

Distributed-OMAR: Reconfiguring a Lisp System as a Hybrid Lisp/(Java) Component

Nichael Cramer
(ncramer@bbn.com)
GTE-Internetworking/BBN Technologies
Cambridge MA

Abstract

We describe our experience in converting OMAR, a large human-performance modeling simulation environment, originally written completely in Lisp, into Distributed OMAR [D-OMAR] a distributed, hybrid Lisp/Java system.

In the resulting system:

- The kernel simulation system (Core-OMAR) remains written purely in Common Lisp, and as such is able to take full advantage of the relevant features of Lisp.

- Other modules of the system (e.g. the User Interface) are written in languages more appropriate to the requirements of those modules (typically Java).

- The D-OMAR system presents itself to an external system as a fully functioning component in that system's embedded target language (e.g. JAVA or C++) and as such is able to take advantage of features in that language (for example, support for middle-ware or other mechanisms for inter-component communication).

- The underlying technique (built on an internal socket-based, inter-process, message-passing serialization stream) is applicable to a broad range of Lisp-based systems.

The result is a system that can be run nearly transparently in a range of external languages, with an assortment of middle-ware choices and on a variety of hardware platforms.

1. Background and Motivation

The Operator Model Architecture (OMAR) is a simulation development environment designed to explore and model human multiple task behaviors.

The central elements of the simulation environment include Lisp-based representation languages, graphical editors and browsers to support model development, and analysis tools to support model evaluation. OMAR has been used principally to examine teamwork situations in which small numbers of human players interact with one another and with complex equipment; air traffic control is typical of the

situations modeled. (Further discussion of the OMAR system, with references, can be found in Appendix 1.)

In the context of the present discussion, we note primarily that the original OMAR system was written in Allegro CommonLisp [ACL] with the GUI components written in CLIM.

While the OMAR system has been successfully deployed as a large, wholly-Lisp system for several years, it became desirable to use OMAR with other systems and, in particular, in ways that took advantage of the current movement towards component-based systems built on distributed architectures. Of particular importance was the desire that OMAR be able to run on multiple hardware platforms and in web-based applications.

Furthermore, it was clear that Lisp/CLOS provided overwhelming advantages in the core underlying simulation application (e.g. Lisp/CLOS provides full multiple-inheritance; dynamic class definition and allocation; full and natural object and class-reflectivity; rapid prototyping capabilities; etc.) As such, it was highly desirable that Lisp be retained as the implementation language for the simulation engine. However it was less clear that "peripheral" modules of the system were provided equal advantages by the use of Lisp.

Particular areas of concern were:

1] *The GUI system.*

Traditionally, the greatest impediment to the portability of Lisp systems has been the GUI/graphics component. This is primarily due to the highly platform-specific nature of graphics systems, whether the system uses a standard commercially available graphics package (e.g. CLIM) or it uses direct calls to platform-specific graphics toolbox.

Furthermore, for various reasons in most available Lisp-based graphics toolkits it is difficult to build interfaces that adhere to the standards of modern GUI-based design.

2] *Lack of direct middle-ware support.*

While some Lisp-based support for standard middle-ware protocols are even now becoming

available (for example, newly available Lisp/CORBA bindings) there are many important, standard middle-ware systems for which there is little or no direct support in Lisp (e.g. HLA, RMI, D-COM). And while there may eventually be Lisp-based support for one or more of these other middle-ware platforms, it seems clear that—at least for the foreseeable future—Lisp will be in a position of playing "catch-up" to other, more fully supported languages.

To summarize briefly, while the advantages of Lisp/CLOS seem clear for a broad range of applications, a design decision was made to consider a system in which the ancillary modules (GUIs, middle-ware connectivity, etc) were written in a non-Lisp language.

2. Specific Goals

During the design of D-OMAR it was deemed desirable that the system should exhibit the following five characteristics:

1] *Distributed*

D-OMAR should be "distributed" in the following two senses.

First D-OMAR should be able to participate as an equal in a system whose architecture consisted of multiple remote components.

Second D-OMAR's internal modules should be able to run in as distributed a manner as needed. (That is, while the standard model would be that the sub-modules within the D-OMAR component would typically run on a single hardware platform, we should not be restrictively tied to this architectural model. Specifically, if it became necessary or useful that the various internal sub-modules separable—say for reasons of workload sharing—the core-Lisp system and the GUI modules would be capable of running on separate platforms.)

2] *Portable/Flexible*

D-OMAR should be usable in a broad range of application-systems. Specifically, D-OMAR should be able to be used in application systems 1] written in a broad range of implementation languages, 2] which communicate through an assortment of possible middle-ware protocols, and 3] which run on various hardware platforms.

Finally, it was desirable that that the core-Lisp modules of the D-OMAR system not be tightly coupled to a particular Common Lisp.

3] *Stable*

D-OMAR should be as stable as possible when faced with the inevitable change of standards.

In particular, because of the volatility of the current middle-ware market, it was desirable that D-OMAR not be tightly bound or irrevocably committed to, for example, CORBA or RMI. D-OMAR should be

able to change as rapidly and painlessly from one system-paradigm to another as possible.

4] *Maintainable*

D-OMAR should be as modular as possible while still being able to work in such a broad variety of systems.

Specifically this means that, to the degree possible, there be only one D-OMAR. That is, there should not be a "CORBA D-OMAR"; an "RMI D-OMAR"; a "HLA D-OMAR"; etc.

D-OMAR should work in all of these "incarnations" but the changes to D-OMAR should be as localized and well-defined as possible. In particular, it was deemed to be important that the underlying Lisp system should not change at all, in an effort to enhance the robustness of the entire system.

5] *Generality*

Finally, we should note that while, throughout the following, we use D-OMAR as the primary example (and, as we shall see, there are certain features of the OMAR system that make it particularly amenable to this approach) there is nothing special in this case about the OMAR system.

In short it was desired that the design process for D-OMAR should result in a general approach, suitable for and applicable to a broad set of Lisp systems.

3. Approach

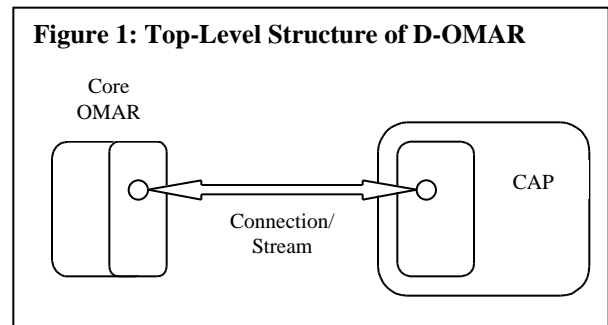
In attempting to achieve these goals, the approach that D-OMAR takes can be summarized as follows:

D-OMAR presents itself to the external system as a single, unified component written in the target language.

That is, in a Java-based external system the D-OMAR component presents itself as a fully functional "D-OMAR Bean". In a C++ based system (such as the standard implementations for the HLA implementation described below), D-OMAR presents itself as a standard "D-OMAR Federate". And so on.

Looking at this statement in a bit more detail (See Figure 1):

1] The Core-OMAR system is written in pure,



standard CommonLisp/CLOS that has been extended to support sockets and processes (as we shall see the use of processes is not essential).

In particular, Core-OMAR directly supports no GUI-code.

Furthermore it is important to note that Core-OMAR can run in any CommonLisp —and on any hardware platform— that supports a socket (and a process) package.

2] The Lisp-based Core-OMAR system communicates with its modules in the external, embedded language through a "Connection" object. This Connection object communicates with the Lisp module by a lightweight socket-based, serializing message/event-passing stream.

This external (i.e. non-Lisp) portion of this Connection can be written in any language that supports socket-based communication and in which we can support the serialization of the message/event-objects. (Currently D-OMAR has available external implementations of this Connection in Java, C++, and Lisp.)

3] The outermost interface between the external system and this Connection object is a "Cap" object that serves as a wrapper around the Connection object and encapsulates all knowledge about the details of communication with the external system.

This approach results in the following four characteristics of the D-OMAR system:

- Because D-OMAR presents itself as a fully functioning component in the target language, it is able to "piggyback" off the desirable features of that language.

Of particular interest are the communication and middle-ware features available in the external language. If, say, a binding to a particular middle-ware (e.g. CORBA, RMI, HLA) exists in the external language, it become readily accessible to D-OMAR.

- The Lisp-based Core-OMAR module remains compact and intact.

That is, the OMAR system is able to use Lisp to its fullest advantage, using those features that make Lisp the most suitable language available for writing the simulation, while off-loading those tasks for which it is arguable less well-suited.

In short, Lisp is allowed to do what it does best.

- The D-OMAR system is highly portable.

The Lisp-based Core-OMAR module remains written in pure CommonLisp and the external modules are written in Java. As such, the D-OMAR component can run on any of the large number of platforms that support these languages.

- As D-OMAR performs its roles in the various systems in which it functions as a component, only the Cap layer of D-OMAR changes.

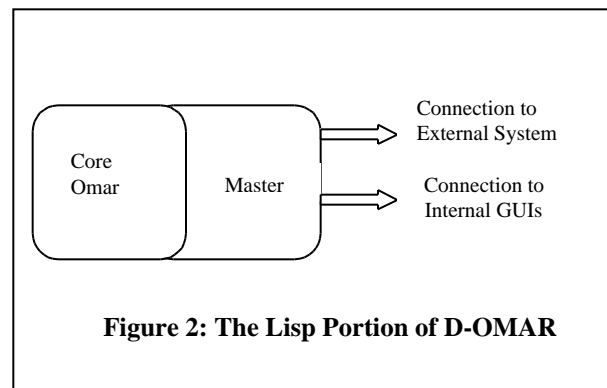
In the various "incarnations" of the D-OMAR system —as it is incorporated into various systems using possibly different middle-ware protocols— the only portion of the D-OMAR structure that is altered is the thin, outermost Cap module. All underlying structures remain unchanged.

4. Structure

Core-OMAR

1] Core-OMAR is the original, Lisp-based "kernel" of the OMAR simulation system. It is written in pure CommonLisp/CLOS with the addition of a process-package. In short it can run in any Lisp implementation that supports these features. Current implements of this module have been run in Allegro CommonLisp (ACL) and Macintosh Common Lisp (MCL).

(We should note that the use of processes is feature specific to Core-OMAR and while the use of



processes is reasonably standard in the various implementations of Lisp, there is nothing about the use of processes that is required in the overall D-OMAR architecture.)

2] Within Core-OMAR, the primary means of communication is event-based. That is, the mechanism for communication between agents within an Omar-Model and between the Core-OMAR simulation and the "external world" is by means of events or "signals".

In this context an "Event" —or more generally a "Message"— should be viewed as an object containing a high-level specification for a request for a state-change in another system module or component.

(Again, we should note that while communication in Core-OMAR is event-based, and while this event-based structure simplifies certain issues that come later, there is nothing about the use of events that is required for the use of a core Lisp-based system in a D-OMAR-like architecture.)

3] Finally, the restructuring of Core-OMAR involved the specification of a well-defined, cleaned-up functional API specifying the nature of communication to and from the Core-OMAR module.

The "Master" Object

Communicating directly with the Core-OMAR systems is an abstract module, the so-called "OMAR Master". Architecturally, this is a module that implements the Core-OMAR functional API and, as such, governs communication between Core-OMAR and the "outside world".

(We will briefly mention two early implementations of the Master object:

1] A "Command Line" master that allows direct "command-line"-like control of Core-OMAR and which printed output from Core-OMAR to a standard Lisp-listener.

2] An "OMAR Classic" Master, that allowed communication with a cleanly separated version of OMAR's original CLIM-based GUIs.)

Distributed/Connection Master.

1] Of particular interest to us in the current discussion is the "Distributed" or "Connection" implementation of the Master object. This is the module of the D-OMAR system that handles communication between the Lisp process (containing the Core-Lisp module) and the portion of D-OMAR that function in the external language (such as Java).

[NOTE: It is important to note that in much of what follows "Java" should be considered as a "meta-variable" or "wildcard". While Java is our language of choice in implementing external modules to the D-OMAR system, there is nothing special or restrictive about our use of Java. The mechanism, as described, works with any external language that can support the appropriate protocols. As will be seen in the discussion of the HLA example below, this mechanism work, for example, with C++]

At the heart of this module is the mechanism supporting communication between the Lisp process and the external Java process. This communication is implemented by a lightweight socket-based, serializing, message-passing stream.

"Sockets": A socket is a standard mechanism for inter-process communication available on virtually all platforms (currently all such communication within the D-OMAR is based on standard Berkeley sockets).

"Serialization": The serialization of an object means to convert its state into a byte stream in such a way that the information contained in the byte stream can be restored into a copy or "clone" of that object, for example in a remote process.

"Message-Passing": In this context, a "message" should be considered simply an abstraction of an event or signal; i.e. an object containing a high-level specification of a state-change, which is passed between two components or modules. It is assumed that communication between the Lisp and the Java sides of the D-OMAR module will be wholly in terms of such messages.

2] On Lisp side (see Figure 3):

The Lisp end of this protocol consists of a socket-based server that functions as a distribution center or "post-office" that handles the transmission and reception of events between Core-Omar and the external system.

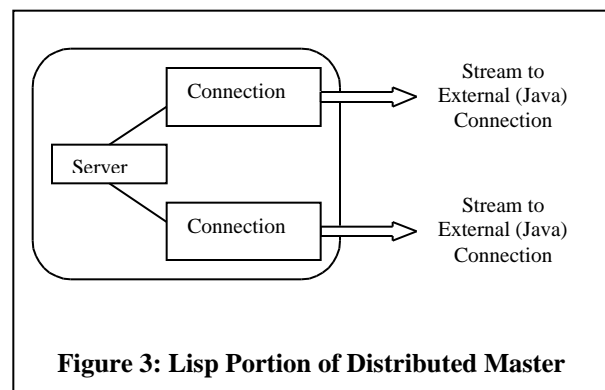


Figure 3: Lisp Portion of Distributed Master

In particular the Lisp portion of the Master:

- Handles the establishing of connections with the external "client" Java-connections.
- Receives in-coming external events and distributes them to the appropriate section of the Core-OMAR API.
- Distributes out-going events generated by the Core-OMAR to the appropriate external client-connection.

The Lisp side of the module is written in pure Common Lisp/CLOS with extensions that support 1] sockets and 2] processes. (Again, in a way that is similar to the case of Core-OMAR, processes are used in the Distributed Master primarily in support of the multiply-threaded server design; as such its use is not strictly required.)

3] On the Java side (see Figure 4):

The Java end of the Connection is implemented as a socket-based client that functions as a basic source/sink of messages within the target language.

(In the current D-OMAR system, implementations of this external Connection exist in Java, in C++, and in Lisp.)

As such, the Java-Connection presents a message-handling "black box" to the system-specific Cap described below.

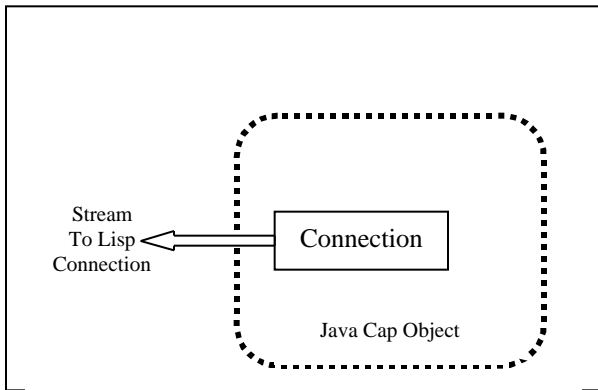


Figure 4: Java Portion of Distributed Master

4] The Connection Stream:

Lying between the Lisp-Connection and the Java-Connection is an "Object-Stream" that passes serialized messages between the two connections. A more detailed discussion of this serialization stream appears below in Appendix 2.

The use of this approach results in three important features of D-OMAR's system for communication with the external system:

- There is a single, well-established protocol for all Connection implementations. On the Lisp side, establishing a new Connection-type simply involves defining a new Server-Connection object that knows how to deal internally with appropriate types of messages.

On the external side, as mentioned above, an event/message-passing connection object already exists in Java (and C++) and would serve as the basis for the portion of the module that exists in the external language.

- In this way we should note that D-OMAR —as viewed through the external Java-Connection— is now "just another Java (or C++) component".

- While the standard model of the D-OMAR component assumes that its various sub-modules, including the various Connections, run on a single hardware platform, the socket-based communication between the parts of the various Connections ensure that this assumption is not an actual restriction.

In short, each of the various external modules of the D-OMAR Component can run on separate

hardware platforms as required by the details and requirements of the overall system architecture.

The Cap

In the D-OMAR system the "Cap" functions as a wrapper around the Java-Connection which presents an appropriate "face" to the external world.

As described above, the Java-Connection functions as a black box whose primary method of communication is through receiving messages from the external systems and transmitting messages to the Lisp side of the module.

In this way the Cap mediates between communications from the Java Connection (in the form of the raw D-OMAR signals) and, for example, the method calls appropriate for a standard Java-Bean.

In short, all information or functionality specific to communication with the specific external system is captured in and restricted to the Cap. In particular, nothing "upstream" from the Cap is system- or application-specific; nor does it change as different middle-ware-specific versions of the Cap are used.

5. Specific Connections in current D-OMAR system

In the current D-OMAR system there exist three external Connection types: an OMAR-Control GUI Connection; an OMAR-Application specific GUI Connection; and an External Connection.

Control GUI Connection:

The standard "Control" GUIs in the D-OMAR are written in Java. These include, e.g. model editing tools, simulation controls, etc.

The structure of this collection of displays consists of:

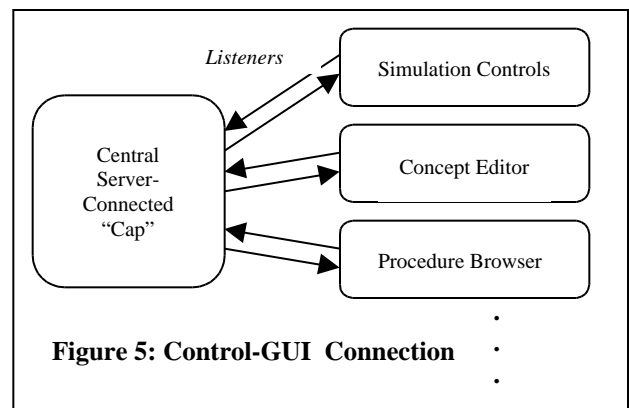


Figure 5: Control-GUI Connection

- The main Server/Core-OMAR-connected Cap Module.

This module is connected directly to Core-OMAR through the protocol described above. As before, the Java-Connection transmits events to and receives events from the Core-OMAR system. Surrounding

this is a “Cap” object that allows the Connection to communicate with the other display components through the standard Java Listener [subscriber] and Source [publisher] mechanism.

This central source in turn communicates with:

- A number of Java based displays.

The various display components communicate with the Core-OMAR system —via the main Server Component— through standard event/message-passing.

In short, when a new display is created, it registers itself as a Listener for the specific class of events which it is interested in receiving. Likewise, the Server Component registers itself with GUI Component as a listener for the GUI Component's "Server Events" through which the GUI Component communicates to the Core-OMAR system.

Once again, we should stress that, to the external Java-based GUI clients, the event-serving Cap —i.e. the "face" that the D-OMAR system presents to the external GUI components— is simply "just another Java Component". As such it becomes possible for the inter-component communication to be modeled —as in the present example— on the standard Java Listener paradigm for Event generation and subscription.

One of D-OMAR's standard GUI Components (the "Procedure Browser") is discussed below in Appendix 3.

Finally, the assumption is that the various Modules of the D-OMAR Components —including the Control GUIs— are part of a single Component and as such would typically run on a single hardware platform. However, as should be clear from the discussion above, this is not a requirement of the system. The Control GUIs —all or any of them— can run locally or on another hardware platform as need by the requirements of the overall system (for example, for reasons of "work-load sharing" or constraints on "screen real-estate").

Application GUI Connection:

This connection is used to handle any application or simulation-model specific GUIs (for example a radar-screen display) that may be required by a specific application.

(Note that the “location” of such a display component in the overall system-architecture is a system-level design decision. For example, in a stand-alone system, in which simulation modeling is completely contained within a single OMAR image, it makes sense to implement such a radar display as a "local" module, within the D-OMAR system. On the other hand, such a radar display —for example, a pre-existing display— might also function as an external Component in a more distributed system architecture.)

The internal architecture for this set of components is structurally equivalent to that of the Control GUIs as described above. That is, a central event-serving Cap module transmits events to and receives events from the various (Java-based) GUI Components.

The External Connection:

This connection is responsible for communication between D-OMAR and the other Components that reside in the external system.

1] As described above, the Java side of this External Connection consists of two sub-modules:

- The Java-Connection which functions as a source and sink of events for communication to and from the Core-OMAR system.

Note that this sub-module is common to and forms the basis of all the implementations of the External Connections as used by the D-OMAR system.

- The system-specific Cap which encapsulates all of the D-OMAR Component's knowledge and details about the external system. That is, the Cap functions as a "mediator" or "translator" between the information supplied by D-OMAR (in the form of internal events/messages) and the form of that information as required by the external system.

As such the details of the inter-component communication through middle-ware (e.g. CORBA, RMI, HLA), and the form in which such information is required by the external system (e.g. through method calls; event-like callbacks; and so on) is handled wholly by the Cap.

2] A simple use of the External Connections occurs in a system whose Components consist entirely of separate instances of D-OMAR components.

In such a system, communication among the D-OMAR Components is structured in terms of external events consisting solely of raw OMAR-signals. Communication between the D-OMAR Components can be handled through any of the standard Java-based communication protocols.

For example, such a system might have a single Java process that contains a central "initializing" component whose responsibilities include launching the various D-OMAR Components objects and "wiring them together" by subscribing each of the D-OMAR Components to the appropriate event types of the other components.

Only slightly more complicated is a version of this system in which the various Components are implemented "remote" objects (in the Java sense) and communicate remotely through Java's RMI protocol.

In such a system each D-OMAR External Cap's responsibilities consist of:

- Receiving out-going events from its internal Java-Connection and passing them on to the external Components which have subscribed to such an event (typically by calling a method on the external Component)
- and
- Receiving in-coming external events from an external component (again, typically as the result of a method call from the external component) and passing the in-coming event into its internal Java-Connection.

3] A somewhat more complicated example is provided by the use of D-OMAR in an HLA-based system.

HLA [High-Level Architecture] is an ORB-like Component platform, which is especially suited to the building of simulation-modeling systems.

The interactions among the active simulation Components (called "Federates" in the HLA system) are completely governed by the HLA system. The central HLA system is responsible for:

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

As well as other standard simulation-related "services".

As such the expected functionality of a D-OMAR based Federate is much more complicated than in the sketch of the system above. For example the D-OMAR Federate is expected to handle specifications for system-wide naming conventions; to handle notification of changes of attributes of external entities; to publish changes to attributes of internal structures; etc.

However, although the basic functionality is significantly different, the fundamental architectural structure of these two implementations of the External Cap module is the same. I.e. the outer enveloping HLA-Cap object communicated directly with the external HLA world. In turn the Cap makes requests on and receives information from the internal Connection object via internal Core-OMAR events.

Finally, we should note that, at present, the primary language for implementation of HLA-based systems is C++. However, from the above discussion, it should be clear that such a language difference

results in no change, architecturally, in the structure of the underlying D-OMAR Federate; the only difference is that the core connection in the target language is implemented in C++ rather than in Java.

(Moreover, it is expected that a Java-based or a mixed Java/C++ implementation of HLA will soon become available.)

6. Conclusions/Summary

In the current state of the D-OMAR system:

D-OMAR runs under Unix, Window (WNT/W98) and on the MacIntosh¹ and has been incorporated into systems based on a variety of middle-wares (CORBA, RMI, HLA).

Furthermore, as described above: With regard to the cases where D-OMAR is used with the various middle-wares, all changes in the D-OMAR system are restricted to the respective CAP module.

Beyond the noted changes to the CAP module, in all these configurations of D-OMAR (with one exception²), the remainder of D-OMAR remains unchanged and runs with—at most—a simple re-compilation of the Lisp portions of the system.

7. Acknowledgements

The author wishes to thank Dr. Michael Young of the USAF Research Laboratories for his continued support in this effort. This research was conducted under Department of the Air Force Contract F41624-97-D-5002. To Patrick Coskren of GTE-I for help with the formatting of this paper. And especially Dr Stephen Deutsch for maintaining the environment in which this work has proceeded.

¹ Specifically, the Java-based modules of D-OMAR have currently been run on Unix (Solaris), Windows (WNT/W98) and Macintosh. The CommonLisp-based Core-OMAR module has been run in (CLIM-less) Allegro CL (available on Unix—Solaris and SGI— and WNT/W98) and in Macintosh CommonLisp.

² The version of the Core-Omar module based MacIntosh CL, the socket-communication is based on Open-Transport rather, than in all other cases, on standard Berkeley Sockets. This may change with the availability of a Berkeley-socket package for MCL.

8. Bibliography

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

Appendix 1: Overview of OMAR

Simulation has been used for many years as a tool to evaluate system performance, but it is only recently that attempts have been made to include realistic models of human operators in these evaluations. The Operator Model Architecture (OMAR) is a simulation system that addresses the problem of modeling the human operator. Its development focused first on the elaboration of a psychological framework that was to be the basis for the human performance models, and then on the design of a suite of software tools to support the development of these models. In using OMAR to build human performance models, particular attention has been paid to the representation of the multi-tasking capabilities of human operators and their role in supporting the teamwork activities of the operators.

The ability to model human operators and their interactions with other team players and with their target systems has opened up several new areas for investigation through simulation. Procedure development is an important one. Simulation can now have an impact on both operator-procedure and maintenance-procedure development. It will now be possible to pursue procedure development by evaluating procedures far earlier in the design cycle of a system than was previously possible. The evaluation of both operating and maintenance procedures as part of the design process can lead to significant improvements in system operability and reduce downstream maintenance costs.

The Simulation Core (SCORE) language, the OMAR procedural language, provides the basis for representing functional capabilities. A SCORE procedure is used to represent a simple capability. The procedure is typically in a wait state, pending activation based on a match to a particular stimulus pattern. Several such procedures may represent the components of a particular functionality. The procedures form a network along whose links pass the

signals that each of them may generate. For any signal generated, one or more related procedures may be enqueued on it. Depending on the response of their respective pattern matchers, some procedures may be activated by a stimulus, while others may ignore it. The pattern of activation in a complex of procedures differs due to variations in the stimulus patterns.

Within any given complex, several procedures may be running concurrently, some representing automatic processing, others representing components of attended processing. In the several layers of concurrent processing, a sensory input may have initiated the "lowest" processing level, with each subsequent "higher" processing layer activated at the behest of the initial "output" from the next lower level. Proactive behaviors are the product of the set of activated goals and procedures that initiate agent behaviors, and interpret patterns of sensory input and channel responses in a manner consistent with agent goals.

As mentioned above the original OMAR system was written in ACL with the GUI components written in CLIM. (More information on the OMAR system and its associated models of human-behavior is available in *Deutsch and Adams 1995* and *Deutsch1998*.)

Appendix 2: Structure of the Object-Stream and Parser

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 basic abstract class underlying the serialization stream is the class *BASIC-OBJSTREAM*.

The *ObjStream* is primarily 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.

The two primary (public) methods on an *ObjStream* are:

```
(defmethod objstream-read-object  
  (OBJSTREAM) ...)
```

```
(defmethod objstream-write-object  
  (OBJSTREAM OBJ) ...)
```

Details of the specific serialization protocol used by such a stream are the responsibility of its internal *PARSER* object. That is, the *PARSER* object knows how to "serialize" an out-going (i.e. "written") Object into an appropriate sequence of bytes and how to read a corresponding series of bytes from which it can construct an in-coming (i.e. "read") Object.

In the standard D-OMAR system, the standard class for an *ObjStream* is *JAVA-SOCKET-OBJSTREAM*. When it is created, this *ObjStream* receives two objects: 1] a physical Socket-Stream (from which it reads and writes bytes) and 2] a parser object.

The standard class for *PARSER* objects used in an *ObjStream* in D-OMAR is the *ObjStreamParser_OMAREvent*.

The class hierarchy underlying this class is

-- *ObjStreamParser* which primarily is responsible for handling objects of unrecognized types and other utility functions.

-- *ObjStreamParser_Base* which is responsible for dealing with error handling and default handling of the Lisp NIL.

-- *ObjStreamParser_Core*, which is responsible for the parsing of common, "standard" language types: such as Strings, integers, arrays, and vectors.

-- *ObjStreamParser_Lisp*, which is responsible for the parsing of standard "lisp-like" language types: such as Cons, Symbols, Keywords, Lisp T and Lisp nil.

-- *ObjStreamParser_OMAREvent* which is responsible for the parsing of classes used specifically in the D-OMAR system, such as the various event classes and the classes of internal data-structures passed to the various displays.

(Note that under this architecture no restrictions are placed on how the *ObjStream* objects are used by the D-OMAR system: i.e. socket-based communication or the set of Lisp-types and classes used by the D-OMAR system.)

Appendix 3: The Procedure Browser: A Fully Hybrid Lisp/Java Gadget.

All of the Display components in the D-OMAR system are written in Java. For most of the more complex display gadgets in the D-OMAR system the standard mechanism is for the gadget to retrieve (or receive) a serialized version of the corresponding Lisp data-structures and to display those data-structures using appropriate native Java display gadgets.

For example, this approach is typified OMAR's Concept Editor. As described, the display receives a data-structure representing a portion of the Concept hierarchy currently residing in the Core-OMAR system and graphs it using a native Java graphing package).

However, D-OMAR's Procedure Browser behaves in a significantly different manner.

Underlying the original version of this display was the CLIM-based PIASTRE ["Piastre Is A STRucture Editor"] system, which was designed to be used for the graphical, structural representation and editing of code. (Note, that while the underlying features of Piastre support structure editing, it is primarily used in the OMAR system as a "browsing" —i.e. non-editing— tool.)

Because the Procedure Browser (and Piastre) represented a significant amount of code that would not translate directly or easily into Java a decision was reached to attempt to use as much as possible of the underlying, original Lisp code in direct support of the planned Java gadget.

Towards this goal a "Draw-Stream" protocol was designed and implemented. This 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*, etc) 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 D-OMAR this is handled by a *DrawStreamDisplayPanel* implemented in Java).

Similarly, the gestures to the Java display (e.g. mouse-clicks) can be passed back to the Lisp-side and the Procedure Browser code can, for example, determine which underlying Lisp object occurs at that screen location, and make an appropriate response (for example, supplying "drill-down").

Two versions of the Draw-Stream have been implemented. First, a fully stream-based communication in which a standard byte-stream was opened between the Lisp process and the Java gadget and the sequence of bytes transmitted directly. Second, (as is used in the current D-OMAR system) a batch-like approach is taken, in which the sequence of bytes is first captured in an array, and this array is then transmitted (for example, through an event/message) to the Java display gadget.

In short, in the Procedure Browser all of the "drawing" (as well as layout, data-manipulation, etc.)

is handled strictly within Lisp, while the "display-rendering" is handled wholly in Java.

Among the advantages of the Draw-Stream architecture are:

1] Converting existing code is very quick. Once a Lisp-side Draw-Stream is in place and the code for the Java-side display gadget is finished, all that is involved is replacing the standard Stream object (to which the display-code draws) with the Draw-Stream object.

2] As such, the great majority of the existing Lisp code can be reused.

3] Again, as stated above, 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 is in place, the graphics/display system (for example CLIM) can be removed from the Lisp application, thereby effecting a

potentially great reduction in the delivered Lisp image size.

(We should note that much of the above discussion assumes that the original Lisp graphics-stream is used in a more-or-less "vanilla" way. That is, that the calls on the graphics-streams are primarily "graphics" calls —e.g. *DRAW-LINE*, *DRAW-TEXT*, *SET-COLOR*, etc.— and that a particular display does not depend on non-standard features of the particular stream; for example such as a display that uses the graphics-stream to, say, use the CLIM-like output-records to layout and manage the components of the display. However, since even such complicated "displays" almost always finally bottom out in standard graphics calls, it is still generally possible to use this approach.)