceived of as a suite of software tools capable of supporting the development of human performance models using a broad range of computational tools rather than providing a computational architecture representative of a particular cognitive architecture. While some models have demanded relatively modest computational resources, others have employed as many as a thousand agents in the detailed modeling of human reading skills (Young, 1998). Plans for future work are envisioned to include models that demand an order of magnitude increase in the number of agents employed.

2.1 Transitioning OMAR to D-OMAR

Given the goal of recreating OMAR to operate in a distributed system environment, it was important to identify the basic components that made up OMAR and how they should be brought forward to operate in the new environment. To operate in a distributed environment, OMAR had to be reconfigured into client and server components. The server side is centered on the simulator. The principal elements include the simulator itself and the three languages that are the representational basis for model building: the frame language, the procedure language, and the rule language. The simulator and the three representation languages are the essential computational-, as opposed to user

interface-elements, necessary to implement and execute an OMAR application. The server configuration is now known as Core-OMAR.

The client sides of the client-server configuration have two basic forms. The first provides the model developer with a user interface at which the development of application code can take place. The second form that the client side will take will be the application interfaces developed to serve the application system users. Figure 1 depicts the client-server configuration for D-OMAR. The lower portion of the figure depicts two Core-OMAR server sites each of which are running an OMAR simulator. The upper portion of the figure depicts three D-OMAR client sites, one serving as a developer's site, and two serving as application sites.

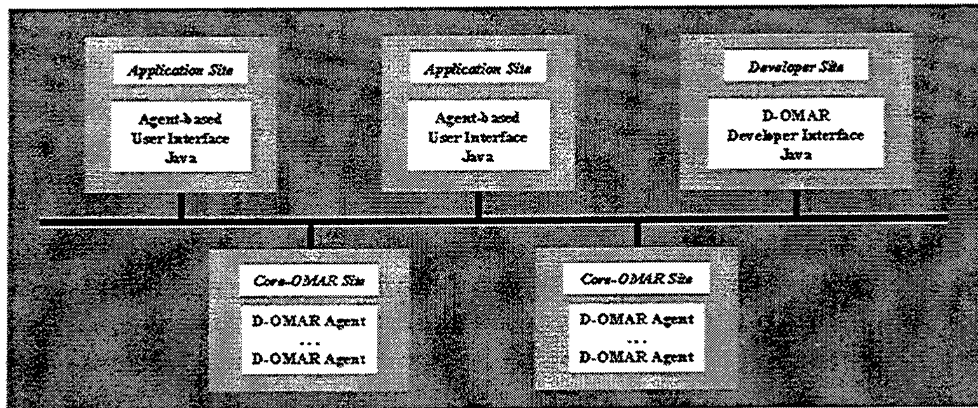


Figure 1. D-OMAR Client-Server Configuration

The D-OMAR developer's workplace consists of two principal elements: a server running Core-OMAR and a developer's user interface as depicted in Figure 2. The languages, their compilers, and the OMAR simulator are installed at the server as Core-OMAR running either locally or remotely. The developer's user interface provides the software development tools for application and model development. The development tools include the graphical editor for the Simple Frame Language (SFL), the graphical browser for the procedure language (SCORE), a window from which to manage the execution of the simulator, and access to the timeline displays that provide detailed data on model execution.

In the transition from OMAR to D-OMAR, it was important to revisit the choice of implementation language for each major software element to be carried forward. The availability of middle-ware options was the critical factor that framed many of the choices. The issue to be addressed was language selection for the existing OMAR software elements to be brought forward to be part of D-OMAR. The broader issue of the selection of the middle-ware substrate and its role in D-OMAR is covered in Section 2.2. The legacy OMAR system was implemented using a Common Lisp that included the Common Lisp Object System (CLOS) as defined in Steele (1990). The particular implementation selected was Allegro Common Lisp from Franz, Inc. The user interface code was developed using the Common Lisp Interface Manager (CLIM) also provided by Franz. OMAR was developed and supported on Sun and Silicon Graphics Unix workstations.

The new implementation of D-OMAR, to be based on a client-server configuration, provided the opportunity to revisit language selection on a component-by-component basis. The outlook for continued support for CLIM did not look good. In particular, the porting of CLIM to the Windows environment had been promised for a long time but was progressing very slowly. The much needed platform independence for user interface code made necessary a move to another implementation language. Java as a provider of platform independence was seen as the most attractive alternative. As will be seen below (in Section 2.2), Java RMI was selected as the basic middle-ware support layer, lending support to the selection of Java as the language for interface development. There were and continue to be some concerns about the performance of Java code, but the feeling is that these concerns will be addressed by the larger community of Java implementers and users. Moving the large body of OMAR code for the simulator control panel, the graphical language editors and browsers, and the analysis displays to Java was assessed to be the best long-term approach to code maintenance. The new D-OMAR client-side software, like the middle-ware software layer, is now in Java.

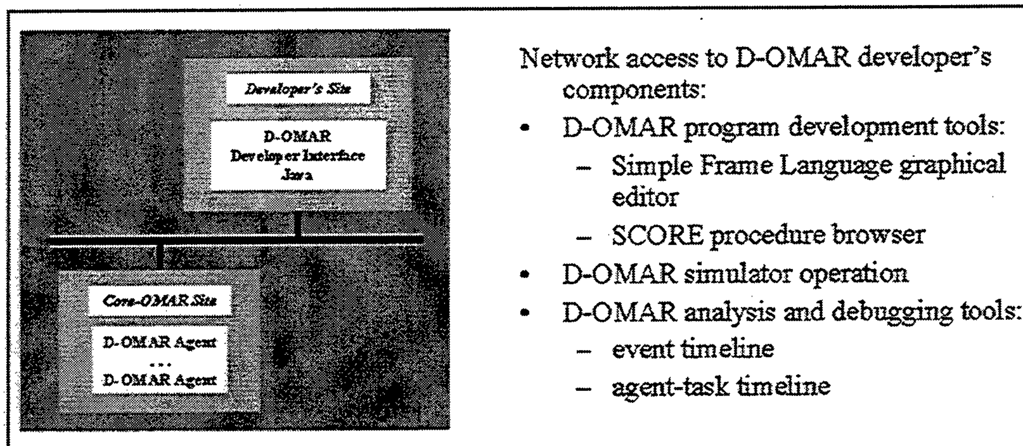


Figure 2. D-OMAR Developer's Configuration

The server-side software, as we have seen, consists of just the simulator, and the representation languages and their compilers. The frame language, SFL, and the procedure language, SCORE, rely heavily on Lisp and there is little likelihood of maintaining their representational strengths and ease of use in either Java or C++, the most reasonable candidates for re-implementation. SCORE language forms, as extensions of Lisp, are readily passed through the Lisp compiler to generate the continuation and closures that are the basis for execution in the simulator. There seemed to be little chance of preserving the significant representational strengths of the OMAR programming languages were the attempt made to re-implement them in another language. With Franz now providing Allegro Common Lisp for Windows NT, Windows 95, and Linux environments in addition to the already available Unix environments, the soundest course of action was to continue to use Lisp for server-side D-OMAR development.

2.2 The D-OMAR Middle-ware Layer

Implementing Distributed-OMAR in a distributed object system framework was the principal goal of this endeavor. That said, there were a number of implementation options and each was expected to have implications that the architecture for Distributed-OMAR would have to accommodate. This was further complicated by the fact that the offerings in distributed object frameworks are rapidly evolving with a number of major software vendors playing key roles and many smaller players making innovative contributions. The major commercial players are Sun with Java and the Remote Method Invocation (RMI); the Object Management Group (OMG), a consortium defining the specifications for CORBA with vendors offering CORBA-based software; and Microsoft with DCOM. In addition, in the military simulation and modeling world guided by the Defense Modeling and Simulation Office (DMSO), compliance with the High Level Architecture (HLA) is required. Of these candidates, the DCOM option was the only option that was not implemented. Should it become necessary, D-OMAR can readily be configured to operate with DCOM as the middle-ware layer. Figure 3 highlights the three middle-ware requirements to be met in developing Distributed-OMAR.

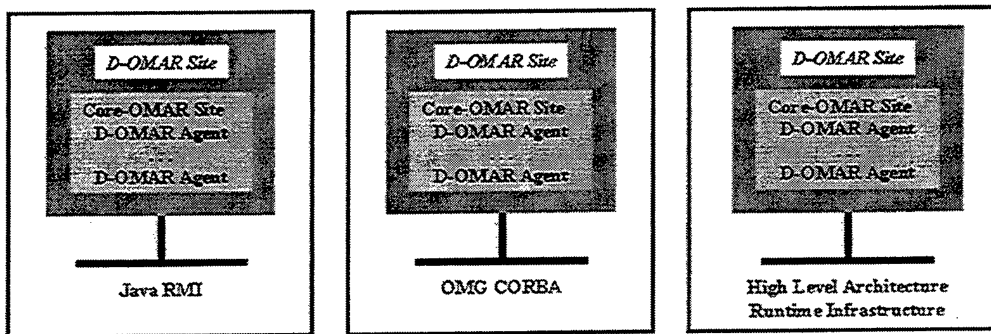


Figure 3. Implementation Options for Distributed Object Services

That said, the most straightforward approach to creating Distributed-OMAR was to supplement the existing OMAR framework with Java-based extensions. The Lisp-based Core-OMAR sites were provided with a bridge between Lisp and Java using a socket-based communication layer. Network connectivity among D-OMAR sites to support message passing, event processing, and object naming services was developed in Java. This communication layer was developed using Java RMI. This is the form that the first D-OMAR implementation took. It also formed the baseline from which the subsequent CORBA and HLA implementations were developed.

The objective of using Distributed-OMAR as part of on-going military applications played an important role in the network-connectivity implementation decision. CORBA has been the middle-ware substrate of choice for a number of military systems that include legacy system components, legacy databases, and legacy simulators as system components. A major CORBA feature is its capability to accommodate system components developed in different computer languages operating on platforms running different operating systems. Some of these systems are already using Lisp-based components. In several application areas, CORBA is already in use, or is likely to be the

required vehicle for network connectivity. It was deemed essential that D-OMAR operate in a CORBA environment.

In the simulation and modeling area, the High Level Architecture (HLA) is required and is expected to supply many essential connectivity services. The High Level Architecture is the DMSO defined standard architecture to support the integration and coordinated execution of military simulators and models. Like HLA, OMAR roots lead back to SIMNET. Predecessor systems (Deutsch, 1993) to OMAR played an important role in the early development of SIMNET Semi-automated Forces. D-OMAR now has the potential to play a significant role in a variety of the HLA simulation environments, both for agent-based supporting roles and for modeling human players or military units.

To enable D-OMAR to operate in the broadest possible range of application areas, the challenge was to provide a D-OMAR architecture that would allow an implementation that could be readily adapted to make use of any one of these middle-ware support layers. Distributed object system support was addressed at an architectural level so that the use of a particular middle-ware layer could be resolved as a low impact implementation level issue.

2.3 Linking Core-OMAR into the Distributed Operating Environment

As its name implies, Core-OMAR comprises the principal internal functional elements of OMAR, the simulator, and the representational languages from which models and applications are built. The first step in building D-OMAR was to separate this body of code and establish it as the basis for a D-OMAR server.

Written in Common Lisp, Core-OMAR can now run under most popular operating systems on most hardware platforms. Franz provides Allegro Common Lisp that operates on most Unix systems, Linux, Windows NT, and Windows 95. All Lisp development work in D-OMAR has been done using Allegro Common Lisp. Allegro Common Lisp includes two extensions that D-OMAR relies on. The first is the ability to run multiple processes. In OMAR and in Core-OMAR one process is used for simulator execution and another process is used in communicating asynchronously with the user interface. This use of processes is internal to Core-OMAR and not a feature of the D-OMAR architecture.

The second extension that Allegro Common Lisp provides is a socket capability available directly from Lisp. The socket capability is the first step in linking Core-OMAR to operate in a distributed operating system environment. Our goal was to enable OMAR agents operating at remote sites to use signals as the basis for communication just as they do for communication among agents operating in a local simulator.

Signals have always formed an important communication function among an agent's procedures within OMAR. The SCORE form, *signal-event*, is used to generate or publish a signal. The signals themselves are just lists with the first element of the list interpreted as the signal type. Procedures may subscribe to a signal using the forms *with-signal* or *asynch-wait*. There may be multiple procedures that have subscriptions to a signal type or possibly, none at all. The handling of signals forms a publish-subscribe protocol. It differs from the typical publish-subscribe strategy in that a subscription expires on the

acceptance of a signal and must be renewed if subsequent signals of that type are to be handled. This model of inter-procedure communication has been extended to form the basis for communication between instances of Core-OMAR agents operating at remote sites. The signal-based communication protocol was used as the base from which to construct the Application Program Interface (API) for Core-OMAR.

The approach developed was designed to establish three important features of the D-OMAR system for communication among D-OMAR component elements and with external systems:

1. There was to be a single, well-established protocol for the implementation of all connections to Core-OMAR. On the Lisp side, establishing a new connection-type simply involves defining a new server-connection object that knows how to deal internally with the set of connection-specific types of messages. On the external side, a signal/message-passing connection object exists in Java (or C++) that serves as the basis for the portion of the module implemented in the external language.

2. When viewed from an external Java or C++ system component, Core-OMAR appears as just another Java or C++ system component.

3. The D-OMAR components may run on a single hardware platform or the individual components, having socket-based connections, may be installed at remote network sites. Taking advantage of socket-based connectivity, each of the D-OMAR modules can run on different local or remote hardware platforms as required to meet system-specific architectural goals.

2.3.1 The OMAR Connector and Its Connectors

The OMAR Connector (Figure 4) plays a central role in the D-OMAR architecture. It is the implementation of the API for Core-OMAR and, as such, governs communication with Core-OMAR. The connector serves as a gateway to the user interfaces to Core-OMAR and to remote servers that may be either other instances of Core-OMAR or non-OMAR system components. Each of the three connectors that communicate with the Core-OMAR through the OMAR Connector uses a common body of code to manage communication with the OMAR Connector. Each is specialized to manage the D-OMAR signals particular to its application.

The D-OMAR user interfaces address two distinct requirements. The first is to provide an interface for the developer of D-OMAR-based systems. The second is to provide user interfaces for applications developed using D-OMAR. The D-OMAR Developer and Application GUI connectors shown in Figure 4 provide network connectivity for these user interfaces. Java RMI is used to support these communication links.

The third connector shown in Figure 4, the Java External Connector, provides connectivity to remote instantiations of Core-OMAR or non-OMAR systems operating in the distributed computing environment. Communication among remote Core-OMAR servers is based primarily on the exchange of OMAR signals. When connecting D-OMAR to non-OMAR systems it will be important to present an appropriate "face" to the external world. This is the function of the Java External Connector. While Core-OMAR

is Lisp-based, it presents itself to remote clients as Java-based. A C++ form of the connector is also available. It is the External Connector that has been specialized to accommodate each of the selected middle-ware layers that D-OMAR currently supports. Three middle-ware specific specializations have been developed: one using Java RMI, one that uses CORBA, and one that operates in an HLA RTI simulation environment.

Before the full implementation of the three connectors shown in Figure 4 were in place, there were actually two distinct users of the OMAR Connector during the D-OMAR development process. As soon as Core-OMAR was operational, a simple "command line" interface was developed and used in validating Core-OMAR operation. The command line interface provided a basic set of keyboard commands to control the operation of the OMAR simulator. A trace of scenario execution provided the only view into simulator operation. Once the operation of Core-OMAR was assured, a second instantiation of the OMAR Connector was used to link Core-OMAR to the original CLIM-based OMAR user interface tools. This provided a framework in which to develop OMAR code in the period before the Java-based D-OMAR developer's interface was available. The successful operation of Core-OMAR and its API was established quite early in the development process.

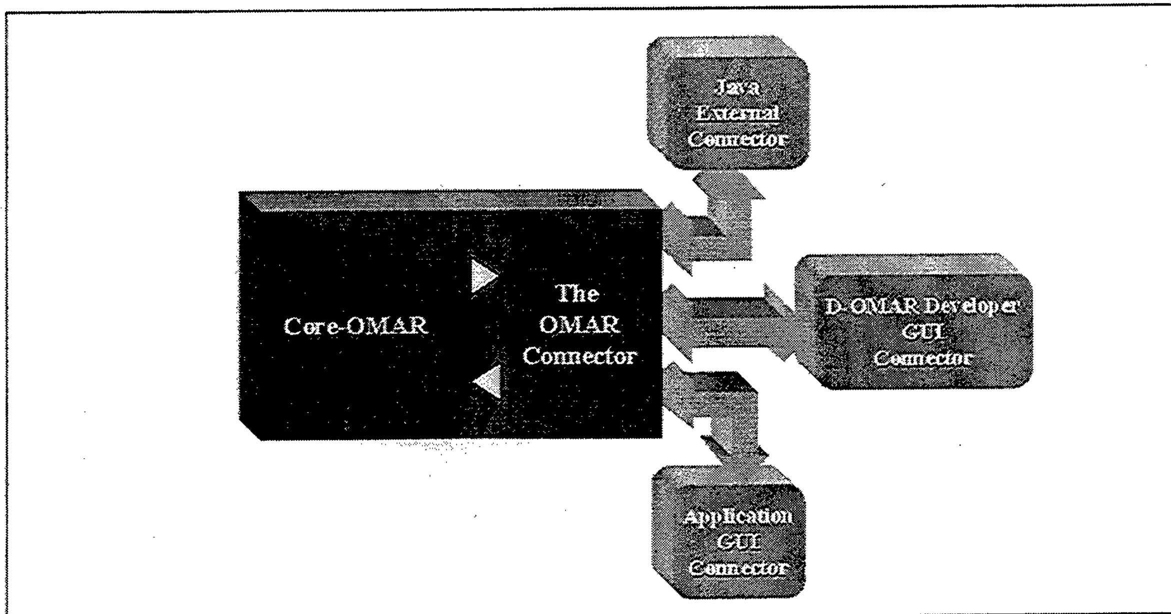


Figure 4. D-OMAR Connectivity for Core-OMAR

The first two uses of the OMAR Connector addressed interim D-OMAR development goals: validating the operation of Core-OMAR and providing an interim workplace for OMAR model development. The third and most important use of the OMAR Connector was to link a Core-OMAR server to D-OMAR clients and remote servers in a distributed computing environment.

2.3.2 The Lisp to Java Connection

The link between Core-OMAR and the OMAR Connector is the point at which the Lisp/non-Lisp language gap is addressed. Bridging the gap between Lisp and Java is a

two step process: using a socket to connect the process running Lisp with the process running Java and establishing a protocol for the transfer of data. A socket is a standard mechanism for inter-process communication available on virtually all platforms. Allegro Common Lisp has extensions that support sockets and there are Java classes that support sockets. Hence, there was a ready vehicle for linking the Core-OMAR server to the Java components of D-OMAR. Similar services are available in C++. The OMAR Connector can also run with C++ peers. The socket-based object-stream passes messages between Lisp and Java or C++ peers.

Given sockets as the means of connectivity between Lisp and Java, *serialization* is used to format an object into a byte stream suitable for the communication stream. The serialization is such that the byte stream representation can be reassembled as an object in the target language at the other end of the connection. The messages being passed are OMAR signals. At this point in time, all communication between the Lisp and the Java components in D-OMAR is wholly in terms of signals.

The basic structures underlying the serialization stream consists of an "object stream" which is responsible for the low-level socket-based inter-process communication, and a "parser" object, embedded in the object stream, which contains the detailed knowledge specific to the set of object types understood by the serialization stream. The object stream is responsible for handling the details of reading bytes from and writing bytes to an underlying physical stream and for the subsequent "conversion" of those bytes into objects in the host language.

Details of the specific serialization protocol used by an object stream are the responsibility of its internal parser object. The parser object knows how to "serialize" an out-going object into an appropriate sequence of bytes and how to read a corresponding series of bytes from which it can construct an in-coming object.

The Lisp end of the protocol consists of a socket-based server that also functions as a distribution center or "post-office" that handles the transmission and reception of events moving between Core-Omar and external system components. In particular the Lisp portion of the connection:

- handles the establishing of connections with the external client Java-connectors;
- receives in-coming events and distributes them to the appropriate section of the Core-OMAR API; and
- distributes out-going signals generated by the Core-OMAR to the appropriate external client-connection.

The Java or C++ end of the connection is implemented as a socket-based client that functions as a basic source and sink for messages in the target language.

2.4 The External Connector

The External Connector (the Java External Connector in Figure 4) is perhaps the most important single element in the D-OMAR architecture. On one side, it supports the network connection to Core-OMAR. On the other side, it may connect to another instance

of Core-OMAR or to a non-D-OMAR server. It is the element that enables multiple instances of Core-OMAR to operate as part of a larger distributed system. It may be a homogeneous system made up of just Core-OMAR servers or a heterogeneous system in which Core-OMAR servers interoperate with non-D-OMAR servers. The intermediate step in the connection between the local Core-OMAR server and the remote server is a middle-ware layer. It is the External Connector that has been designed so that it may be easily specialized to accommodate a broad range of middle-ware products.

The External Connector has a main body of code shared by the instantiations for each middle-ware implementation. This is the code that manages communication with Core-OMAR via the OMAR Connector. The External Connector is then specialized to operate first with the middle-ware layer and then with the particular remote server type. The middle-ware specific code encapsulates the Core-OMAR server's knowledge and requirements of the particular middle-ware type employed. It functions as a mediator or translator for information transmitted between Core-OMAR (in the form of external signals or messages) and the protocol dictated by the middle-ware layer (e.g., through method calls or event-like callbacks). The remote server could be another Core-OMAR node supported by an OMAR Connector or a heterogeneous non-D-OMAR server. For a heterogeneous node, the External Connector must include code to accommodate the particular requirements of the interface to that node.

2.4.1 The Core-OMAR External Connector

The first External Connector developed linked a homogeneous network of Core-OMAR servers using Java RMI as the middle-ware layer. Communication among Core-OMAR nodes is accomplished using SCORE language external signals. Procedures at the local node or at remote nodes subscribe to arriving signals using the *with-signal* or *asynch-wait* SCORE forms. The OMAR Connector receives out-going signals from Core-OMAR and passes them on to remote Core-OMAR nodes via the External Connector and the mirror image OMAR Connector on the receiving end. The signals then enter the remote simulator event streams for distribution to procedures that have subscribed to their signal type. This External Connector in combination with the Developer's GUI Connector is the essential building block dyad for D-OMAR. Together they integrated the new Core-OMAR server into a distributed computing environment for the first time. Core-OMAR and the Developer's GUI formed the new implementation of the OMAR model developer's programming environment.

2.4.2 The External Connector for the High Level Architecture

The Defense Modeling and Simulation Office (DMSO) has played a central role in distributed simulation for the Department of Defense. The High Level Architecture (HLA) is the latest in a series of distributed simulation architectures that started with SIMNET and evolved into Distributed Interactive Simulation (DIS). One of the important goals for D-OMAR is that it provide human performance models as computer generated forces that can operate in HLA simulation environments. The current implementation of HLA is Version 1.3 of the Real-time Infrastructure (RTI). Our goal was thus to provide a interface so that Core-OMAR would operate as an HLA player via the RTI.

Viewed from the perspective of D-OMAR, the HLA RTI is a middle-ware layer linking Core-OMAR servers and non-D-OMAR remote servers in a distributed computing environment. However, the RTI is also somewhat more than the typical middle-ware layer. It provides a distributed simulation framework and has an established set of services and protocols governing participant interactions with the RTI. Figure 5 lists the functional categories of the HLA services. When operating in an HLA environment, a cluster of Core-OMAR nodes are configured as an HLA Federate defined by a standard HLA Federation Object Model (FOM) and Simulation Object Model (SOM). With these definitions in place, HLA services are provided for:

- Control of the life cycle of the Federates,
- Naming of entities within the Federates,
- Governing event subscription and publishing among the Federates,
- The passing of events (e.g. attribute-change events) among the Federates, and
- Run-time synchronization.

The External Connector for integrating a D-OMAR Federate into an HLA simulation must fulfill the HLA dictated obligations in utilizing the RTI services. Hence, this connector is somewhat more complicated than the connector that supports a homogeneous D-OMAR system that links Core-OMAR servers. For example, using RTI services, a D-OMAR Federate is expected to utilize specifications for system-wide naming conventions, handle notification of changes in values for attributes of external entities, and publish changes to attribute values for internal entities.

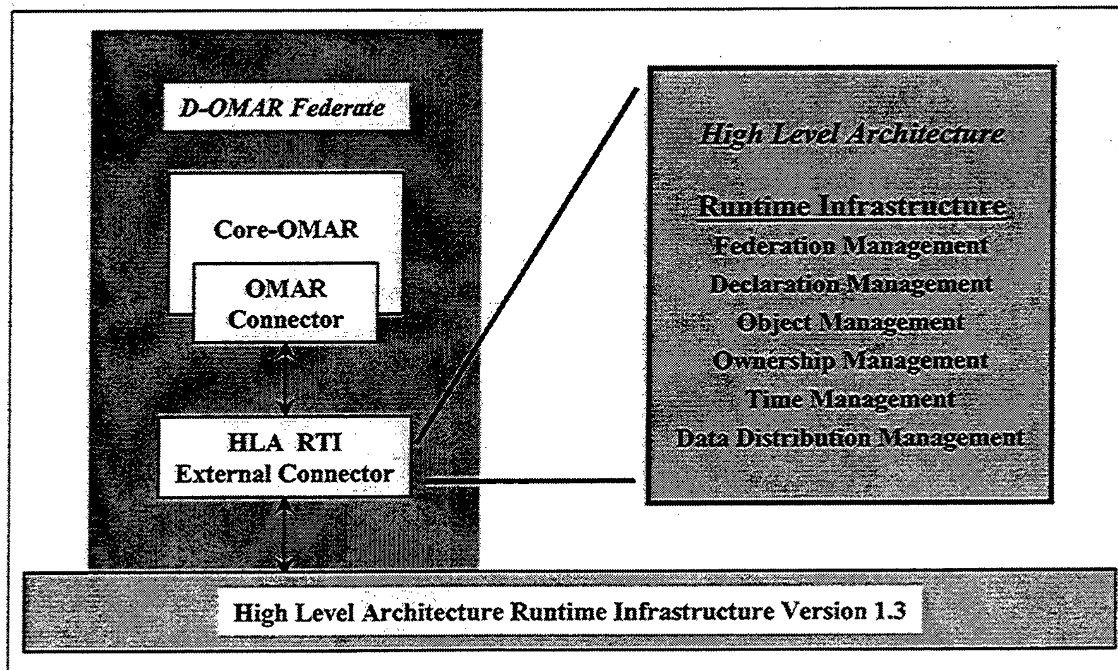


Figure 5 A D-OMAR Operating as an HLA Federate

The handling of attribute values forms the RTI publish/subscribe protocol and is built on the publish/subscribe strategy for D-OMAR signals. This is representative of the role of the External Connector for RTI operation. HLA not only provides the middle-ware layer linking the nodes of the network, it also specifies how interaction on the network are defined and executed. The External Connector incorporates the set of services necessary so that a Core-OMAR node may operate in an HLA simulation. D-OMAR running homogeneous Core-OMAR servers is a very different distributed simulation environment than is an RTI implementation of HLA. Even so, in making D-OMAR capable of executing in an HLA environment the only D-OMAR software module that had to be customized was the particular External Connector for RTI operation.

The current RTI implementation for HLA is in C++. In the early work with HLA, a C++ version of the OMAR Connector was used to integrate Core-OMAR servers for HLA operation. More recently, the RTI has been provided with a Java "cap." That is, RTI services are now available in Java as well as C++. The latest D-OMAR implementation for HLA is using the Java "cap" supported by the standard Java version of the OMAR Connector.

2.4.3 The External Connector for CORBA

The CORBA specification outlines perhaps the most complete set of middle-ware services available. The D-OMAR External Connector makes a rather modest set of demands on a subset of the specification that can be easily met by the offerings of any of a number of the commercial vendors. BBNT has ready and free access to the Visigenic Object Relation Broker (ORB) making that the obvious choice for use as the test-bed for CORBA development. The Java version of VisiBroker, the Java implementation of the Visigenic ORB, was used.

To demonstrate the use of CORBA as the middle-ware layer for D-OMAR it was necessary to implement D-OMAR signal passing among Core-OMAR servers. This was accomplished using the CORBA remote procedure call facility. On the Core-OMAR side, the CORBA version of the External Connector communicated with the OMAR Connector, on the other side it was specialized to communicate with the VisiBroker ORB.

2.5 The Developer's GUI Connector and the Developer's Interface

The D-OMAR Toolbar (Figure 6) is the entry point to the D-OMAR Developer's Interface (see Figure 2), the suite of software tools used in developing human performance models or agent-based systems. The Toolbar provides access to the D-OMAR components designed to assist the developer in managing large bodies of code, operating the simulator, and providing insight into agent operation. The D-OMAR Developer's Interface components accessible from the Toolbar include the Concept Editor, the Procedure Browse, the Simulation Control panel for controlling execution of the simulator, and the timeline displays of both current simulation run (Timelines in Figure 6) and data restored from previous runs (Analysis in Figure 6).

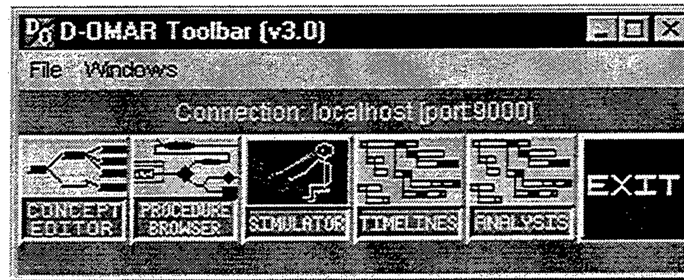


Figure 6 D-OMAR Toolbar

2.5.1 D-OMAR Developer's Interface Client-Server Operation

Each of the D-OMAR Developer's Interface components existed in OMAR. The interface code, previously written in Lisp and CLIM, has now been implemented in Java. The Developer's Interface modules operate a as client of a Core-OMAR server. The D-OMAR Developer's GUI Connector (Figure 4), also written in Java, provides the link between client and server. The Developer's GUI display components communicate with the Core-OMAR system via the Developer's GUI Connector and the OMAR Connector using standard D-OMAR signals. The OMAR Connector serves as the interface between Core-OMAR and the Developer's GUI Connector. It handles the transmission of data (including the translation of Lisp formatted data to and from Java formatted data) between Core-OMAR and the Developer's GUI Connector.

Like each of the connectors, the Developer's GUI Connector is specialized, in this case, to communicate with each of the Developer's Interface display components using the standard Java Listener (i.e., subscriber) and Source (i.e., publisher) mechanisms. Each display module registers itself as a Java Listener for the specific events classes that it is interested in receiving. The GUI Connector then routes data to and from the appropriate display modules.

The Core-OMAR server component registers itself with the GUI component as a Java Listener for the GUI component's server events through which the GUI Component communicates with the Core-OMAR system. To the external Java-based GUI clients, the event-server—the "face" that Core-OMAR presents to the external GUI components—is just a Java component. The inter-component communication is modeled on the standard Java Listener paradigm for Event generation and subscription.

2.5.2 The Concept Editor

The Concept Editor is a graphical editor designed to support the development and maintenance of the SFL concept hierarchy defining the objects of a model. A Concept Editor Network window (Figure 7) provides a view of a user-selected portion of the concept hierarchy. A Concept Editor Table window (Figure 8) provides a table of the slots and additional specifications for a particular concept. Data required by the Concept Editor, the SFL representation of concepts and roles for a given simulation, resides in Core-OMAR. The Concept Editor user interface code accesses the SFL data as required via the Developer's GUI Connector and the OMAR Connector. The presentation of the data in the Concept Editor Network and Table windows is generated in Java.

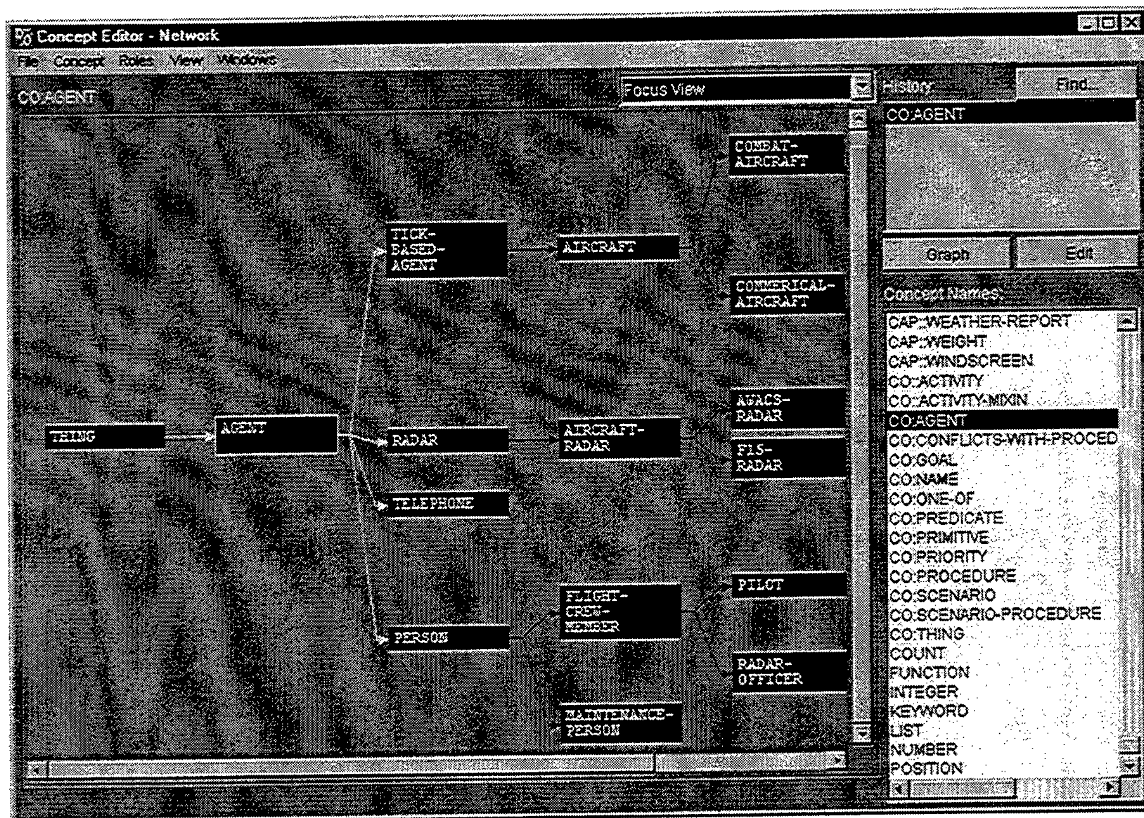


Figure 7 Concept Editor Network

The Network window, with concepts presented as the nodes of the graph, supports the maintenance of the concept hierarchy by providing operations for adding concepts to or deleting concepts from the network. The inheritance among concepts, the links to the parent and child concepts of a selected concept, may also be modified as required. A history pane is provided to make it easy to move back and forth between recently presented focus nodes. There are several options available in viewing the graph for the concept hierarchy. The shape of the nodes, and hence the layout of the graph, may be varied by choosing whether or not to insert a carriage in the concept name at the points at which they are hyphenated. In Figure 7, the concept names include carriage returns at the points at which they are hyphenated. An overview mode is also available for viewing the structure of large concept hierarchies.

The Concept Editor Table primarily supports the management of the slots for a concept. Slots may be added to or deleted from a concept, and the number and value restrictions, and default value for the slot may be set or revised. Separate views are provided to present all the slots for the concept or just the locally defined slots. The editor will flag inconsistent number or value restrictions. The "All Slots" view shows the restriction as resolved by the SFL *completion* algorithm and the "Local Slots" view shows the value as entered by the user. A locally entered number or value restriction that is flagged in red indicates that the local entry is either redundant or violates the restrictions inherited from parent concepts. The Concept Editor Table also allows the user to specify whether or not

an underlying class is to be defined for the concept and whether or not the concept symbol is to be exported.

Figure 8 Concept Editor Table

The D-OMAR User/Programmer Manual Version 3.0 is available on World Wide Web.[1] It provides detailed information on SFL and the use of the Concept Editor. It also provides detailed information on the SCORE language and the Procedure Browser, the Simulation Control Panel, and the Timeline and Analysis views into simulation runtime data as outlined in the following sections.

2.5.3 The Procedure Browser

SCORE is the procedural language in D-OMAR. Code is developed using an EMACS editor, a text editor that Franz has linked closely to their Lisp runtime environment. The Procedure Browser provides several graphical views of the code designed to support model development and debugging. The views are designed, in part, to provide insight into the large-scale structure of the code.

One pair of views presents the calling sequence relations among procedures. The first of the pair shows the sub-procedure calls (Figure 9). The second shows the inverse of the first, the "who calls" view. As in the Concept Editor the shape of the nodes, and hence the appearance of the graph, is determined by whether or not the procedure names have a carriage return inserted at the point at which they are hyphenated. Figure 9 is an example in which carriage returns are not used. It is fast and easy to move back and forth between the two views to find the one suitable for a particular graph. The utility of one view or the other varies depending on the structure of the particular graph and the region of interest within the graph.

The second pair of Procedure Browser views supports the use of *signals*, the publish/subscribe capability of the SCORE language. The SCORE forms for publishing signals are *signal-event* and *signal-event-external*, those for subscribing to events are *asynch-wait, with-signal* and *with-multiple-signals*. One view presents a graph that traces the signals generated by a selected procedure and the signals generated by the procedures subsequently contacted. The second view presents a trace of the signals subscribed to by the procedure.

[1] http://...

The complexity of the code for a model can lead to large graphs. An overview form of the graph is provided for each viewing option to assist the user in managing the complexity. Figure 10 provides an example of an overview graph of the signals generated by a procedure. As in each of the other views, additional information on the nodes of the overview graph may be obtained through mouse gestures.

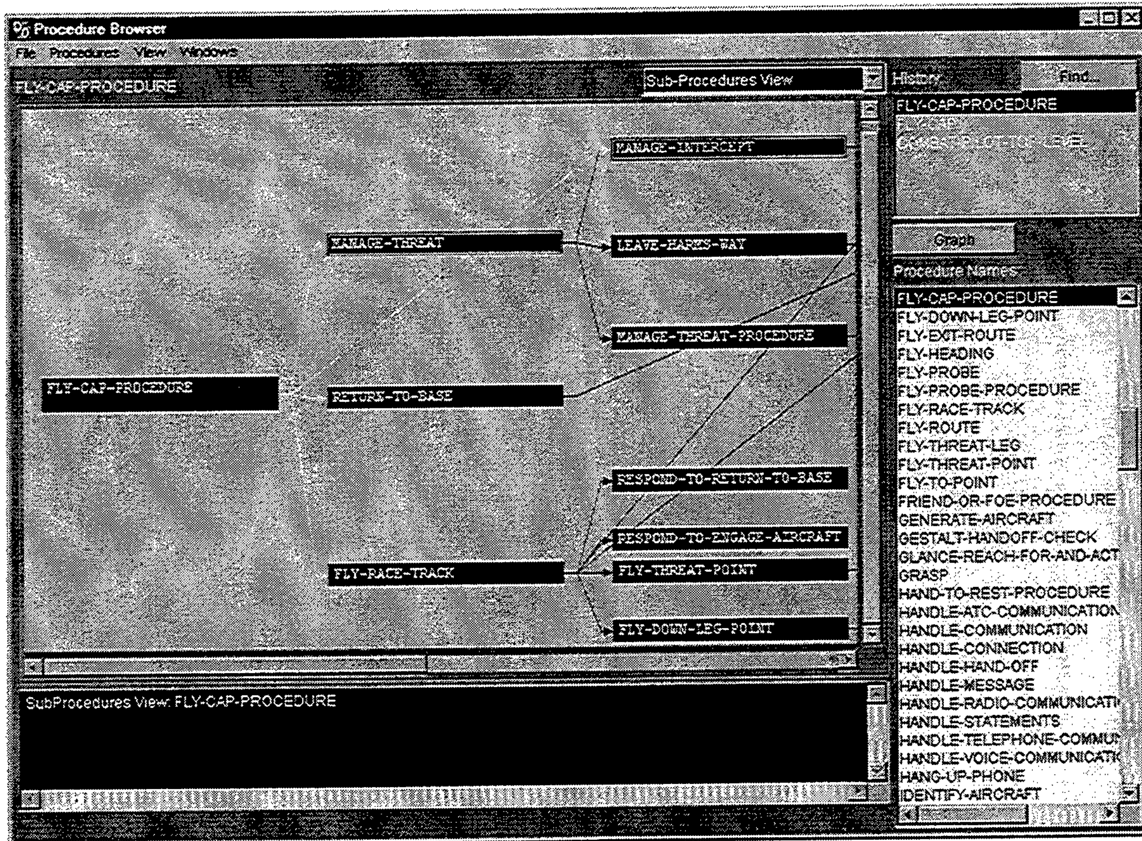


Figure 9 Procedure Browser Sub-procedures View

Underlying the original version of Procedure Browser is the CLIM-based PIASTRE (Piastre Is A STRucture Editor) system, designed to be used for the graphical, structural representation, and editing of code. While the underlying features of PIASTRE support structure editing, it is used in the OMAR system as a browsing, rather than editing tool. Because the Procedure Browser and PIASTRE represented a significant amount of code that would not translate directly or easily into Java the decision was made to use as much as possible of the underlying, original Lisp code in direct support of the planned Java-based user interface.

Towards this goal a "draw stream" protocol was designed and implemented. The protocol works as follows:

1. The stream that would normally be handed to the CLIM-based drawing-code in the Procedure Browser is replaced with a Draw-Stream object.

2. It is the responsibility of the Draw-Stream object to "serialize" the sequence of draw-commands called on the Draw-Stream (e.g., *draw-line*, *draw-rectangle*, *set-color*) into a sequence of bytes.
3. This sequence of bytes is then transmitted to the display gadget, where it is decoded and displayed locally in Java.

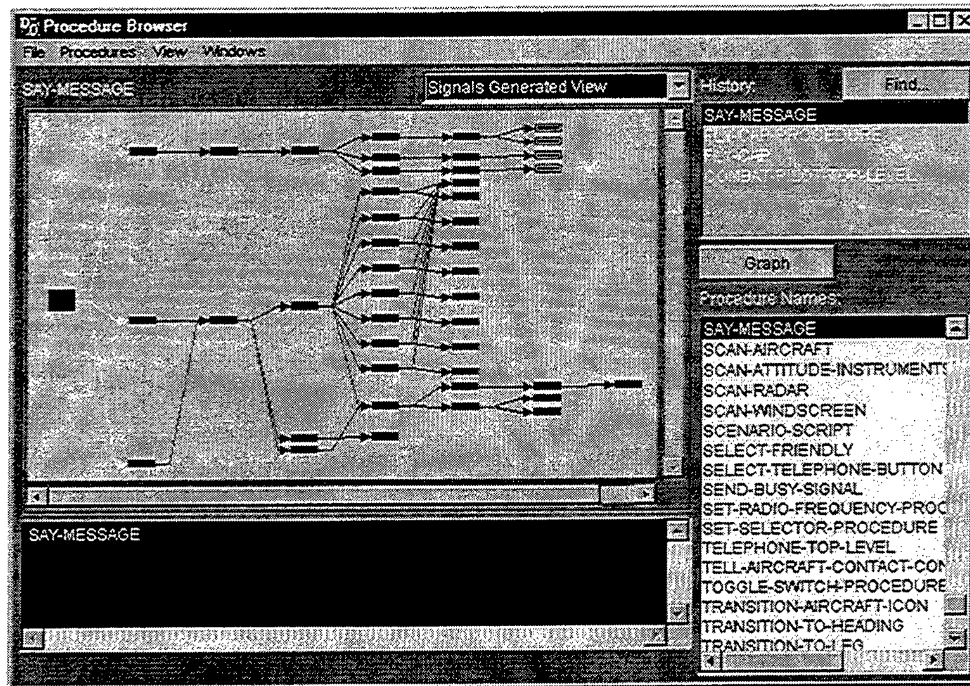


Figure 10 Procedure Browser Signals Generated Overview

On the input side, the gestures to the Java display (e.g., mouse-clicks) are passed back to the Lisp-side and the Procedure Browser code can, for example, determine which underlying Lisp object appears at that screen location. It then provides an appropriate response, for example, generating the request for the description of a screen object. In D-OMAR, Draw-Stream has been implemented in a batch-like approach in which the sequence of bytes is first captured in an array, and the array is transmitted as an event to the Java display gadget. Hence, in the Procedure Browser all of the "drawing," as well as layout, and data-manipulation are handled in Lisp, while the "display-rendering" is handled in Java.

There were several advantages to the Draw-Stream architecture:

1. Converting existing code proceeded very quickly. Once the Lisp-side Draw-Stream was in place and the code for the Java-side display gadget was finished, all that was involved was replacing the standard stream object (to which the display-code draws) with the Draw-Stream object.
2. The majority of the existing Lisp code was reused.

3. There need only be one Lisp-side Draw-Stream class and one Java-side display gadget class. (That is, the Draw-Stream protocol is intended to behave as true stream and, as such, to be general; it is not hardwired to a specific application.)
4. Once the Draw-Stream protocol was in place, the graphics/display system (CLIM) could be removed from the Lisp application. This was particularly important because CLIM was the only potential bottleneck in the path to platform independence for D-OMAR.

2.5.4 The Simulator Control Panel

The Simulator Control Panel provides control over the operation of the D-OMAR simulator. Once scenarios have been loaded, usually the last step in bringing up Core-OMAR at a server node, the Control Panel may be used to select a scenario to run. It may then used to initiate, run, pause, and resume scenario execution. A trace pane provides information on the runtime events of the simulation. The user has control of the agents for which the information is presented, as well as the types of events presented in the trace. The event data from a simulation run may also be saved to disk for later use. Application developers are responsible for providing event types yielding information relevant to the operation of a scenario. Very detailed events recording of the execution of SCORE code is available. During application development, event tracing, used in conjunction with the Task and Event Timeline displays can be an important tool to support model debugging and program verification.

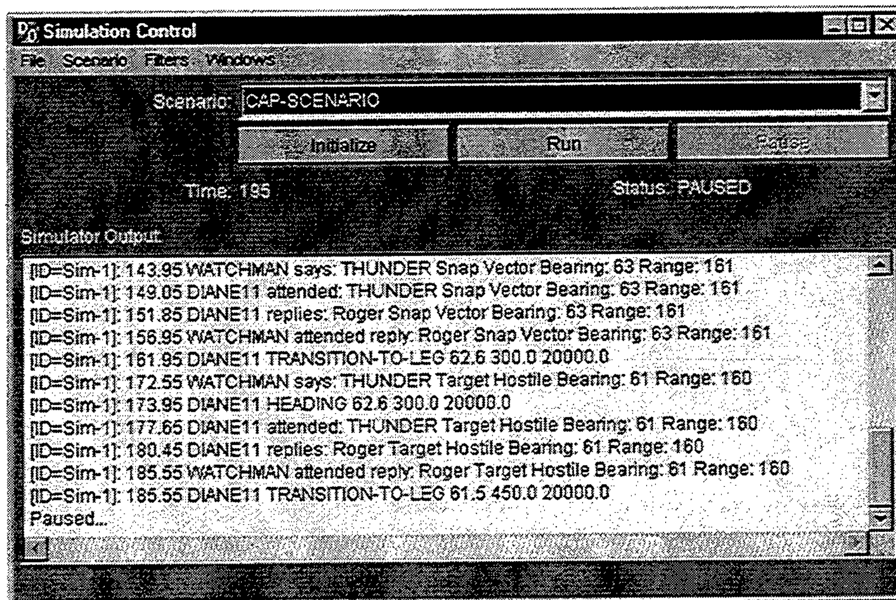


Figure 11 Simulator Control Panel

2.5.5 The Timeline and Analysis Data

The D-OMAR Toolbar includes separate entry points for accessing Timeline and Analysis data. They each provide access to the same presentation types for the data, the Timeline entry point working from runtime data and the Analysis entry point working

from data restored from an earlier simulation run. The distinction is made to draw attention to the fact that the full functionality of the simulator is available to continue operation with respect to Timeline or runtime data, but that the simulator can not be invoked for the case of Analysis data restored from an earlier run. The process of saving data from a simulation run does not save sufficient information to allow the run to be continued at a later time.

The Timeline and Analysis displays allow the model developer or analyst to precisely access the detailed data on the execution of their models during a simulation run. The displays have proven to be an essential resource for gaining insight into the operation of and assessing the validity of the models developed to date.

Two timeline displays are provided: a Task Timeline and an Event Timeline. The agents included in the display and the time span covered for each of the displays may be controlled by the user. Mouse gestures may be used to adjust the window forward or backward in time. The Task Timeline (Figure 12) provides detailed information on the goals and procedures that an agent is executing. Each horizontal bar of the Gantt-style chart provides information on one of the agent's goals or procedures. Information includes the name, start time, and priority for the goal or procedure. For a completed goal or procedure, the stop time and success or failure of the procedure is noted. The calling goal or procedure and its priority are also identified.

The Event Timeline (Figure 13) display presents event data by agent. A number of event types are built into the SCORE language and can always be made available for display. A subset of these event types provides the data necessary to support the information requirements of the Task Timeline. The SCORE language also includes a *defevent* form that may be used add new event types important to the scenario being developed. The *record-event* form is used to generate the runtime data that is recorded to support this display. The analyst can select the agents to be included in the display, the time frame for the display, and the event types to be included. Signal events are an important class of events that track SCORE signal propagation. The Event Timeline for signal events enables the developer to track this aspect of agent behavior.

Verbal communication is an important activity in virtually all of the scenarios developed in OMAR and D-OMAR, hence communication events play an important role in the analysis of many simulation runs. The Event Timeline shown in Figure 13 presents the communication events for two of the players in the CAP Scenario discussed in Section 4. It traces the radio communication between the radar office and the pilot of the interceptor aircraft during the conduct of the intercept. The display graphically summarizes the interaction between two of the key players in the scenario. Filters for the agents to display, the time frame, and event types to display are used to interactively isolate the data of interest.

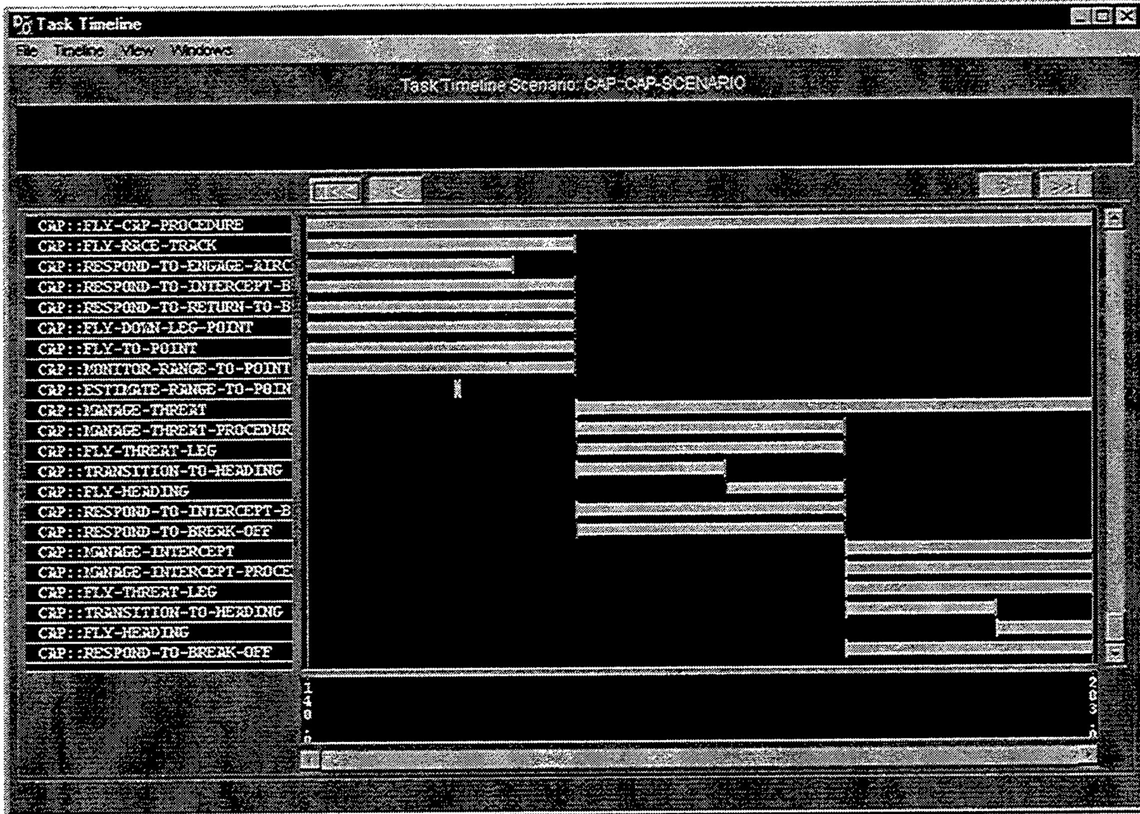


Figure 12 Task Timeline Display

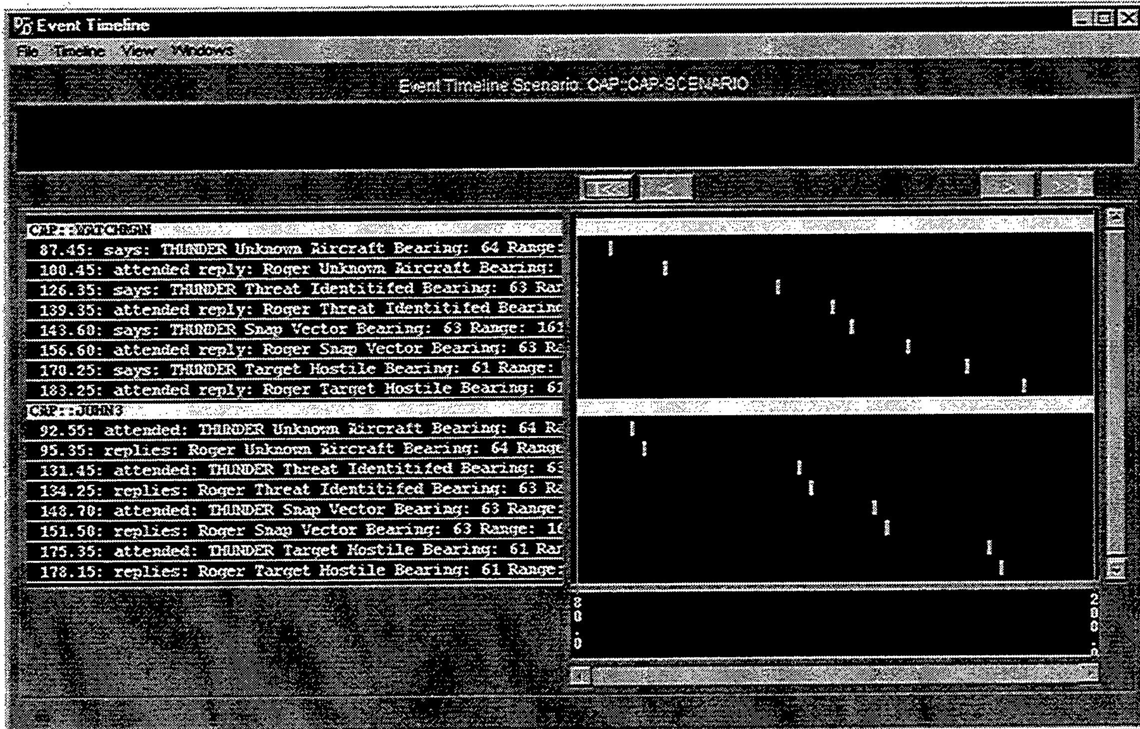


Figure 13 Event Timeline Display

2.6 The Application GUI Connector

D-OMAR may be used as a distributed simulation environment populated by human performance models or as the basis for developing an agent-based system. In either case, the application will require one or more application interface sites. In the D-OMAR distributed computing environment, a Core-OMAR node will act as a server supporting the application interface. The client code supporting the interface may be developed in whatever language is appropriate for the application. The Application GUI Connector is designed to serve as the network connection between Core-OMAR and the display code for the application interface. In the CAP scenario, the radar display (see Figure 15) is an example of the use of the Application GUI Connector.

The architecture for the Application GUI Connector is structurally equivalent to that of the Developer's GUI Connector. The core of the Connector transmits events between Core-OMAR and the modules that drive the application interface. The Connector is then specialized to meet the requirements of the events particular to the application interface. The physical display may operate locally at the same node as the Core-OMAR server or it may operate remotely at any node of the network.

3. D-OMAR Agents and Agent-based Systems

The original target application for OMAR was the development of human performance model. D-OMAR provides a new framework in which to pursue the same goals. The OMAR representation languages were designed to facilitate the modeling of the multi-tasking behaviors of team members interacting with complex equipment in pursuing collaborative enterprises. The representation languages that support the modeling of human-like agency can also be used to develop models of agency necessary for the agents that will be required in agent-based systems.

Recent research on agent-based systems has focused on the communication and cooperation among agents operating in distributed computing environments and on communications between system users and the agents that support them in the execution of their tasks. The new capabilities being demonstrated in agent-based systems reside in their potential to support widely dispersed users in their collaborative efforts to effectively manage evolving situations. This represents a new application that the development of D-OMAR has been designed to support.

3.1 Agency in Human Models and in Agent-based Systems

Agents in OMAR were originally envisioned as the basis for developing human performance models. While D-OMAR was designed to address the same human performance modeling goals as OMAR, it has the additional goal of providing a framework suitable for the development of agent-based application systems. The representation languages, principally the procedure language SCORE, designed to support the modeling of human performance is seen as having the attributes necessary to develop agent behaviors suitable for the agents that play an active role in providing services to human operators of complex systems.

3.1.1 Agency in Human Performance Models

In human performance models that have been developed in OMAR and D-OMAR (Deutsch, 1998; Deutsch & Adams, 1995), each of the human players that is modeled typically has several tasks in process and interruptions are a common occurrence. Like human players, the models are capable of proactive goal-directed behaviors, while at the same time, they must be capable of responding properly to the sequence of events evolving in complex situations. In supporting these behaviors, the models must exhibit teamwork skills and communication skills. They draw on the capabilities of other players, using them as resources, and they must be capable of responding to requests made by the other players in the environment. They must appropriately shift attention from one task to another in pursuing multiple goals while sensitive to an often complex chain of events.

The models are built from a foundation of individual perceptual, cognitive, and motor skills that are the resources that may be drawn upon to accomplish the subtasks of a task in process. There is a necessary layer of "basic person" skills and there must be a range of domain specific skills for the environment in which the models are expected to operate. There is an on-going interaction among thoughtful cognitive and automatic perceptual, cognitive, and motor performance elements in the execution of each subtask. The transition from one task to another can be the result of a thoughtful decision process or the outcome of an automatic behavior. Moreover, there are bounds on what can be done and on what can be done concurrently.

3.1.2 Agency in Agent-based Systems

Intelligent agents, software agents, or just agents: however they are designated, they immediately provoke challenging questions. What are agents? How are they different from programs? Asked in a more modern version: How are they different from objects? Or, how are they different from objects in a distributed object computing environment? Franklin and Graesser (1996) provide a good survey of the attempts by the developers of a number of agent-based systems to address these questions. One of the more complete definitions that they cite is that of Wooldridge and Jennings (1995). Jennings and Wooldridge (1996) provided a slightly amended version of their "key hallmarks of agenthood:

- *Autonomy:* agents should be able to perform the majority of their problem solving tasks without the direct intervention of humans or other agents, and they should have a degree of control over their own actions and their own internal state.
- *Social ability:* agents should be able to interact, when they deem appropriate, with other software agents and humans in order to complete their own problem solving and to help others with their activities where appropriate.
- *Responsiveness:* agents should perceive their environment (which may be the physical world, a user, a collection of agents, the Internet, etc.) and respond in a timely fashion to changes which occur in it.
- *Proactiveness:* agents should not simply act in response to their environment, they should be able to exhibit opportunistic, goal-directed behaviors and take the initiative where appropriate."

Jennings and Wooldridge (1996) then attempt to differentiate software agents from object-oriented systems, artificial intelligence, and distributed computing by citing the high-level tasks that are delegated to agents and the degree of autonomy that the agents are granted in carrying out their tasks. They also point to the dynamically changing environments in which agents operate. We prefer a viewpoint that downplays the distinction that Jennings and Wooldridge attempt to make. The development of agents owes much to earlier work in object-oriented programming and artificial intelligence, and agents will operate very naturally in distributed object system environments with their dynamically changing environments.

3.2 Agents in D-OMAR

D-OMAR provides a broad range of software tools for developing agents, either as human performance models or the agents for agent-based systems. Section 2.5 provided an overview of the D-OMAR Developer's Interface that includes the Concept Editor, the Procedure Browser, the Simulation Control panel, and the Timeline and Analysis displays of simulation data. This section provides an overview of the features of the procedural language, the Simulation Core (SCORE) language, available for developing agent behaviors.

3.2.1 Agent-Agent Communication

The principal form of communication among agents in D-OMAR, those that reside at the local site and those that reside at remote sites, is *signal* passing. In D-OMAR, an agent may, via one or more of its running procedures, subscribe to a signal. That is, the agent makes a request to be notified of the occurrence of a particular event announced as a broadcast signal. The signal is a list with the first element defining the signal type. The subsequent elements of the list include data elements that provide information to describe the event. The SCORE form, *signal-event,* is used to broadcast a signal locally. *Signal-event-external* is used to broadcast a signal to remote Core-OMAR servers as well as the local server.

The SCORE language provides three forms for subscribing to signals. In each case, the compiler generates a closure for the function to be executed when the agent has elected to process the signal. At any given time, there may be one or more agents, each with one or more procedures that subscribe to the occurrence of any given signal. It may also be the case that a signal occurs that is not attended by any agent.

The *asynch-wait* form is the simplest form used to subscribe to a signal. It specifies the signal type and the argument values expected. Upon subscribing to a signal, that thread of the agent's processing is suspended until the occurrence of a signal with an exact match on the signal type and argument values. The *with-signal* form provides more flexibility in selecting a signal to process. In using this form, an agent gains access to each signal of the specified type and may then examine any or all of the signal's arguments for specific values before deciding to process the signal. If through the use of the test clause, the agent elects not to process the signal, the *with-signal* form remains in effect awaiting the next occurrence of the specified signal type.

The third subscribe form, *with-multiple-signals*, as its name suggests, provides the capability for processing several signals that may not be strictly ordered in time. As with the form *with-signal*, a test clause may be specified for each signal specified in the form. Time-out conditions may be associated with each signal and the occurrence of a signal may be tagged as optional within the form. Hence, two or more signals may be required to occur in a given time frame if further processing is to take place in a given procedure. The optional signal may provide additional information that could, but is not required to support subsequent processing. Having processed a signal, a procedure must execute another subscribe form if it is to be responsive to the next occurrence of that signal type.

There are distinct differences between subscribing to a signal and making a subroutine call or message passing in an object-oriented environment. A subroutine call or a message sent to an object invokes a single respondent; a signal may be processed by any number of procedures active on one or more agents, or perhaps, none at all. A called subroutine always returns control to its caller; an agent's procedure that processes a signal is operating in a thread independent of the source of the signal. Agents process a signal in their own code thread and do not return control or values to the initiator of the signal. Unless a particular effort is made to make it known, the initiator of a signal will not know which procedures acted in response to the signal.

3.2.2 Proactive and Reactive Behaviors

D-OMAR agents are capable of proactive behaviors, that is, they pursue objectives, and they maintain an agenda of things to do in order to accomplish their objectives. The objectives are expressed as *goals* and the actions to accomplish those goals are organized in a *plan*. The plan to accomplish a goal may partition the goal into sub-goals. Actions to accomplish the goal or its sub-goals are expressed as *procedures*. Goals and procedures are typically defined to operate in an environment in which failures are anticipated and alternate paths to successful execution are prepared. In building a good plan, the paths that might lead to failure are studied and triggers are identified to anticipate situations leading to the failure of a portion of a plan. Multiple plans may need to be available to achieve a goal—each may be available to operate from a different set of initial conditions. Lastly, in a complex environment a plan in process to accomplish a goal may succeed or fail independently of the actions underway to achieve the goal

Within D-OMAR, the SCORE language is used to define the goals, plans, and procedures of D-OMAR agents. Goals and procedures are expressed using the *defgoal* and *defproc* forms. The plan for a goal is expressed through the sub-goals and procedures that the goal invokes through subroutine calls. The called subroutine may be another goal or a procedure. The *defgoal* form includes an argument provided for specifying the initial conditions required by the plan for the goal. It also provides arguments to specify the success or failure conditions that might prematurely terminate actions to address a goal. The *success-conditions* defining the successful outcome of a goal may be used as the basis of selecting a goal to achieve that objective.

Reactive behaviors addressing a single event, an isolated signal, are quite straightforward. The development of agents to address situations that are more complex is made possible by two important D-OMAR capabilities. First, SCORE language forms enable a

framework to be established in which events are processed in the context of an agent's active goals. Agent behaviors are seldom designed to be purely reactive. Even in simple cases, the occurrence of a particular signal relevant to an agent will need to be addressed by that agent in the context of some goal to be achieved. The event might mark the successful achievement of a sub-goal or it might mark a condition that indicates the failure of one tactic to achieve the goal and the need to choose another approach. Agent behaviors are often required to be a combination of proactive and reactive components. The processing of a signal marking an event plays an important role in the plans developed for the goals of D-OMAR agents.

The second source of complexity that may be addressed by SCORE language forms is the management of agent response to particular event sequences. Agents must deal with sequences of events that may or may not be well ordered. The response of a D-OMAR agent to evolving situations is developed through the processing of the chains of events made available to the agent as signals. An agent's sensitivity to a given signal is set up by an agent's procedure subscribing to that signal. The response to that signal may then be to set up the same procedure or another procedure in anticipation of the recurrence of the same signal, another particular signal, or a new combination of the two. A signal processed in one situation may be ignored or treated differently in another situation. An agent might well have two evaluation processes going on concurrently, each with the capability to shut down the other and determine the response, a winner-take-all strategy.

3.2.3 Multi-task Performance

D-OMAR agents are proactive as is expressed in their plans and procedures to achieve their goals. Some of an agent's goals may be independent of one another while others may be sub-goals within a plan with explicit dependencies among them. The agents can have several active high level goals, that is, they are capable of multi-tasking. One dimension of the dependencies is the order in which sub-goals or procedures within a plan execute. They may be required to execute sequentially with one procedure strictly following another or they may be allowed to execute in parallel. In D-OMAR, as a modeling environment, the OMAR simulator provides the emulation of parallel procedure execution. In a real world applications, execution of the D-OMAR simulator in real-time enables D-OMAR agents to pursue sub-goals in parallel. Termination conditions for sub-goals being pursued in parallel are important. In the winner-take-all case, the procedure that completes first has met the local objective and the concurrent procedures pursuing the same objective should be shut down. In other cases, all procedures operating in parallel may be required to complete before subsequent procedures in the thread can be initiated.

Several key SCORE forms provide the capability to define multi-tasking behaviors for D-OMAR agents. Parallel execution of an agent's goals and procedures are specified using the forms *join, race,* and *satisfy*. Procedure invocations enclosed in a *join* form must each execute to completion before subsequent forms are evaluated. The winner-take-all case is supported by the *race* form—the first procedure to complete causes all of it siblings to be terminated and initiates execution of subsequent forms. The *satisfy* form is a slight

variation on the *race* form. Procedures executing in parallel within this form continue to execute until the first of them completes successfully.

Conflicts among the goals and procedures for an agent are an essential aspect of multi-tasking behaviors. The execution of one procedure may preclude the execution of another related procedure. Priorities for goals and procedures, the basis for resolving these conflicts, are computed dynamically. The form of the priority calculation and its input parameters are specified by the developers of the agent's goals and procedures. Importantly, D-OMAR provides support for the basic fact that not all procedures conflict with all other procedures. Goals and procedure are class members and one aspect of class membership covers procedure conflicts. The SCORE forms for defining goals and procedures, *defgoal* and *defproc* specify the SFL concept membership for the goal or procedure, and this in turn specifies the class membership. Procedures within a class may conflict with one another, or procedures in one class may conflict with procedures in one or more related classes. Procedures with potential conflicts include the *conflicts-with-mixin* in their definition. The *conflicts-with?* method, a method on the *conflict-with-mixin,* is the basis for resolving the priority-based suspension and resumption of conflicting goals and procedures. When a new procedure vying to execute has class membership such that it conflicts with an executing procedure, the priorities for both procedures are computed. If the new procedure has sufficient priority, it will begin execution and the executing procedure will be suspended. If it lacks sufficient priority, it will be suspended until the executing procedure completes or until the balance in priorities shifts in its favor. The SCORE form *on-suspend* can be used to define the "clean up" steps that may be required by a procedure that is about to be suspended.

3.2.4 Invoking a Goal or a Procedure on an Agent

The invocation of a goal or a procedure takes the form of a subroutine call with the name of the goal or procedure as the first element of the call and the arguments for the called procedure specified as keyword-value pairs. The arguments to a goal or procedure, defined in the *defgoal* or *defproc* form, are slots of the concept that is the basis for the goal or procedure definition. Agents typically invoke sub-goals or sub-procedures with themselves as the agent, but the agent for the procedure is a legitimate argument, hence the invocation can be on any local agent. The calling procedure can either await the completion of the sub-procedure or *spawn* the sub-procedure in a separate thread of execution. In the former case, the object returned on completion is the procedure object for the called sub-procedure. A reader macro (i.e., !r) is available to obtain the "result" of the procedure rather than the procedure itself. When a procedure spawns a sub-procedure launching that procedure in a separate thread, it immediately continues execution of the next form in sequence. By default, the called procedure is launched with the priority of the calling procedure. The priority argument may be used to set the priority of the called procedure to a value other than the default value.

On completion, goals and procedure may either succeed or fail. The default procedure outcome is to succeed. A procedure may declare itself to have failed, or it may force the termination of another goal or procedure and attribute success or failure to it. The SCORE *succeed* and *fail* forms, when used without an argument, cause the executing

procedure to terminate in success or failure respectively. When used with an argument, the argument specifies the procedure that is to be terminated. A procedure forced to terminate, whether successfully or unsuccessfully, may need the opportunity to "clean up" before it actually completes execution. The *on-succeed* and *on-fail* forms are available to specify code to be executed when a procedure succeeds or fails. The behavior of the procedure that invoked the procedure that failed is important. The default behavior of a failed sub-procedure is to cause the failure of the calling procedure. This pattern will continue up through the sequence of calling procedures unless trapped. The SCORE form, *fail-handler,* may be used by the calling procedure to trap and handle the failure of a sub-procedure. The fail handler will prevent the default behavior, the upward propagation of the failure through the calling sequence.

3.3 D-OMAR Agent-based Systems

Agent-based systems are expected to be called upon to operate in large pre-existing and new distributed computing environments. Moreover, it is reasonable to expect that there may be more than one agent-based system, each of which acts as the provider of new capabilities within the existing framework. D-OMAR agents might well be called upon to interact with the agents of another agent-based system. This section briefly outlines the role that D-OMAR can play as an agent-based system operating in such an environment.

3.3.1 Homogeneous and Heterogeneous Agents

The architecture for D-OMAR addresses agent heterogeneity on three levels: syntactic, control, and semantic heterogeneity as identified by Bird (1993). From its earliest days, OMAR knowledge representation (Freeman, 1997; Deutsch, Adams, Abrett, Cramer, & Feehrer, 1993) was based on a frame language, the Simple Frame Language (SFL), and a procedural language, the Simulation Core language (SCORE). More recently, a rule-based language has been included in OMAR. At the syntactic and control levels, D-OMAR will continue to be an open system allowing the inclusion of new knowledge representation schemes, reasoning mechanisms, and communication languages.

At the semantic level, there are reasons for both homogeneity and heterogeneity. The case for homogeneity is strongest when addressing communication among D-OMAR agents. D-OMAR agents should have a common semantics as a basis for their communication. This is not enforced by the architecture, but is encouraged as good programming practice. The argument for heterogeneity is based on the pragmatics of the growth in agent-based systems. It is highly unlikely that D-OMAR will be the only source of agent-based software in any given large system. Such systems are likely to be populated by agents from several sources (see Figure 14) and while CORBA may address syntactic heterogeneity, semantic heterogeneity will have to be addressed by D-OMAR agents and non-OMAR agents that they communicate with. The functionality provided by additional agents in a system should be viewed as a resource, while semantic heterogeneity is an obstacle to be overcome in taking advantage of that functionality.

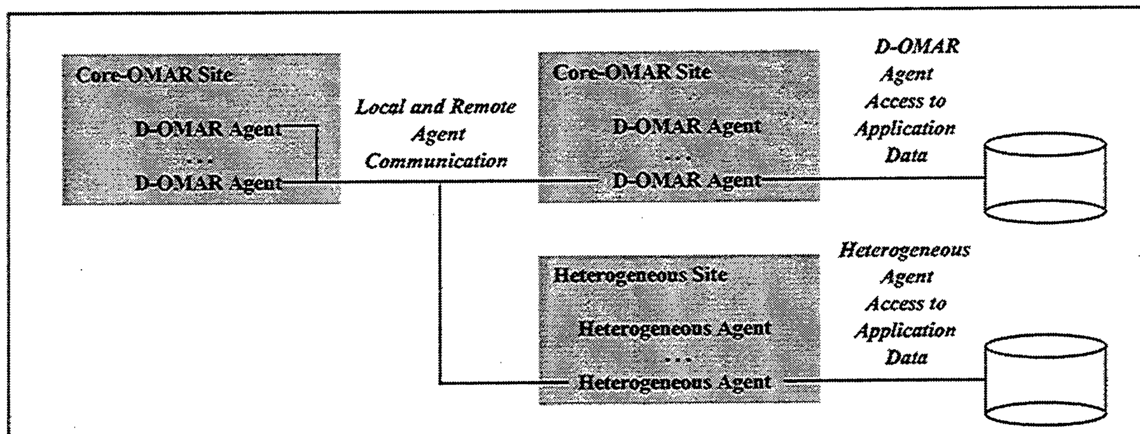


Figure 14 D-OMAR and Heterogeneous Agents

3.3.2 Agent Access to Application Data

Access to application data can be expected to take several forms. When D-OMAR is a new addition to the existing system, the architectural requirements for D-OMAR will be dictated to some extent by the architecture of the particular application system. Viewed in the most forward looking perspective, D-OMAR will be operating in a distributed object computing environment. A CORBA environment is a very likely possibility. At the other end of the spectrum, data required by a D-OMAR agent might reside in a legacy database, most likely as part of a large legacy system.

Over the last several years, there have been a number of systems built around pre-existing legacy system components. It has typically been the case that the components have included legacy databases and large application software subsystems with individual components implemented in different computer languages. These systems are often referred to as *federated* systems. The middle-ware to provide interoperability is usually CORBA. Hence, CORBA would provide the middle-ware layer that D-OMAR would use to operate in one of these federated environments. At the architectural level there are not likely to be federated systems demands that can not be met by the D-OMAR architecture. Indeed, at the implementation level, some of the software building blocks of the current systems may well be available for reuse.

As the legacy systems are replaced by new systems based on distributed object architectures, these architectures can be expected to closely resemble the D-OMAR architecture. Such systems may rely heavily on distributed system message passing and D-OMAR agents will access objects and services in these systems via message passing. The distributed systems may also include event services that are very similar to the signal-based communication that D-OMAR agents use for their own inter-agent communication. D-OMAR supports signal-based communication and can support message passing for operation in a distributed object system environment.

3.3.3 Agent Mobility

Several issues complicate agent mobility. Some of the easier problems are being addressed. There are solutions to the problem of sending messages to agents that have moved on to another site, and self-initiated moving is available for simple agents. One argument for utility in agent mobility is based on sending an agent to a large data source, thereby eliminating extensive remote queries. Nevertheless, it is easy to see that there may be cases in which an agent is engaged in a complex on-going effort at a local site and at the same time needs to extensively probe a large database at a remote location. This is an instance in which the remote data access might best be delegated to another agent, while current local activities continue to be pursued. Agent mobility may turn out to be useful only in single-task agents. An agent seeking a resource or perhaps the best "price" for a resource that is available at more than one site is an example.

A complex multi-tasking agent is more likely to rely on communicating with a remote agent to request extensive services at a remote site. The multi-tasking agent may have on-going activities at the local site making it impractical to travel to a remote site to obtain information from a database. The data retrieval can be delegated to a remote agent while the local agent carries on its concurrent tasks.

A second complicating issue is the question of how the mobile agent's current state is to be reestablished at the new site. Agent mobility today is accomplished either by using simple scripting languages (Gray, 1996) or by agents that reestablish operations based on minimal pre-move data. It is not yet clear just how one might reestablish the full computational state of a complex agent at a new site. D-OMAR does not currently support agent mobility, but many of the simpler aspects of mobility can be readily developed using services currently in place.

4. The Combat Air Patrol Demonstration

By their very nature, distributed systems are far more complex than systems that operate on a single machine. The systems themselves are more difficult to design and implement, and the applications built for distributed environments are more difficult to develop. A Combat Air Patrol (CAP) scenario was selected as a test case to use to help validate the operation of D-OMAR and provide a demonstration of D-OMAR in operation. The scenario operates in the standard D-OMAR configuration using the Java External Connector to link Core-OMAR nodes via Java RMI, it works using CORBA as the middle-ware layer, and it works in an HLA environment using RTI Version 1.3.

4.1 CAP Scenario Agents

The principal non-human elements of the CAP scenario are modeled as OMAR agents. They included the scenario aircraft—an AWACS, and friendly and bogey fighter aircraft. The AWACS had a much-simplified flight deck and a single radar station. The model was derived from earlier models of commercial aircraft used in air traffic control scenarios. The interceptor aircraft each also have a simplified flight deck that includes a radar subsystem. Like the aircraft, the radar subsystems are each modeled as agents. The

AWACS radar scanned 360 degrees and has a longer range than the forward-looking radar on the interceptor aircraft.

The CAP scenario required human performance models for a number of aircraft crew roles. The AWACS aircraft has a pilot, copilot, and radar officer. The friendly interceptor and bogey aircraft each have a pilot. These models were developed as derivatives of the human performance models used in an earlier simulation of a commercial air traffic control environment. The AWACS pilot and copilot models are based on captain and first officer models for commercial aircraft. Like all of the models, they each have a set of generic "basic person" skills and a set of specialized skills appropriate to the tasks that they are required to execute. Domain knowledge and hand-eye coordination are essential to workplace operations. Teamwork and verbal communication skills are necessary for coordinating and conducting the successful completion of multi-person tasks.

The pilot and co-pilot manage their aviation and communication tasks much like their commercial flight deck counterparts. They handle in-person flight deck conversations and attend to radio messages. Like their commercial counterparts, they have the capability to fly the aircraft using the Mode Control Panel (MCP) or the Flight Management System (FMS). While these systems may not reflect the actual systems found on AWACS aircraft, these functions were not critical to the scenario and hence were simply carried over from the commercial aircraft models.

The AWACS radar officer and the interceptor pilot models have roles that are more complex. The radar officer's principal responsibility is to monitor the airspace for unidentified aircraft. When an aircraft is detected, the radar officer has to determine its friend-or-foe status, and when appropriate, select a friendly aircraft to conduct the intercept and manage that aircraft's operation via radio communication.

The scenario begins with the interceptor aircraft flying a racetrack pattern. Unlike the commercial aircraft, the interceptor and the bogey have neither an MCP nor an FMS. The pilot models had to be significantly enhanced to enable them to "fly" the aircraft. Radar system use and observation also made new demands on pilot capabilities. The operations of initiating and conducting an intercept make extensive demands on the multi-tasking capabilities of the pilot models. The intercept is initiated by a radio-based interchange between the radar officer and the pilot. The pilot must vector the aircraft to the threat direction, make use of the radar to acquire the bogey, and maintain radio communication to track the status of the threat.

A task timeline display for a small portion of the tasks being executed by the pilot of the interceptor aircraft is shown in Figure 12. The timeline captures the high-level procedures that are being executed by the pilot in managing the transition from holding in the racetrack to initiating the intercept of the bogey aircraft. The pilot has first been vectored in the direction of the approaching bogey and then directed to pursue the intercept. The several procedures operating in parallel such as maneuvering the aircraft to the new vector, monitoring the primary flight displays, and tracking the bogey on the radar appear in off-screen portions of the timeline.

4.2 CAP Scenario Modes of Operation

The CAP scenario is representative of the many scenarios that have been developed using the OMAR simulation environment. These scenarios typically include a number of human players involved in the operation of complex equipment and are in communication with one another to carry out their tasks and address their responsibilities. When all of the players are human performance models, the system can be run in fast-time to take advantage of the speed of execution of the simulator, even when running complex models.

As it was in OMAR, in D-OMAR it is also possible to insert human players into a scenario replacing one or more human performance models. This is usually accomplished through some minor changes in the construction of the *defscenario* form used in setting up a simulation run. A simplified agent is used to shadow the human player. Its function is primarily to collect real-time data on the human player's interaction with the system. In operating the user interface, the human player operates workplace controls in the same manner that a model does.

The easy substitution of human players and models is made possible by the design and implementation of the user interface. The user interfaces are built so that they may be operated normally as a person might operate any user interface, but they may also be operated by the human performance models. The human performance models are able to "see" the screen and interact with input modalities of the user interface as a human player would. When there are human players, simulator operation is done in real-time. In the CAP scenario, it is the radar officer on the AWACS that plays the critical role in the simulation and the one for which a human player may be substituted.

The first version of the CAP scenario to run operated at a single Core-OMAR node. The AWACS and the friendly and bogey aircraft operated in D-OMAR just as they might have in OMAR. The basic scenario was then split into two separate scenarios, the first including the AWACS and the friendly interceptor, and the second including just the bogey aircraft. This provided a version of the CAP scenario that operated at two Core-OMAR nodes: the AWACS and the friendly interceptor operating at one node and the bogey operating at the second Core-OMAR node. In each of these scenarios, an aircraft signal provided the radar systems with information that they needed to track the aircraft. The aircraft agents published their position on a regular basis and the radar agents subscribed to these signals. The SCORE form, *signal-event-external,* is used to publish the aircraft location information. Locally, *signal-event-external* operates just as *signal-event* does. The combination of the OMAR Connector and the External Connector (see Figure 4) takes care of the remote propagation of the signals.

One version of the CAP scenario operates in fast-time using human performance models at all operator positions. A second version of the scenario operates in real-time and enables a human player to act in the role of the AWACS radar officer. The version of the scenario that operates on at two Core-OMAR nodes also operates in real-time with the role of the radar officer played by a human player.

Figure 15 provides a view of the radar screen as seen by a human player. The main pane shows the AWACS (WATCHER) flying a surveillance pattern, an aircraft identified as a

bogey intruding form the east, and a friendly just embarking on its intercept vector. The radar officer uses the selection items at the right to initiate the generation of a message. The template for the message appears above the radar screen. The receiver for the message and the object of the message are chosen from the radar screen via mouse gestures. The "Send" button is used to transmit the completed message. Outbound and inbound messages are recorded in the panel below the radar screen.

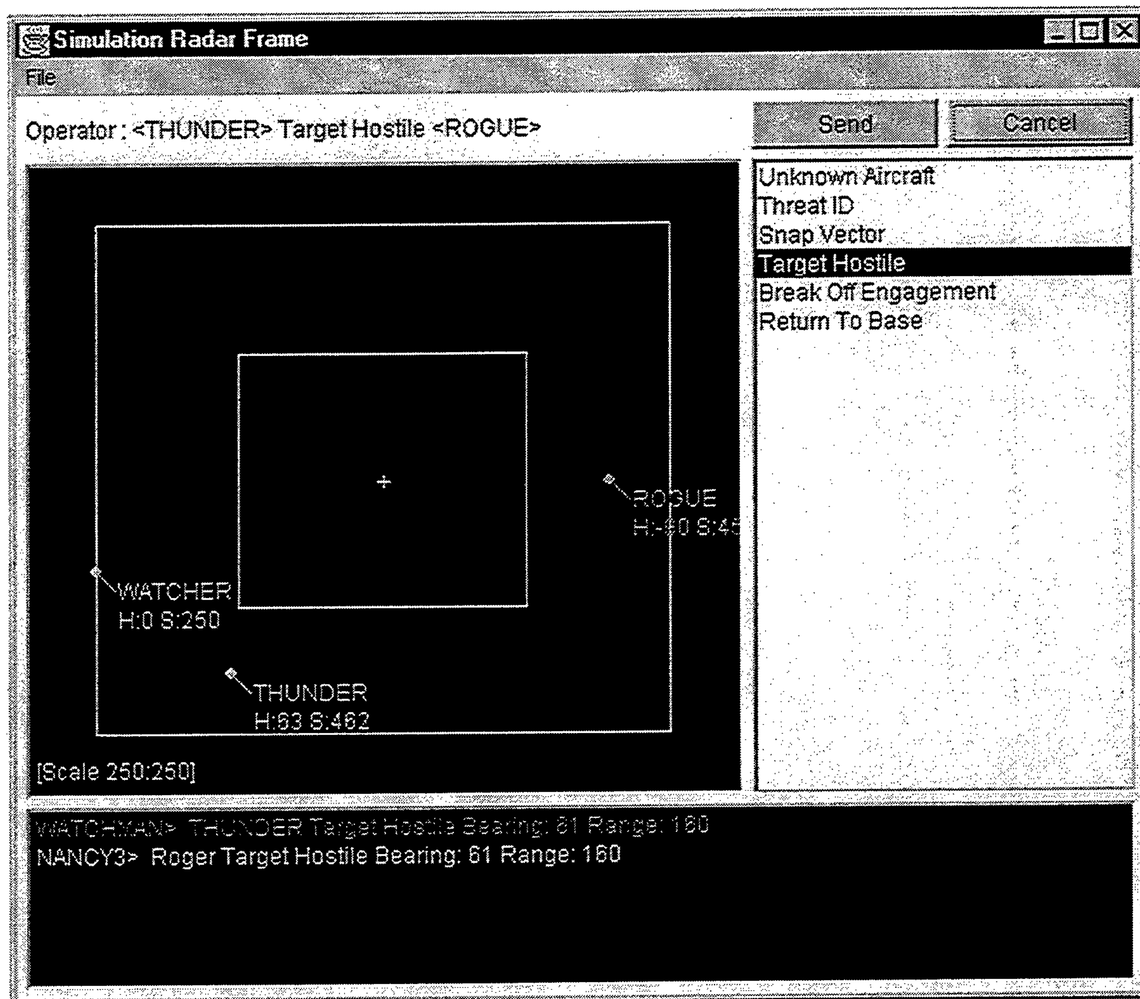


Figure 15. CAP Scenario Radar System Workplace

Figure 11 provides a view of the Simulator Control Panel during the execution of the CAP scenario. It is used by the operators of the simulation to manage the execution of the scenario and is not intended to be part of the application environment itself. The trace pane provides the operators of the exercise with an overview of the status of the scenario. It is the developers of the scenario software that control trace content and must insure that appropriate information is available to those managing scenario runs. The Control Panel provides filters that may be used to select which agents are traced and which event types are presented.

5. Acronyms

ACL	Allegro Common Lisp
CLIM	Common Lisp Interface Manager
CLOS	Common Lisp Object System
CORBA	Common Object Request Broker Architecture
DARPA	Defense Advanced Research Project Agency
DIS	Distributed Interactive Simulation
DMSO	Defense Modeling and Simulation Office
FMS	Flight Management System
FOM	Federation Object Model
GUI	Graphical User Interface
HLA	High Level Architecture
IIOP	Internet Inter-ORB Protocol
MCP	Mode Control Panel
OMAR	Operator Model Architecture
OMG	Object Management Group
ORB	Object Request Broker
PIASTRE	Piastre Is A STRucture Editor
RMI	Remote Method Invocation
RTI	Real-time Infrastructure
SCORE	Simulation Core Language
SFL	Simple Frame Language
SOM	Simulation Object Model

6. References

Bird, S. D. (1993). Towards a taxonomy of multi-agent systems. *International Journal of Man-Machine Studies 36,* 689-704.

Cramer, N. L. (1995). MIRAGE: A CLIM-based editor for building gadget-oriented graphical user interfaces. *Proceedings of the Association of Lisp Users Meeting and Workshop*, Cambridge, MA.

Deutsch, S. E. (1998). Interdisciplinary foundations for multiple-task human performance modeling in OMAR. *Proceedings of the Twentieth Annual Meeting of the Cognitive Science Society,* Madison, WI.

Deutsch, S. E. (1993). Notes taken on the quest for modeling skilled human behavior. *Proceedings on the Third Conference on Computer Generated Forces and Behavioral Representation,* Orlando, FL.

Deutsch, S. E., & Adams, M. J. (1995). The operator model architecture and its psychological framework. *Proceedings of the 6th IFAC Symposium on Man-Machine Systems.* Cambridge, MA.

Deutsch, S. E., Adams, M. J., Abrett, G. A., Cramer, N. L., & Feehrer, C. E. (1993). *Research, Development, Training and Evaluation Support (RDTE) Operator Model Architecture (OMAR) Software Functional Specification* (AL/HR-TP-1993-0027). Armstrong Laboratory, Wright-Patterson AFB, OH.

Deutsch, S. E., Cramer, N. L., & Clements, B. (1996). *Vehicle/aircrew procedure and control modeling: Volume 1: Aircrew procedure modeling.* BBN Technical Report No. 8121, BBN Corporation, Cambridge, MA.

Deutsch, S. E., MacMillan, J., Cramer, N. L., & Chopra, S. (1997). *Operator model architecture demonstration final report* (AL/HR-TR-1996-0161). Armstrong Laboratory, Wright-Patterson AFB, OH.

Freeman, B. (1997). *OMAR User/Programmer Manual, Version 2.0.* BBN Report No. 8181. Cambridge, MA: BBN Corporation.

Franklin, S. & Graesser, A. (1996). Is it an agent, or just a program? A taxonomy of autonomous agents. *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages.* Springer-Verlag.

Gray, R. (1996). Agent Tcl: A transportable agent system. *Proceedings of the Fourth Annual Tcl/Tk Workshop,* Monterey, CA.

Jennings, N. & Wooldridge, M. (1996). Software agents. *IEE Review,* January 1996, 17-20.

MacMillan, J., Deutsch, S. E., & Young, M. J. (1997). A comparison of alternatives for automated decision support in a multi-task environment. *Proceedings the 41st Annual Meeting of the Human Factors and Ergonomics Society.*

Steele, G. L. (1990). *Common Lisp: The language, Second edition.* Digital Press.

Wooldridge, M. & Jennings, N. (1995). Agent theories, architectures, and languages: A survey. In M. Wooldridge & N. Jennings (Eds.), *Intelligent agents* (pp. 1-22). Berlin: Springer-Verlag.

Young, M. J. (1998). *Computational modeling of memory: The role of long-term potentiation and Hebbian synaptic modification in implicit and schema memory.* Ph.D. Thesis, Miami University, Oxford, OH.