Interfacing External Decision Processes to DIS Applications

Elizabeth L. White, Kenneth E. Frosch, Vincent P. Laviano, Michael R. Hieb, J. Mark Pullen
Department of Computer Science and C3I Center
George Mason University
4400 University Drive, Fairfax, VA 22030
white@cs.gmu.edu

1. Abstract

The Distributed Interactive Simulation (DIS) protocols (IEEE 1995) allow heterogeneous applications to share exercise information. As DIS applications mature and transition to the High Level Architecture (HLA), their utility can be increased by integrating them with external decision processes. External decision processes comprise a wide range of applications such as inference methods, learning algorithms, and collaborative planning tools. There are several possible approaches to accomplishing such an integration. We analyze various criteria to identify the best architectural approach for a specific set of requirements. The criteria discussed are distribution of processes, data representation, interprocess communication, and quality of service. This paper describes three approaches and presents two case studies in constructing interfaces to the Modular Semi-Automated Forces (ModSAF) CGF simulation. By conducting a more formal analysis of the interface design process, we provide system designers with the basis to make a more informed choice when designing an interface.

2. Introduction

DIS-compliant simulations have been extensively utilized in recent automated warfighting exercises. Interfacing external decision processes to these simulations can improve their behavior and make them more widely applicable. The interface of an external decision process to a DIS-compliant simulation may take one of several forms. We have identified several criteria for designing such an interface of non-DIS applications to DIS simulations:

1) distribution of processes — describes the degree to which the processes are distributed, ranging from a single centralized executable to processes distributed over a network.

2) data representation — determines the format the processes use to communicate information.

3) interprocess communication — the method that the processes use to communicate, for example, shared memory or message passing.

4) quality of service factors — describe requirements for distributed system latency, security and robustness.

Section 3 covers these issues in detail. From these criteria we identify three general approaches to interfacing external decision processes to a DIS application. Sections 4 describes two case studies as examples of two of the approaches. In these case studies, the DIS application with which we integrated our external processes was ModSAF. Further details about ModSAF are given in Section 4.1. We compare our approaches with related work in Section 5 and analyze the impact of the HLA on our integration methods. Conclusions are presented in Section 6.

3. Interface Design Issues

Several issues must be considered when interfacing an external process to a DIS application. Among those are the four criteria enumerated above. In addition to these technical factors, another factor that may influence design decisions is the existence of components, such as libraries implementing an Application Program Interface (API). Such components constrain the choices available for the architecture, however their use can also save considerable time and effort. For example, COMPASS provides libraries for performing a Remote Procedure Call (RPC) between a client application (ModSAF in our case) and the COMPASS server.

Table 1 lists the issues that need to be addressed with possible options based on the quality of service factors.

Issue	Option
Degree of Process	Single process

Distribution	Multiple processes, single host
	Multiple hosts, single subnet
	Multiple hosts, multiple subnets
Data Representation	Standard format
	Ad hoc format
IPC	Shared Memory
	Remote Procedure Call
	Message Passing

Table 1: Design Issues and Options

3.1 Degree of Process Distribution

One of the most fundamental decisions that must be made when designing the software architecture for an integrated application is the degree of distribution of the constituent processes. The level of distribution will have a great effect on the other issues of the design, particularly the IPC mechanism.

The two applications can be integrated into a single process (Figure 1), eliminating the need for interprocess communication, or the integrated application can be split into multiple processes (Figure 2), providing additional flexibility, but necessitating interprocess communication. If these processes reside on a single host, then this additional flexibility is limited, but there is a wide selection of interprocess communication mechanisms and several data representation issues can be avoided because a homogeneous environment is guaranteed.

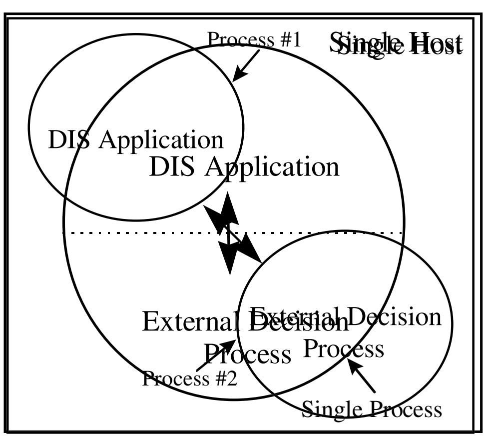

Figure 1: Single Process on a Single Host
Figure 2: Multiple Processes on a Single Host

If the processes reside on multiple hosts, then the characteristics of the intervening network become an issue (Figure 3). The network may be a local area network (LAN) in which case there will be high reliability and little latency, but the geographic scope of any DIS exercise run using the integrated software will be severely limited. On the other hand, the application may be distributed across a wide area network (WAN), in which case there will be less reliability and increased latency, but the geographic scope of an exercise will be virtually unlimited. These characteristics may vary depending on whether the WAN is private (and can be dedicated to the DIS exercise in question at the time it is running) or public (and having to support a great deal of unrelated traffic at the same time the exercise is running).

3.2 Data Representation

Data representation is an important issue because the external process and the DIS application need to share information. Given the fact that these applications may reside on different types of hardware, any solution should accommodate heterogeneous environments. There are two ways to allow information sharing: 1) converting one of the applications to the other's data representation or 2) using a standard format as an intermediate language.

The first option is often taken. The process of converting one application's data representation is often tedious and error-prone. Using a standardized format, such as the DIS protocols, does require changes to both applications and additional encoding and decoding effort at runtime. However, once an application has been modified to allow it to use the standard format, it can interoperate with any other application conforming to the standard as well.

With both options, the data must either be initially represented in both applications in some common format, or translated into a common format that both applications can interpret and manipulate. The common data format may be either a standardized format (e.g. the DIS protocols or the External Data Representation (XDR) standard used to transfer data between different machine architectures (Sun, 1987)), or an ad hoc format.

3.3 Interprocess Communication

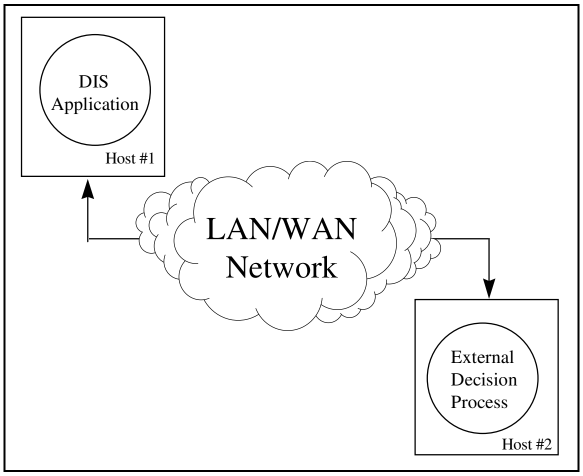

Figure 3: Processes on Multiple Hosts on a Network

As noted above, if the integrated application is composed of multiple processes, some form of interprocess communication (IPC) will be necessary in order to allow the processes to share data. The choice of an interprocess communication mechanism will follow largely from the choice of a process architecture (single process vs. multiple processes, distributed over a network, etc.). The need for IPC presents a design challenge because of the real-time requirement imposed by the DIS component of the integrated application. This requirement usually necessitates the use of either non-blocking IPC or a timeout mechanism. A non-blocking IPC would allow the DIS application to continue executing while the external process services the request.

The two fundamental forms of IPC are shared memory and message passing. Shared memory is known to be faster in most circumstances (Stevens 1990), but its use is limited to applications running on a single host. Therefore, message passing will often have to be used. A popular paradigm for message passing is the Remote Procedure Call (Birrell and Nelson, 1984), in which the message passing primitives are hidden from an application programmer in linkable libraries, allowing the programmer to call both local and remote functions using the same semantics. A case study of an RPC interface is given in section 4.1.

Another possibility for interprocess communication is the distributed object technology embodied in the Object Management Architecture (OMA). Central to the OMA is the Common Object Request Broker Architecture (CORBA), which provides for transparent communication between distributed application objects (Yang and Duddy, 1996). The prototype implementation of the HLA's Run Time Infrastructure (RTI) uses CORBA. The RTI's API is specified in CORBA's Interface Definition Language (IDL) (Calvin and Weatherly, 1996).

3.4 Quality of Service

Integrated applications with a DIS component are likely to have special requirements in one or more of the following areas:

Latency. The integration of an external decision process with a DIS application implies a real-time requirement for the composite application. Meeting this requirement may be a challenge, because the external process may not have been designed with real-time performance in mind. However, if the DIS application is utilized in a mode where it is not operating in real-time, such as generation of simulation scenarios, then latency does not need to be considered. Examples are given in Section 4.

Correctness. In order to maintain the consistency of a distributed simulation it is necessary that all participants receive similar data about the simulated virtual world. This implies a requirement that some simulation data be transmitted reliably. In general, DIS applications are designed to handle the loss or corruption of an entity state PDU, because they are sent with great frequency and each new entity state PDU supersedes previous ones. However, there are other, individual PDUs which represent important events such as collisions and must therefore be received reliably in order to ensure the consistency of the distributed simulation. Since reliable transmission of data contributes to latency, a balance needs to be struck between correctness and latency.

Robustness. Unlike a traditional centralized application, a distributed application can continue to exist after the failure of a host on which parts of the application are executing. This may entail the relocation of simulated entities (or an external decision process) to other hosts, or it may be sufficient for the simulation to continue without the entities in question and without interaction with the decision process.

Security Issues. When an application is distributed across multiple hosts, the intervening network increases vulnerability, especially if the constituent processes reside on multiple subnets separated by a wide-area network that is outside the administrative domain of those organizations conducting the exercise. Encryption and/or other forms of security may be required. The interface will need to be designed with this and other administrative concerns in mind.

Concurrent Access. It may be necessary or desirable for multiple clients to access resources managed by the integrated application concurrently. This is especially likely to be true for the external decision process portion of the application if it is implemented as a separate process. A need for concurrent access to resources will impose additional design constraints on the interface.

4. Case Studies

In this section, we first describe three basic approaches to interfacing an external decision process to a DIS application and then present detailed case studies of two of these approaches. The three basic interface approaches that we have identified are to 1) create a single executable, 2) utilize existing IPC mechanisms for distributed processes, and 3) use DIS PDUs to implement the interface.

The first approach requires integrating the DIS application and the decision process into a single executable. The Soar/Intelligent Forces (IFOR) architecture is an example of this approach. Soar provides the reasoning capabilities of intelligent automated agents for ModSAF (Laird et al, 1995, Soar/IFOR, 1996). The advantage of this approach is that no IPC is required between the two components, resulting in minimal latency. However, the disadvantages of this approach are 1) the lack of concurrency between the two components may result in an application with unacceptable performance due to the real-time demands of the DIS component, and 2) the complexity resulting from the combination of two control flows into a single flow.

The second approach is to use system-provided communication facilities for the required interactions. One of the most common interprocess communication paradigms, RPC, is supported by many underlying systems, meaning that special non-standard interfaces to the external applications need not be constructed. The advantage of this method is that the application components can be distributed across multiple heterogeneous platforms. The disadvantages of this method include additional latency introduced by local-area or wide-area network connections between the application components and the possible degradation of performance in the face of network or host failures. In addition, distributed approaches require greater attention to security.

An example of this architectural style is our interface between ModSAF and the COMPASS system (SAIC, 1995, SAIC 1996). COMPASS is the Common Operational Modeling, Planning and Simulation Strategy. The ModSAF-COMPASS interface allows planners to evaluate collaborative mission plans by using ModSAF simulation capabilities. Implementing this interface involved creating a new ModSAF library that uses the API provided by the COMPASS client library. This client library encapsulates the application-level protocol that enables the exchange of overlay and route information between the COMPASS server and ModSAF via RPC. Additional details are given in Section 4.1.

The third integration approach is to utilize DIS protocol data units (PDUs) to exchange information between application components. To do this, new PDU types must be defined and additional software to generate and process these new PDUs must be written for both systems. In the HLA, this approach would use the RTI to communicate between the two processes. The advantages of this approach are that the application components can be distributed and that the DIS application's basic flow of control need not be disrupted. Some disadvantages of this method are that the integration is more complex than the first two approaches, and that additional software is required to make it reliable.

The DIS PDU approach is the one we have taken in interfacing the Virtual Commander (VCDR) learning functions to ModSAF. VCDR is a Lisp program that uses Disciple to perform learning for a ModSAF instructable agent (Hieb, Tecuci & Pullen 1996). In this case, we constructed an interface using experimental PDUs that transport the data necessary for the instruction process. Existing ModSAF library functions are used to send/receive the new PDU and register the appropriate callback. A DIS front end was implemented in Lisp for Disciple. The front end call functions to send and receive PDUs during the instruction process. Additional details of this integration are given in Section 4.2.

4.1 ModSAF-COMPASS Interface

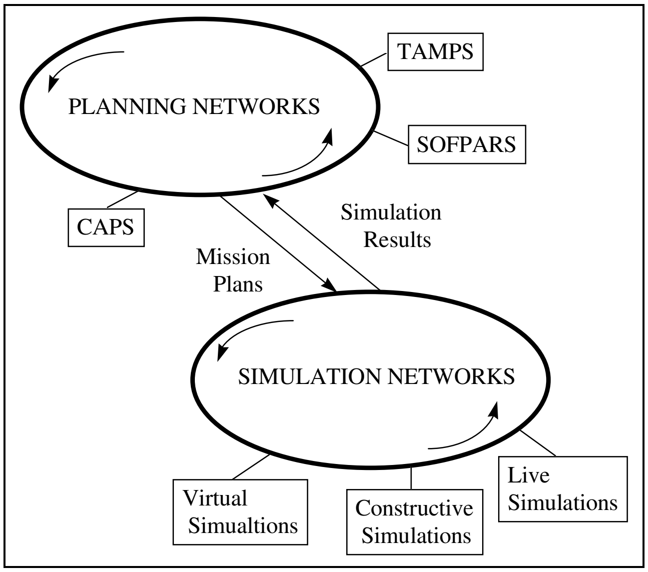

Figure 4: Mission Planning and Evaluation with COMPASS

In this section, we discuss the interface developed between COMPASS and ModSAF. COMPASS is a collection of distributed collaborative planning (DCP) services that allow legacy mission planning systems to interoperate with other COMPASS-compliant mission planning systems and DIS simulation systems. COMPASS is implemented as a collection of servers providing DCP session management, shared map overlay exchange, composite route preview, and DIS-based composite mission preview. COMPASS currently supports several air operations planning systems, such as the Coordinated Adaptive Planning System (CAPS), the Special Operations Force Planning and Rehearsal System (SOFPARS), and the Tactical Air Mission Planning System (TAMPS).

ModSAF is a DIS-compliant simulation system consisting of a graphical user interface, one or more simulators, and an optional logger (Ceranowitz, 1994). ModSAF simulates entity-level actions and provides realistic terrain reasoning capabilities. ModSAF uses the DIS protocols to share information with other DIS-compliant simulation systems (Loral, 1995).

The interface constructed illustrates the potential for linking virtual simulations such as ModSAF to the COMPASS architecture. This linkage allows a collection of heterogeneous mission planning systems to access ModSAF (through COMPASS) in order to evaluate collaborative mission plans. Figure 4 shows the iterative mission planning and evaluation process using heterogeneous mission planners, the COMPASS servers, and simulators. Users of heterogeneous mission planning systems can collaborate, each using a system that is familiar to them, and create composite missions. These missions are then exported to the COMPASS servers and retrieved and executed by a variety of simulation systems. The simulation results can be fed back in the form of DIS PDUs and viewed by the participants in the DCP session. The results can be evaluated and the mission restructured to take advantage of the feedback obtained from the simulation.

4.1.1 Design Issues

We will now consider the design of the COMPASS-ModSAF interface in the context of the issues raised in section 3: data representation, degree of process distribution, interprocess communication, and quality of service requirements.

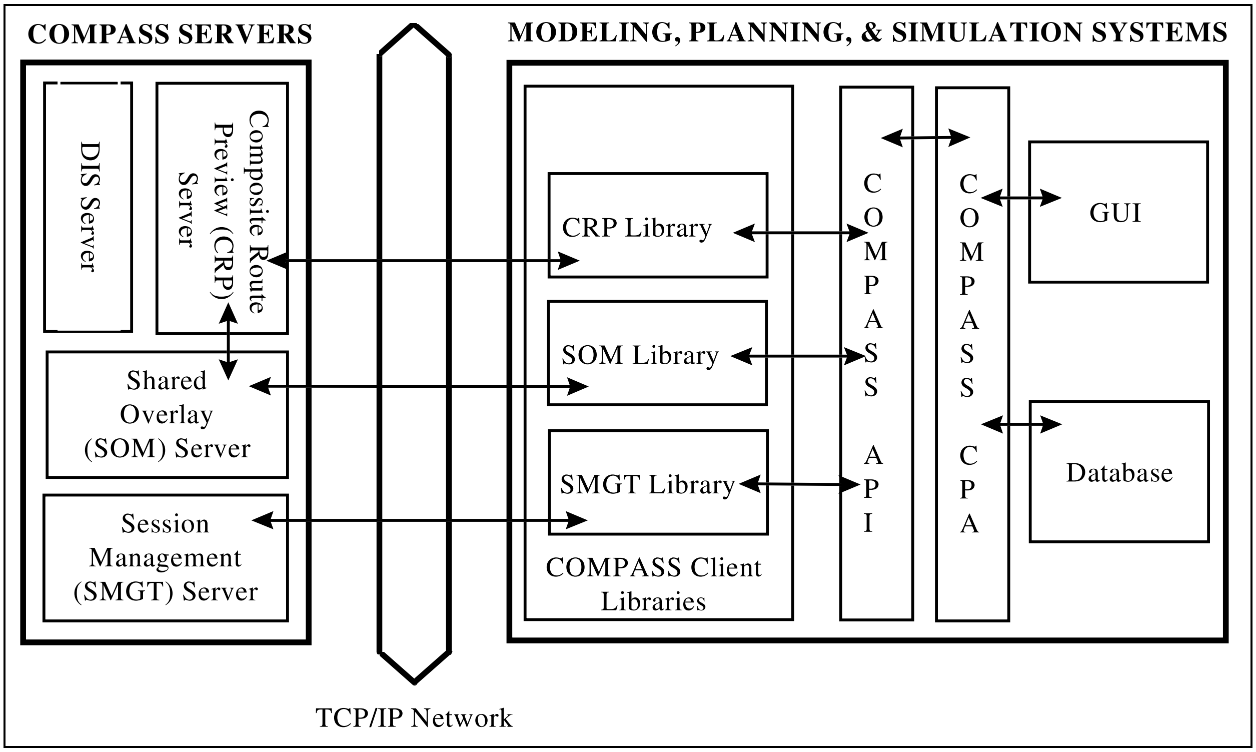

Figure 5: COMPASS Interface Architecture

As stated above in section 3.3, the choice of interprocess communication mechanism is very much based on the prior choice of the degree of process distribution. In this case, binaries for linkable libraries that implement the API function described in the COMPASS ICD were made available to us. The API functions provide an RPC facility, allowing interaction with the COMPASS servers without explicit use of message passing primitives.

The Interface Control Document (ICD) for the COMPASS Architecture (SAIC 1995) provides a detailed description of a data structure and representation to be used by COMPASS-compliant applications when communicating with the COMPASS servers. This document also describes the API through which application programs must communicate with the COMPASS servers. As these API functions expect arguments in a specific data format, the choice of data representation was predetermined. Similarly, the document describes an interface architecture, shown in Figure 5, in which the

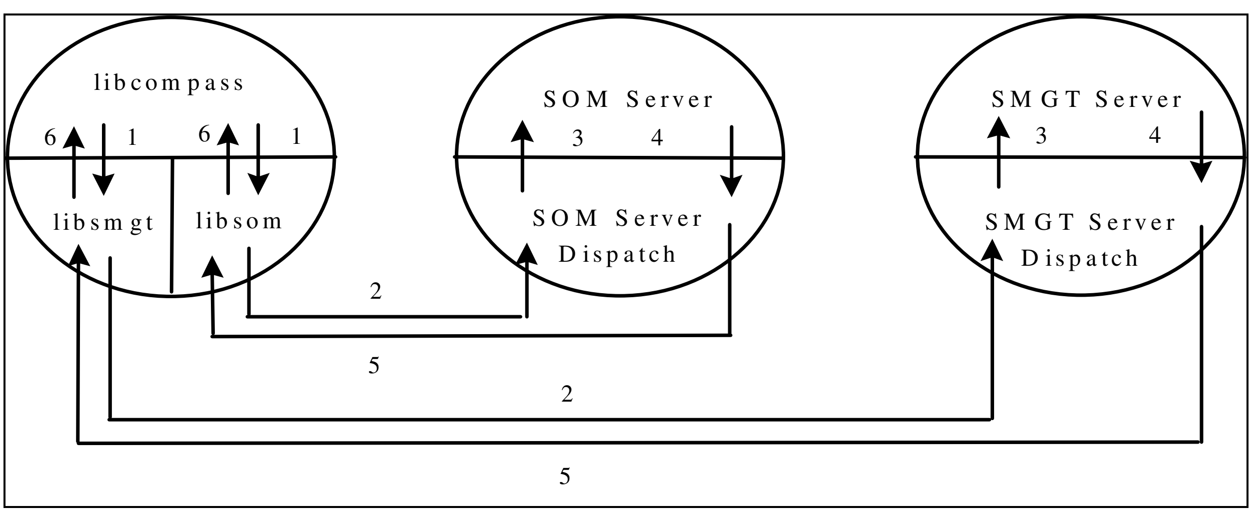

Figure 6: Remote Procedure Call in COMPASS

COMPASS servers are separate processes and the collaborative planning applications communicate with them through the COMPASS libraries. Therefore, we chose the architectural approach in which ModSAF and COMPASS reside on separate hosts and communicate via a network.

An example of RPC as used in COMPASS is shown in Figure 6. Two of the COMPASS libraries, *libsom* and *libsmgt*, are linked with ModSAF at compilation time. When a ModSAF user invokes a *libcompass* function requiring a remote function be executed on one of the COMPASS servers, e.g. the SOM server, (1) *libcompass* calls a function in *libsom.* This function (2) is responsible for packaging the arguments for the server routine into one or more RPC messages along with a code indicating which remote function is to be executed and sending the message to the SOM server. When the server receives the RPC message (3), it unpacks it and determines which function to call based on the aforementioned code. The function is called with the arguments received in the message. When the server function has completed (4), the results are passed back to the dispatch function in the SOM server. (5) The results are packed into a reply message and sent back to *libsom*. (6) Upon receiving the reply message, *libsom* unpacks the results and returns them to its caller, a function in *libcompass.* Thus, from the perspective of *libcompass*, the remote procedure has been invoked through a simple function call without the explicit use of message-passing primitives. As shown Figure 6, the same process occurs when remote functions are executed on the SMGT server.

In the absence of multiple threads of execution clients block or poll for results after sending an RPC request to the server. This makes RPC difficult to integrate into a real-time application's existing control flow, especially if the application needs to perform other tasks (such as simulating entities) and is instead waiting for data, effectively freezing the entire simulation. To address this problem, the client stub may timeout after not receiving a reply in a specified period of time, causing the RPC to fail. Still, although RPC serves to abstract away message passing primitives, it cannot be said that the message passing is truly transparent to the client application, as new forms of failure are now possible beyond those that could occur if the server procedure were being called locally.

Our interface design addresses the quality of service issues described in section 3.4. Since ModSAF is a DIS application, low latency is a very important requirement. Communication between ModSAF and the COMPASS servers takes time, especially if the COMPASS servers are located on a remote subnet. Because the interaction between ModSAF and COMPASS is used to establish a ModSAF simulation

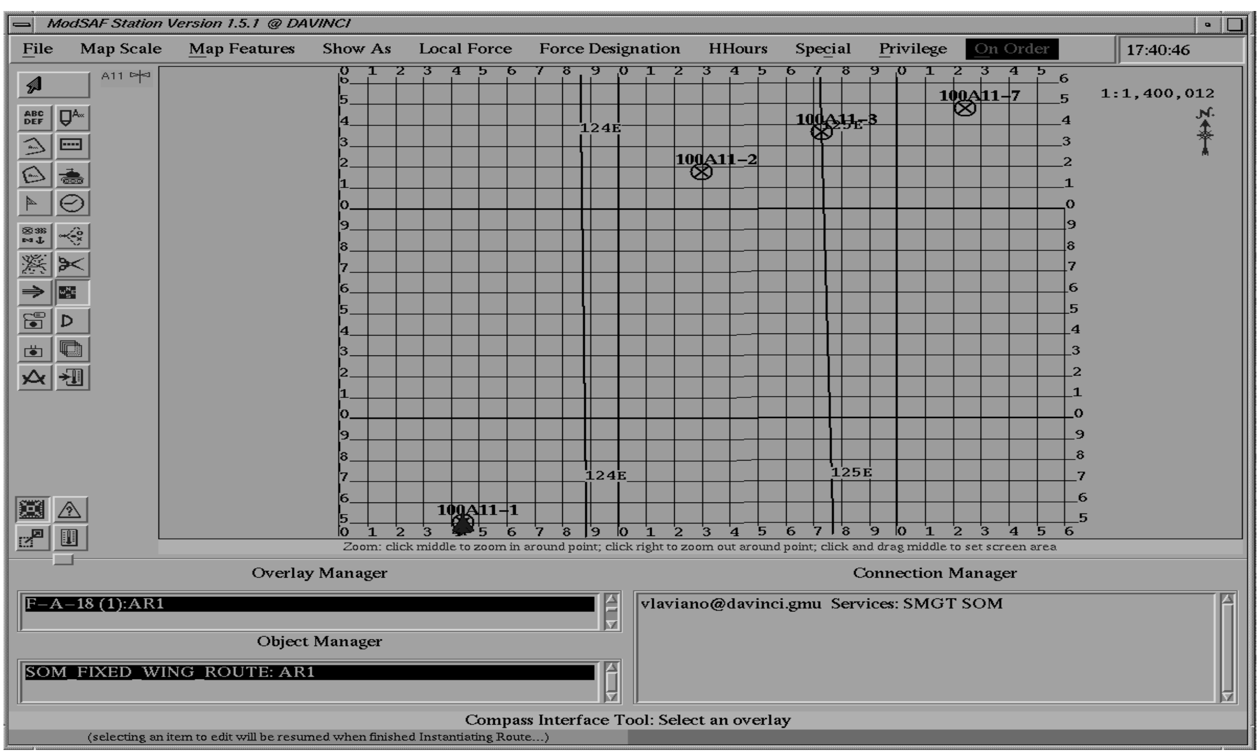

Figure 7: COMPASS Interface Tool

scenario, it takes place before any simulation activity occurs. In this case, additional latency is not a consideration because the system need not meet real-time latency requirements.

In this interface, we did not consider communication reliability or security issues because all message passing was encapsulated within the COMPASS libraries that were provided to us. Both the COMPASS servers and ModSAF are robust. ModSAF has facilities to shift entities from host to host if a host goes down via the PO database, and COMPASS maintains similar databases from which servers can recover if they crash.

4.1.2 Architecture

We now describe the architecture of the COMPASS-ModSAF interface as pictured in Figure 8. ModSAF has been designed with modularity and ease of expansion in mind, and for this reason, it consists of over 450 libraries. We decided that the best way to add to the ModSAF architecture without disrupting existing components was to create a new COMPASS interface library, *libcompass*, to isolate changes within this library

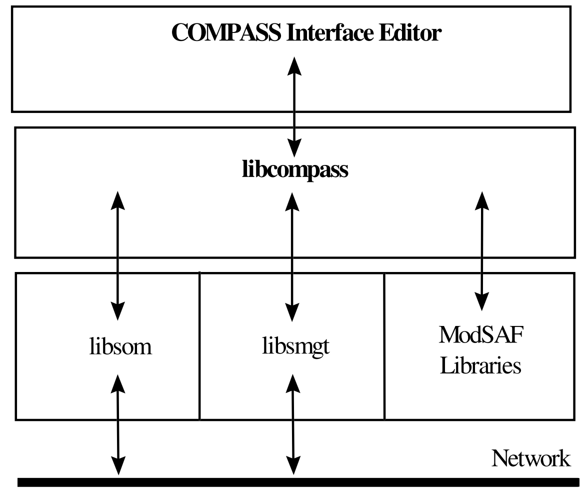

Figure 8: ModSAF-COMPASS Interface Architecture

The ModSAF GUI provides several editors that allow the user to perform various tasks such as creating new entities, and adding graphical images to the map overlay. Among the additions to ModSAF found in *libcompass* is a new ModSAF editor, developed in Motif, that is accessed by pressing a new COMPASS button on the ModSAF GUI, as shown in Figure 7. The editor provides a list of users who are currently participating in the DCP session and a list of other COMPASS services these users are currently subscribed to. The editor also provides a list of shared overlays that currently reside in the COMPASS Shared Overlay Management (SOM) server's database. This information is obtained by *libcompass* through the COMPASS API functions provided in the COMPASS libraries, *libsmgt and libsom*, compiled with ModSAF. The editor allows a user to select an overlay. This causes *libcompass* to obtain a list of route objects contained within the selected overlay from the SOM server. The user can then click on the route object corresponding to the route to be simulated.

A COMPASS route consists of an entity description, a set of waypoints comprising the route, flight characteristics which indicate when the entity arrives at each waypoint, and a set of events (each of which is associated with a waypoint).

Once a route is selected, *libcompass* maps the COMPASS entity described in the route to the most appropriate available ModSAF entity. This is done via a mapping function that makes use of an extensible mapping file that can be modified without recompilation of ModSAF. After the mapping is complete, the coordinates of the entity's starting location are converted from latitude/longitude format to topocentric coordinates (TCC) for use by ModSAF. The ModSAF entity is then created at the initial waypoint though the utility functions in *libunitutil*, another ModSAF library.

Then, all other waypoints are converted to TCC format and placed in ModSAF's persistent object (PO) database by using the functions in *libpo,* a ModSAF library relating to the PO database. A chain of tasks is then created, by using the utility functions in *libtaskutil* along with some additional code, tasking the entity to fly from waypoint to waypoint, one after another. This task is placed "On Order", meaning that simulation of the route does not begin immediately, but is pending and will begin at the time the user invokes the task.

4.2 Virtual Commander Interface

The second case study we present is an implementation of an interface between ModSAF and Disciple (Tecuci, 1988) for the Virtual Commander

System. Disciple is a machine learning system written in Lisp that runs on a typical Unix workstation. It employs multistrategy learning, combining explanation-based, analogical, and example-based methods with knowledge acquisition to learn from a subject matter expert (SME).

As CGF technology advances sufficiently to allow CFORs to be created and deployed, knowledge acquisition will become a critical issue. The Virtual Commander system is an approach to solving the problem of knowledge acquisition for CFORs. VCDR is an instructable agent which utilizes Programming by Demonstration (Cypher, 1993) and Machine Learning techniques to allow a SME to teach an agent. Programming by Demonstration systems give an end user the ability to create programs by demonstrating their action.

We have prototyped this approach with the Captain system (Hieb et. al. 1995) which consisted of a file-level integration of the apprenticeship learning system Disciple and ModSAF. In order to integrate Disciple fully with ModSAF we expanded upon the Captain interface in two areas. Rather than utilizing a textual interface for the training/learning process, we are developing VCDR ModSAF editors that allow the use of the terrain map interface (plan view display) in ModSAF. To convey the data from the ModSAF editors, we are interfacing the learning functions of Disciple to ModSAF using an experimental DIS protocol data unit (PDU).

4.2.1 Design Issues

Our design and implementation is driven by three significant factors: real-time performance, data representation, and control of the learning cycle. The initial concern addressed in this implementation was the difficulty of integrating an application that relies on real-time delivery of data (ModSAF) and one that does not (Disciple).

The performance or latency issue was important since ModSAF, a DIS application, typically requires real-time transmission of data. The ModSAF/Disciple interface does not require this type of service. In fact, due to the algorithms involved in the various learning phases of Disciple, it would be difficult to achieve real-time response from Disciple. Therefore, the concern about real-time performance is how to construct a non-real-time communication that does not cause perceptible delay in ModSAF. We implemented this non-real-time communication by

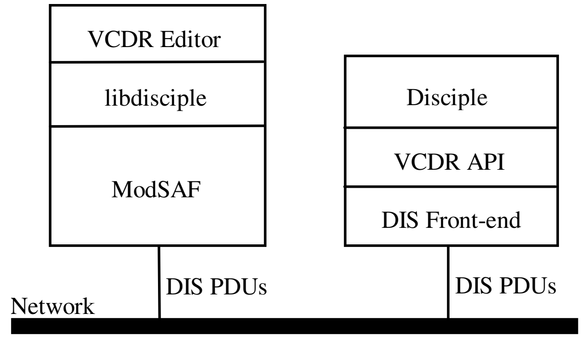

Figure 9: Virtual Commander Architecture

distributing the processes over a network allowing each process to utilize the full resources of their respective workstations, since both applications are very computationally intensive.

To construct this interface we delivered messages via DIS PDUs. Integrating this message passing scheme into ModSAF was done using the extensive libraries that ModSAF makes available, namely the PDU API and PDU Processing libraries. By using these libraries our messaging was integrated into the ModSAF control structure such that ModSAF executed normally. In other words, ModSAF does not block while waiting for our message, which would disrupt the flow of ModSAF's execution. This was achieved by using functionality provided by the ModSAF library, *libpduproc,* to register a callback to handle the message upon ModSAF receiving it. When ModSAF receives one of these messages it will call the registered handler function. Integrating a message passing facility into Disciple required implementing a front-end to Disciple in C that was callable from Lisp via its foreign function interface (Harlequin, 1994).

A second issue was the format in which the data would be communicated between the applications. While ModSAF stores its data in a Persistent Object (PO) database, Disciple represents its data in the form of a semantic network. In order for the two systems to communicate, a common format was constructed, described in the PDU in Table 2. The message within the PDU is data extracted from either the PO database or from Disciple's semantic network. The experimental PDU format was developed and introduced into ModSAF. By utilizing DIS PDUs to perform the messaging for the VCDR system, we can use a messaging infrastructure already established for DIS.

Description	Size (bits)
DIS 2.03 header	96
Message ID	32
Fragment number	32
Learning phase	32
Message type	32
Message length	32
Message	8*length

Table 2: Virtual Commander Experimental DIS 2.03 PDU Format

The final issue was the need to move the control interface from Disciple to ModSAF. Originally, the steps taken by the SME to teach Disciple were selected from a menu presented by the Disciple software in Lisp, based on which phase of the learning process was being executed. When introducing the GUI editor into ModSAF, the control mechanism was transferred to ModSAF. A new library, *libdisciple*, was built into ModSAF to implement the control features needed. The resulting interface is much more natural to its intended user.

4.2.2 Architecture

Figure 9 depicts the architecture of Virtual Commander consisting of ModSAF, Disciple, a new library for ModSAF called *libdisciple*, a DIS front-end for Disciple, and an API for Disciple.

The front-end developed for Disciple to send and receive DIS PDUs implements three API functions: *vcdr_init()*, *GetVCDRPDU()*, and *SendVCDRPDU()* as shown in Figure 10. The initialize function is called once when Disciple is started. It creates a socket with a well-known port to receive incoming messages. The *GetVCDRPDU()* function performs a blocking read of the socket. This is the state Disciple is usually in, waiting for the next request from the SME via ModSAF. Upon receiving a PDU the message contents are returned to Disciple. The *SendVCDRPDU()* function, on the other hand, packages and sends a message as a VCDR PDU to ModSAF.

In the newly created ModSAF library, *libdisciple*, several functions were added to perform the VCDR functions. These components and their interaction is shown in Figure 11. An editor was created, and is initialized in *discip_init_gui(),* for the GUI to accept commands from the SME. The learning data and phase of the learning cycle is packaged in the PDU and sent directly to Disciple's socket as a unicast message by *discip_done()*. (This is a departure from the use of ModSAF-provided functionality of broadcasting or multicasting.)

When the VCDR PDU is received by ModSAF it passes it to the callback function, *vcdr_pdu_handler()*, that was registered earlier. From this point, one of several procedures is called based on the value of the phase field in the PDU. If appropriate, the plan view display or the Disciple editor in ModSAF will be updated to reflect the

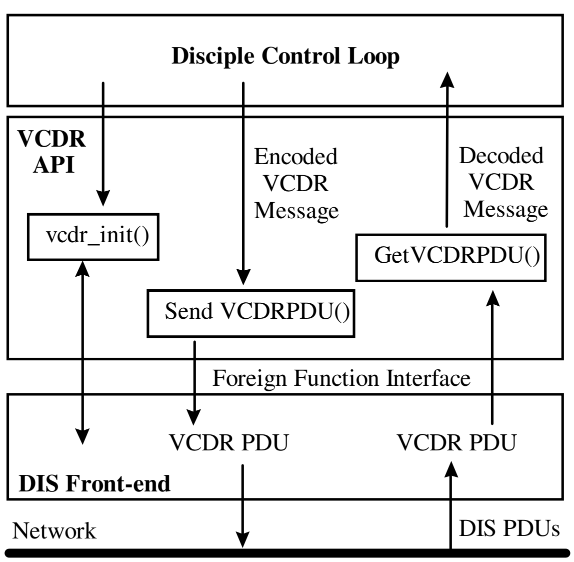

Figure 10: Disciple VCDR Components

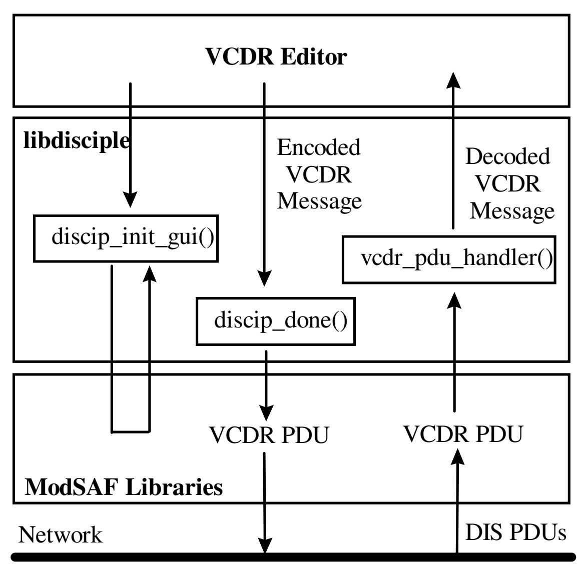

Figure 11: ModSAF VCDR Components

message in the last VCDR PDU received.

Although we have considered the fact that the messages between ModSAF and Disciple need to be delivered reliably, the current implementation does not guarantee that. It is common in situations like this for the application to take on the responsibility of ensuring successful transmissions where the network protocol is unreliable, such as with UDP. In the future, the use of a protocol such as the proposed Selectively Reliable Transport Protocol (SRTP) (Pullen and Laviano, 1995), which provides varying levels of reliability would allow the application to choose the degree of reliability necessary.

5. Related Work

With the advent of Command Forces (CFORs) and Intelligent Forces (IFORs) it has become necessary to add many more external reasoning and learning processes to their agent architectures. Some of these methods are already under development

MITRE is developing a Command and Control Simulation Interface Language (CCSIL) to provide a common language for CFORs (Salisbury et. al. 1995). CCSIL represents a language, a set of vocabulary and terminology, that is used to communicate between command entities and subordinate entities or vehicles. The interface is implemented by transporting the CCSIL messages within a DIS Signal PDU. This corresponds to the third approach that was described earlier. Additional effort is necessary to interface existing applications to DIS Simulations.

The Soar/IFOR research group also is developing an architecture to provide intelligent forces. The interface of Soar and ModSAF represents the first approach we described earlier. Soar is a general cognitive architecture that provides intelligent agents. The Soar architecture combines with the graphical interface, network interface, and scenario creation and execution tools of ModSAF to provide an architecture for intelligent forces. Currently, the group is integrating the CCSIL command language (Hill 1996).

The RTI of the HLA presents another integration approach. The RTI will provide a set of services which facilitates integration of simulations, C^4I systems, and engineering models. The RTI will utilize a more reliable transport mechanism than DIS and the RTI APIs will be more standardized, which will eliminate some of the disadvantages of a DIS interface, which lacks a standardized API.

6. Conclusions

The choice of interface approach may be determined partially by external factors (as with COMPASS, where a client library was provided that facilitated RPC calls). However, there is always latitude in some aspects of the implementation. The crucial factors that will determine the best choice are 1) Data Representation; 2) Distribution of Processes and Associated Communication Issues; and 3) Transport Mechanism. Consideration of the issues raised above will allow a more informed design when constructing the interface between a DIS application and an external process as demonstrated in the case studies above.

7. Acknowledgments

This research was conducted in the Center for Excellence in Command, Control, Communications and Intelligence and the Computer Science Department at George Mason University. Work on ModSAF-COMPASS was sponsored in part by the Defense Modeling and Simulation Office (DMSO) under DISA contract DCA100-91-C-0033. The authors thank Dr. Gheorghe Tecuci for his assistance with the Disciple Learning System and the COMPASS group at SAIC, Inc. for their cooperation and assistance.

8. References

Birrell, A., and Nelson, B. (1984). "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, Vol. 2, pp. 39-59.

Calvin, J. and Weatherly, R. (1996). "An Introduction to the High Level Architecture (HLA) Runtime Infrastructure (RTI)," *14th Workshop on Standards for the Interoperability of Distributed Simulations,* paper 96-14-103, May 1996.

Ceranowicz A. (1994). "ModSAF Capabilities," *4th Conference on Computer Generated Forces and Behavior Representation*, May, Orlando, Florida.

Cypher, A. (Ed.). (1993). "Watch What I Do: Programming by Demonstration," *MIT Press*, Cambridge, MA.

Harlequin Group Limited, The (1994). *LispWorks User's Guide*.

Hieb, M.R., Tecuci, G., Pullen, J.M., Ceranowicz, A., Hille, D. (1995). "A Methodology and Tool for

Constructing Adaptive Command Agents for Computer Generated Forces," *5th Conference on Computer Generated Forces and Behavioral Representation*.
Hieb, M.R., Tecuci, G., Pullen, J.M. (1996). "Virtual Commander – An Instructable ModSAF Agent," *6th Conference on Computer Generated Forces and Behavioral Representation*.
Hill, Jr., R. (1996). *Intelligent Forces for Simulated Environments: Contractor's Progress Status & Management Report*, May, 1996 <URL: http://www.isi.edu/soar/ifor/reports/monthly/may96.html>
IEEE Standard 1278.2 (1995). *Standard for Distributed Interactive Simulation – Communication Services and Profiles*, Institute of Electrical and Electronics Engineers, Inc.
Laird, J. E., et al. (1995). "Simulated Intelligent Forces for Air: The SOAR/IFOR Project 1995," *5th Conference on Computer Generated Forces and Behavioral Representation*.
Loral Advanced Distributed Simulation (1995). *ModSAF 2.0 Developer's Kit*.
Pullen, J.M., and Laviano, V.P. (1995). "A Selectively Reliable Transport Protocol for Distributed Interactive Simulation," *13th DIS Workshop on Standards for the Interoperability of Distributed Simulations*, Paper 95-13-102.
SAIC (1995), *Interface Control Document (ICD) for the COMPASS Architecture*, for NRaD.
SAIC (1996), *Common Operational Modeling, Planning and Simulation Strategy.* <URL: http://prw7.saic.com>
Salisbury, M.R., Booker, L.B., Seidel, D.W. and Dahmann, J.S. (1995). "Implementation of Command Forces (CFOR) Simulation," *5th Conference on Computer Generated Forces and Behavioral Representation*.
Soar/IFOR Research Group (1996). *Soar/IFOR Documentation.* <URL: http://krusty.eecs.umich.edu/ifor>
Sun Microsystems. (1987). "XDR: External Data Representation Standard," *RFC 1014*, June, 1987.
Tecuci, G. (1988). "DISCIPLE: A Theory, Methodology and System for Learning Expert Knowledge," Ph.D. Thesis, University of Paris South.
Tecuci, G., Hieb M.R., Hille D. & Pullen J.M.(1994). "Building Adaptive Autonomous Agents for Adversarial Domains," *Proceedings of the AAAI 94 Fall Symposium — Planning and Learning: On To Real Applications*.
Yang, Z. and Duddy, K. (1996) "CORBA: A Platform for Distributed Object Computing," *Operating Systems Review,* Vol. 30, No. 2, pp. 4-31.

9. Authors' Biographies

Elizabeth White is an Assistant Professor at George Mason University (GMU). She received M.S. and Ph.D. degrees in Computer Science from the University of Maryland in 1990 and 1996, respectively, and a B.S. degree from the College of William and Mary in 1985.

Ken Frosch is a Graduate Research Assistant in the C^3I Center at GMU. He is currently on leave of absence from Lockheed Martin Federal Systems where he has been a Systems Engineer since 1991. He received his B.S. in Computer Science and Mathematics from the University of Wisconsin - Madison in 1991.

Vince Laviano is a Graduate Research Assistant in the C^3I Center at GMU. He received his B.S. in Computer Science from GMU in 1995. He has published papers in the area of distributed simulations.

Michael Hieb received a Ph.D. in Information Technology at GMU in 1996. Dr. Hieb is currently working for SAIC on the Multiple Reconfigurable C^4I Interface (MRCI) project. He has published over 20 papers in the areas of learning agents, multistrategy learning and knowledge acquisition.

J. Mark Pullen is Associate Professor of Computer Science and member of the C^3I Center at GMU. Dr. Pullen was with the Defense Advanced Research Projects Agency (DARPA) from 1986 to 1992, where he was Program Manager for Advanced Computing, Networking and Distributed Simulation.