Integrating CLIPS Applications into Heterogeneous Distributed Systems

Richard M. Adler

Symbiotics, Inc.
875 Main Street Cambridge, MA 02139 (617) 876-3633

Abstract. SOCIAL is an advanced, object-oriented development tool for integrating intelligent and conventional applications across heterogeneous hardware and software platforms. SOCIAL defines a family of "wrapper" objects called *Agents*, which incorporate predefined capabilities for distributed communication and control. Developers embed applications within Agents and establish interactions between distributed Agents via non-intrusive message-based interfaces. This paper describes a predefined SOCIAL Agent that is specialized for integrating CLIPS-based applications. The Agent's high-level Application Programming Interface supports *bidirectional* flow of data, knowledge, and commands to other Agents, enabling CLIPS applications to initiate interactions autonomously, and respond to requests and results from heterogeneous, remote systems. The design and operation of CLIPS Agents is illustrated with two distributed applications that integrate CLIPS-based expert systems with other intelligent systems for isolating and managing problems in the Space Shuttle Launch Processing System at the NASA Kennedy Space Center.

INTRODUCTION

The central problems of developing heterogeneous distributed systems include:

- communicating across a distributed network of diverse computers and operating systems in the absence of uniform interprocess communication services;
- specifying and translating information (i.e., data, knowledge, commands), across applications, programming languages and development shells with incompatible native representational models, programmatic data and control interfaces;
- coordinating problem-solving across heterogeneous applications, both intelligent and conventional, that were designed to operate as independent, standalone systems.
- accomplishing these integration tasks *non-intrusively*, to minimize re-engineering costs for existing systems and to ensure maintainability and extensibility of new systems.

SOCIAL is an innovative distributed computing tool that provides a unified, object-oriented solution to these difficult problems (Adler 1991). SOCIAL provides a family of "wrapper" objects called *Agents*, which supply predefined capabilities for distributed communication, control, and information management. Developers embed applications in Agents, using high-level, message-based interfaces to specify interactions between programs, their embedding Agents, and other application Agents. These message-based Application Programming Interfaces (APIs) conceal low-level complexities of distributed computing, such as network protocols and platform-specific interprocess communication models (e.g., remote procedure calls, pipes, streams). This means that distributed systems can be developed by programmers who lack expertise in system-level communications (e.g., Remote Procedure Calls, TCP/IP, ports and sockets, platform-specific data architectures). Equally important, SOCIAL's high-level APIs enforce a clear separation between

application-specific functionality and generic distributed communication and control capabilities. This partitioning promotes modularity, maintainability, extensibility, and portability.

This paper describes a particular element of the SOCIAL development framework called a CLIPS Knowledge Gateway Agent. Knowledge Gateways are SOCIAL Agents that are specialized for integrating intelligent systems implemented using standardized AI development shells such as CLIPS and KEE. Knowledge Gateways exploit object-oriented inheritance to isolate and abstract a shell- and application-independent model for distributed communication and control. Particular subclasses of Knowledge Gateway Agents, such as the CLIPS Gateway, add a dedicated high-level API for transporting information and commands across the given shell's data model and data and control interfaces. To integrate a CLIPS application, a developer simply (a) creates a subclass of the CLIPS Gateway Agent class, and (b) specializes it using the high-level CLIPS Gateway API to define the desired message-based interactions between the program, its embedding Gateway, and other application Agents.

The remainder of the paper is divided into three major parts. The first section provides an overview of SOCIAL, emphasizing the lower-level distributed computing building blocks underlying Gateway Agents. The second section describes the architecture and functionality of Knowledge Gateway Agents. Structures and behaviors specific to the CLIPS Gateway are used for illustration. The third section presents two examples of SOCIAL applications that integrate CLIPS-based expert systems with other intelligent systems for isolating and managing problems in the Space Shuttle Launch Processing System at the NASA Kennedy Space Center.

OVERVIEW OF SOCIAL

SOCIAL consists of a unified collection of object-oriented tools for distributed computing, depicted below in Figure 1. Briefly, SOCIAL's predefined distributed processing functions are bundled together in objects called *Agents*. Agents represent the active computational processes within a distributed system. Developers assemble distributed systems by (a) selecting Agents with suitable integration behaviors from SOCIAL's library of predefined Agent classes, and (b) using dedicated APIs to embed individual application elements within Agents and to establish the desired distributed interactions among their embedded applications. A separate interface allows developers to create entirely new Agent classes by combining (or extending) lower-level SOCIAL elements to satisfy unique application requirements (e.g., supporting a custom, in-house development tool). These new Agent types can be incorporated into SOCIAL's Agent library for subsequent reuse or adaptation. The following subsections review SOCIAL's major subsystems.

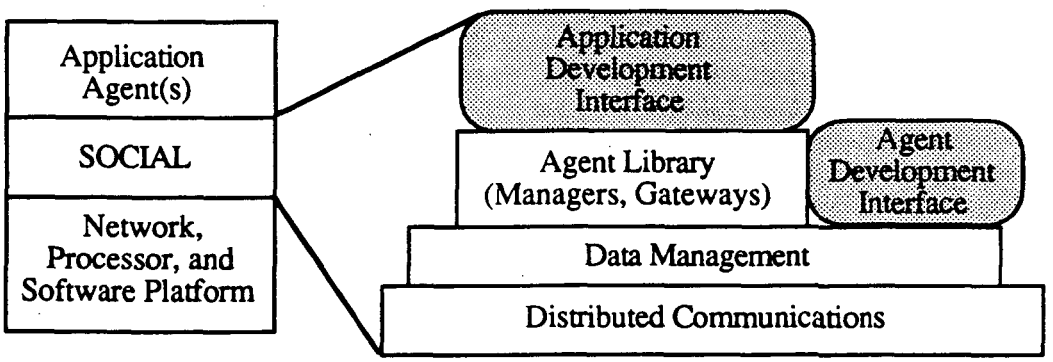

Figure 1. Architecture of SOCIAL

Distributed Communication

SOCIAL's distributed computing utilities are organized in layers, enabling complex functions to be built up from simpler ones. The base or substrate layer of SOCIAL is the MetaCourier tool, which provides a high-level, modular distributed communications capability for passing information between applications based on heterogeneous languages, platforms, operating systems, networks, and network protocols (Symbiotics 1990). The basic Agent objects that integrate software programs or information resources are defined at SOCIAL's MetaCourier level. Developers use the MetaCourier API to pass messages between applications and their embedding Agents, as well as among application Agents. Messages typically consist of (a) commands that an Agent passes directly into its embedded application, such as database queries or calls to execute signal processing programs; (b) data arguments to program commands that an Agent might call to invoke its embedded application; or (c) symbolic flags or keywords that signal the Agent to invoke one or another fully preprogrammed interactions with its embedded application.

For example, a high-level MetaCourier API call issued from a local LISP-based application Agent such as *(Tell :agent 'sensor-monitor :sys 'Symb1 '(poll measurement-Z))* transports the message contents, in this case a command to poll measurement-Z, from the calling program to the Agent *sensor-monitor* resident on platform *Symb1*. The Tell function initiates a message transaction based on an asynchronous communication model; once the message is issued, the application Agent can immediately move on to other processing tasks. The MetaCourier API also provides a synchronous "Tell-and-Block" message function for "wait-and-see" processing models.

Agents contain two procedural methods that control the processing of messages, called :in-filters and :out-filters. In-filters parse incoming messages, based on a contents structure that is specified when the Agent is defined . After parsing a message, an :in-filter typically either invokes the Agent's embedded application, or passes the message (which it may modify) on to another Agent. The MetaCourier semantic model entails a directed acyclic computational graph of passed messages. When no further passes are required, the :in-filter of the terminal Agent runs to completion. This Agent's :out-filter method is then executed to prepare a message reply, which is automatically returned (and possibly modified) through the :out-filters of intermediate Agents back to the originating Agent. Developers specify the logic of :in-filters and :out-filters to meet their particular requirements for application interactions.

A MetaCourier runtime kernel resides on each application host. The kernel provides (a) a uniform message-passing interface across network platforms; and (b) a scheduler for managing messages and Agent processes (i.e., executing :filter methods). Each Agent contains two attributes (slots) that specify associated Host and Environment objects. These MetaCourier objects define particular hardware and software execution contexts for Agents, including the host processor type, operating system, network type and address, language compiler, linker, and editor. The MetaCourier kernel uses the Host and Environment associations to manage the hardware and software platform specific dependencies that arise in transporting messages between heterogeneous, distributed Agents (cf. Figure 2).

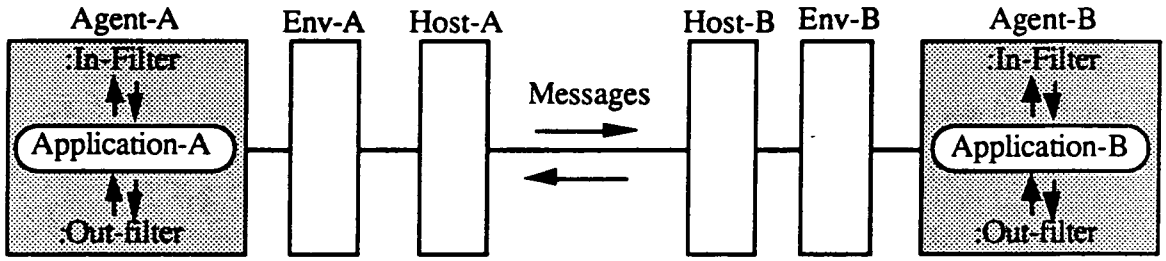

Figure 2. Operational Model of MetaCourier Message-Passing

MetaCourier's high-level message-based API is basically identical across different languages such as C, C++, and Lisp. Equally important, MetaCourier's communication model is also symmetrical or "peer-to-peer." In contrast, client-server computing, a popular alternative model for distributed communication, is asymmetric: clients are active (i.e., only clients can initiate communication) while servers are passive. Moreover, while multiple clients can interact with a particular server, a specific client process can only interact with a single (hardwired) server. MetaCourier's communication model eliminates these restrictions on interprocess interactions.

Data Specification and Translation

A major difficulty in getting heterogeneous applications and information resources to interact with one another is the basic incompatibility of their underlying models for representing data, knowledge, and commands. These problems are compounded when applications are distributed across heterogeneous computing platforms with different data architectures (e.g., opposing byte ordering conventions).

SOCIAL applies a uniform "plug compatible" approach to these issues. This approach consists of two elements, a design methodology and a set of tools to support that methodology. SOCIAL defines a uniform application-independent information model. In the case of Knowledge Gateways, the information model defines a set of common data elements commonly used in intelligent systems, including facts, fact-groups, frames/objects, and rules. SOCIAL's Data Management Subsystem (DMS) provides tools (a) for defining canonical structures to represent these data types, and (b) for accessing and manipulating application-specific examples of these structures. These tools are essentially uniform across programming languages. Equally important, DMS tools encode and decode basic data types transparently across different machine architectures (e.g., character, integer, float).

Developers use SOCIAL's DMS tools to construct intermediate-level APIs for the Gateway Agent class that integrates particular applications. This API establishes mappings between SOCIAL's "neutral exchange" structures and the native representational model for the target application or application shell. For example, the API for the CLIPS Knowledge Gateway Agent translates between DMS frames and CLIPS deftemplates or fact-groups (e.g., Make-CLIPS-fact-group-from-frame Frame-x). Similarly, the KEE Gateway API transparently converts DMS frames to KEE units and KEE units back into frames. If necessary, new DMS data types and supporting API enhancements can be defined to extend SOCIAL's neutral exchange model. This uniform mapping approach simplifies the problem of interconnecting N disparate systems from O(N*N) to O(N), as illustrated in Figure 3.

SOCIAL integrates DMS with MetaCourier to obtain transparent distributed communication of complex data structures across heterogeneous computer platforms as well as across disparate applications: developers embed DMS API function calls within the :in-filter and :out-filter methods of interacting Agents, using MetaCourier messages to transport DMS data structures across applications residing on distributed hosts. DMS API functions decode and encode message contents, mapping information to and from the native representational models of source and target applications and DMS objects. SOCIAL thereby separates distributed communication from data specification and translation, and cleanly partitions both kinds of generic functionality from application-specific processing.

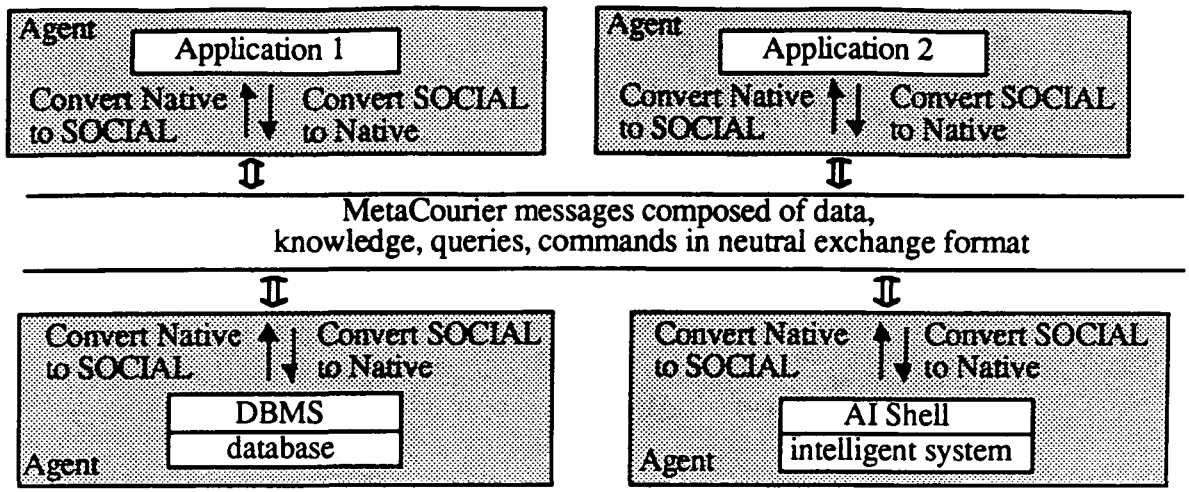

Figure 3. SOCIAL's Plug-Compatible Approach to Managing Heterogeneous Data

Distributed Control (Specialized Agents and Agent APIs)

SOCIAL's third layer of object-oriented tools establishes a library of predefined Agents classes and associated high-level API interfaces that are specialized for particular integration or coordination functionality. MetaCourier and DMS API functions are used to construct Agent API data and control interfaces. These high-level Agent APIs largely conceal lower-level MetaCourier and DMS interfaces from SOCIAL users. Thus, developers typically use specialized Agent classes as the primary building blocks for constructing distributed systems, accessing the functionality of each such Agent type through its dedicated high-level API. If necessary, developers can define new Agent classes and APIs by specializing (e.g., modifying or extending) existing ones.

Currently, SOCIAL's library defines Gateway and Manager Agent classes. Gateways, as noted earlier, simplify the integration of applications based on development tools such as AI shells, DBMSs, CASE tools, 4GLs, and so on. Manager Agents are specialized to coordinate application Agents to work together cooperatively. The HDC-Manager (for Hierarchical Distributed Control) functions much like a human manager, mediating interactions among "subordinate" application Agents and between subordinates and the outside world. The Manager acts as an intelligent router of task requests, based on a directory knowledge base that identifies available services (e.g., data, problem-solving skills) and the application Agents that support them. The Manager also provides a global, shared-memory "bulletin-board." Application Agents are only required to know the names of services within the Manager's scope and the high-level API for interacting with the Manager; they do not need to know about the functionality, structure, location, or even the existence of particular application Agents. The Manager establishes a layer of control abstraction, decoupling applications from one another. This directory-driven approach to Agent interaction promotes maintainability and extensibility, and is particularly valuable in complex distributed systems that evolve as applications are enhanced or added over an extended lifecycle.

KNOWLEDGE GATEWAY AGENTS

Knowledge Gateway Agents combine several important SOCIAL tools and design concepts:

- MetaCourier's high-level, message-based distributed communication capabilities for remote interactions across disparate hardware and software environments;

- DMS data modeling and mapping facilities for transparently moving data, knowledge, and control structures across disparate applications and shells;

- a modular object-oriented architecture that defines a uniform partitioning of integration functionality;

- a non-intrusive design methodology for programming specific, discrete interactions between the application being integrated, its embedding Gateway Agent, and other application Agents;

- extensibility to encompass generalized hooks for security, error management, and session management utilities.

Gateway functional capabilities are distributed across the class hierarchy of Gateways to exploit object-oriented inheritance of behaviors of common utility across Agent subclasses. The partitioning and inheritance of behaviors are summarized below in Figure 4.

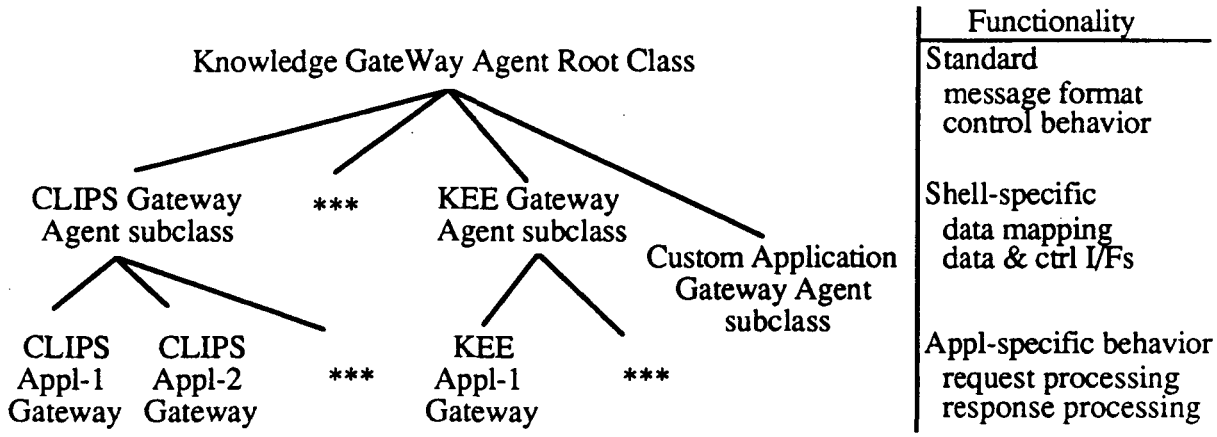

Figure 4. Inheritance of SOCIAL Knowledge Gateway Agent Behaviors

The root Knowledge Gateway Agent, KNOWL-GW, defines the overall structure and functional behavior of all Agent subclasses that are developed to integrate shell-based (or custom) intelligent applications. In particular, KNOWL-GW establishes:

- the uniform MetaCourier/DMS message format structure for communicating with *all* Knowledge Gateway Agents;

- the Agent :in-filter method for parsing and managing incoming messages;

- the Agent :out-filter method for post-processing results;

- default (stub) API methods that are overridden at the Gateway Agent subclass level.

Knowledge-based systems, like most conventional systems, typically function as servers, responding to programmatic (or user) queries or commands. In a server configuration, data and control flow in, while data (results) alone flows out. However, intelligent systems can also initiate control activities autonomously, in response to dynamic, data-driven reasoning. This active (client) role entails "derived" requirements for capabilities to process results returned in response to previous outgoing messages. Since intelligent systems can be configured to act as clients, servers, or play *both* roles within the same distributed application, any generalized integration technology such as Knowledge Gateways must support *bidirectional* flow of data and control.

Accordingly, the generic :in-filter method handles two cases (a) a MetaCourier message coming in from some external application Agent to be handled by the Knowledge Gateway's embedded application, and (b) a message from the embedded application that is to be passed via the Knowledge Gateway Agent to some external application Agent. The KNOWL-GW distinguishes the two cases automatically, based on the message's target Agent. Similarly, the generic :out-filter method handles two cases (a) dispatching the embedded application's reply via the Knowledge Gateway Agent to the external requesting Agent, and (b) processing the response from an external application Agent to a message passed by the Gateway from its embedded application and injecting it back into the embedded application via the shell. This simple dual logic in the :filter methods enables Gateway Agents to function as clients or servers, as required by particular messages.

KNOWL-GW also establishes (a) a uniform, top-level Gateway API; and (b) a uniform model for applying or invoking this API in the filter methods. Specifically, the KNOWL-GW Agent class establishes stub versions of the top-level Gateway API methods. Gateway Agent subclasses for specific development shells override the stubs with method definitions tailored to the corresponding knowledge model,and data and control interfaces. The top-level API consists of the following five methods:

- :extract-data;
- :inject-data;
- :initialize-shell;
- :process-request;
- :process-response.

The first three top-level API methods are defined for each Gateway Agent subclass in terms of intermediate-level DMS API functions, which differ in reflection of variations in shell architectures. However, each subclass API contains elements from the following categories:

- external interface functions;
- data interface functions (for data and knowledge access control);
- shell control functions.

The first API category encompasses shell-specific functions for passing data and commands from an application out to the Knowledge Gateway. Depending on the shell in question, the Gateway API may be more or less elaborate. For example, the Gateway for CLIPS V4.3 defines a single external API function to initiate interactions between rules in a CLIPS application and its embedding Agent, which hides MetaCourier and DMS API functions completely. The CLIPS Gateway API will be extended to reflect the procedural and object-oriented programming extensions in CLIPS Version 5.

Functions in the second category combine (a) the DMS API, which maps data and knowledge between SOCIAL's canonical DMS structures and the representational model native to a specific shell with (b) the shell-specific programmatic data interface used to generate, modify, and access data and knowledge structures in the native representational format. Examples include asserting and retracting structures, sending object-oriented messages, and modifying object attributes. For example, Make-CLIPS-fact-from-fact converts a DMS fact into a string, which is automatically inserted into the current CLIPS facts-list using the CLIPS assert function. The third

category encompasses shell-specific control interface capabilities such as start, clear, run, reset, exit, and saving and loading code and/or knowledge base files.

The top-level :extract-data and :inject-data Gateway methods consolidate the intermediate-level data interface API functions. Typically, :inject-data and :extract-data consist of program Case statements that invoke conversion functions for translating between different types defined in the native information model for a given shell and structures in the SOCIAL/DMS model. For example, :inject-data may call a DMS-level API function to map and insert a DMS frame structure as a fact-group into a CLIPS knowledge base and another to insert a DMS fact. Access direction (reading or writing) is implicitly reflected in the developer's choice of Extract (read) or Inject (write). Both methods are preprogrammed to dispatch automatically on data type, with options to override defaults (e.g., to map a DMS frame into a CLIPS fact-group instead of a deftemplate). Similarly, the :initialize-shell method represents the locus for control interface functions. Behavior is again classified by case and dependent on the target tool or program. For example, CLIPS employs different API functions to load textual and compiled knowledge bases.

The remaining pair of top-level Knowledge Gateway API methods, :process-request and :process-response, are application-specific. A shell-based application is integrated into a distributed system by specializing the Gateway Agent subclass for the relevant shell. Specialization here consists of overriding the stub versions of Process-request and :process-response inherited from KNOWL-GW and defining the required integration behaviors. Developers redefine these two methods by employing the generic API functions :extract-data, :inject-data, and :initialize-shell to pass information and control into and out of the target application via its associated shell.

The Gateway model is particularly powerful for integrating shell-based applications, in that the shell-specific methods (viz., :inject-data, :extract-data, :initialize-shell), are defined only once, namely in a KNOWL-GW subclass for the given shell. Application developers do not have to modify these API elements unless API extensions are necessary. Any application based on that shell can be embedded in a Gateway that is a subclass of the shell-specific Gateway Agent. The application Gateway inherits the generic tool-specific API interface, which means that the developer only has to program the methods :process-request (for server behaviors) and :process-response (for client behaviors). Individual interactions with the shell are specified using the inherited API to extract or inject particular data and to control the shell.

For custom applications, all five API methods are defined in one and the same Gateway Agent, namely the KNOWL-GW Agent subclass level. Therefore, inheritance does not play as powerful a role in assisting the application integrator as it does for multiple programs based on a common shell interface. Nevertheless, the generic :in-filter and :out-filter methods are inherited, providing the standardized message control model for peer-to-peer interactions. Moreover the Gateway model is useful as a methodological template in that it prescribes a uniform and intuitive partitioning of interface functionality: specific interactions between an application, its Gateway, and external systems are isolated in :process-request and :process-response, which invoke the utility API functions such as :inject-data as appropriate.

CLIPS Knowledge Gateway

CLIPS-GW is a subclass of the KNOWL-GW Agent class. As with all other Knowledge Gateway Agent subclasses, it inherits the KNOWL-GW message structure, :in-filter and :out-filter, and stub API methods. CLIPS-GW defines custom :inject-data, :extract-data, and :initialize-shell methods tailored to the CLIPS knowledge model, data and control interfaces. These custom methods are built up from a set of intermediate level API functions, which are summarized in Table 1. Specifically, :inject-data is based on Load-CLIPS-Data, which depends on CLIPS-Dispatch, and Load-CLIPS-Files. :extract-data relies on the function gw-return. :initialize-shell invokes the

basic shell control API functions, based on keyword symbols specified in incoming messages. Analogous APIs are defined for Knowledge Gateways for other AI shells, such as KEE.

Category	Function	Behavior
Shell Control	CLIPS-Start	starts CLIPS and sets a global flag
	CLIPS-Clear	clears all facts from CLIPS fact-list
	CLIPS-Init	if flag is set, calls CLIPS-Clear otherwise calls CLIPS-Start
	CLIPS-Run-Appl	runs CLIPS rule engine to completion accepts optional integer to limit # of rule firings
	CLIPS-Load-Appl	loads a specified rule base file into CLIPS
	CLIPS-Reset	asserts deffacts facts into CLIPS fact-list
	CLIPS-Assert	asserts a fact (string) into CLIPS fact-list\
	CLIPS-Retract	retracts fact (C pointer) from CLIPS fact-list
	CLIPS-Display-Facts	displays facts to output stream
Data Interface	CLIPS-Dispatch	calls a C dispatch routine to translate DMS structures and create CLIPS facts, fact-groups, deftemplates, or rules, as appropriate.
	Load-CLIPS-Data	reads DMS data from composite DMS structure and calls CLIPS-Dispatch on each one (except files)
	Load-CLIPS-Files	reads the composite DMS object for file pathname strings and calls CLIPS-Load-Appl
External Interface	gw-return	an external/user function defined to CLIPS for passing data from rules back to Agent

Table 1. Intermediate Level API for the CLIPS Gateway Agent (Version 4.3)

The CLIPS Gateway API defines an external user function that provides a high-level interface between a CLIPS application and its embedding Gateway . This function, called gw-return, enables CLIPS applications to pass data and/or control information to their Gateway Agents by stuffing a stream buffer that is unpacked using the top-level :extract-data command. gw-return function calls appear as consequent clauses, as illustrated in the example rule shown below. gw-return takes two arguments - a DMS structure type such as a Fact and a string or pointer. The first item is used to parse the datum and convert it into the specified type of DMS structure. Multiple gw-return clauses can be placed into the right-hand side of a single rule, Also, multiple rules can contain gw-return clauses.

```
(defrule TALK-BACK-TO-GATEWAY
   "rule that passes desired result, a fact, that has been asserted
   into appl KB back through the Gateway to requesting Agent"
      ?requestor <- (requestor ?appl-agent)
      ?answer  <- (answer $?result)
 =>
      (printout t "Notifying " ?requestor "of result" $?result" crlf)
      (gw-return FACT (str-implode $?result))
```

Messages to Knowledge Gateway Agents contain five elements, the target Agent, Environment, Host, data, and command options. In server mode (responding to messages from other application Agents), the CLIPS-GW :in-filter executes :initialize-shell for the specified command options to prepare CLIPS, invokes :process-request for the incoming data, and sets the results. Typically :process-request injects data, which includes loading rule bases, runs the rule engine, and extracts results. The :out-filter translates the :in-filter results into SOCIAL neutral exchange format, which constitute the reply that MetaCourier returns to the requesting Agent.

In client mode, a CLIPS application initiates a message to some external application Agent via its embedding CLIPS Agent. Here, the :in-filter invokes :extract-data and passes the message contents and any specified command options to the target application Agent. The :out-filter then invokes :process-response to deal with the reply. Typically, :process-response invokes :inject-data to introduce response data into the CLIPS fact-list and restarts the CLIPS rule engine to resume reasoning.

EXAMPLE APPLICATIONS OF CLIPS GATEWAY AGENTS

This section of the paper describes two demonstration systems that employ CLIPS Gateways to integrate expert systems for operations support for the Space Shuttle fleet. Processing, testing, and launching of Shuttle vehicles takes place at facilities dispersed across the Kennedy Space Center complex. The Launch Processing System (LPS) provides the sole direct, real-time interface between Shuttle engineers, Orbiter vehicles and payloads, and associated Ground Support Equipment to support these activities (Heard 1987). The locus of control for the LPS is the Firing Room, an integrated network of computers, software, displays, controls, switches, data links and hardware interface devices. Firing Room computers are configured to perform independent LPS functions through application software loads. Shuttle engineers use Console computers to monitor and control specific vehicle and Ground Support systems. These computers are connected to data buses and telemetry channels that interface with Shuttles and Ground Support Equipment. The Master Console is a computer that is dedicated to operations support of the Firing Room itself.

Integrating Configuration and Fault Management

The first application illustrates the use of a CLIPS Gateway in a server role to integrate expert systems that automate configuration and fault management operations support tasks (Adler 1990). X-Switcher is a prototype expert system that supports operators of the Switching Assembly used to manage computer configurations in Firing Rooms. X-Switcher was implemented using CLIPS V4.3 on a Sun workstation. OPERA (for Operations Analyst) is an integrated collection of expert systems that automates critical operations support functions for the Firing Room (Adler 1989). In essence, OPERA retrofits the Master Console with automated, intelligent capabilities for detecting, isolating and managing faults in the Firing Room. The system is implemented in KEE and runs on a Texas Instruments Explorer Lisp Machine. PRACA, NASA's Problem Reporting and Corrective Action database, was simulated using the Oracle relational DBMS, again on a Sun workstation.

A distributed system prototype was constructed with SOCIAL, using appropriate library Agents to integrate these three applications - a CLIPS Gateway for X-Switcher, a KEE Gateway for OPERA, and an Oracle Gateway for PRACA. The prototype executes the following scenario. First, OPERA receives LPS error messages that indicate a failure in a Firing Room computer subsystem. OPERA then requests a reconfiguration action from X-Switcher via the OPERA KEE Gateway Agent. This request is conveyed via a MetaCourier message to the CLIPS Gateway Agent. The message contains a DMS fact-group that specifies the observed computer problem, the pathname for the X-Switcher rule base on the Sun platform, and the current Firing Room Configuration Table. OPERA models the Configuration Table as a unit, which is KEE's hybrid frame-object knowledge structure. The OPERA Agent automatically unpacks slot data from the Table unit and appends it to the DMS fact-group via KEE Gateway API calls.

Upon receiving the KEE Gateway's message, the CLIPS Gateway Agent executes the following sequence of tasks. First, CLIPS is loaded, if necessary, and initialized. Second, the X-Switcher expert system rule base is loaded. Third, the DMS OPERA data object from the KEE Gateway message is translated and asserted as a CLIPS fact-group. Fourth, CLIPS is reset and the rule engine is run. X-Switcher rules derive a set of candidate replacement CPUs for the failed Firing

Room computer and prompt the user to select a CPU. It then displays specific instructions for reconfiguring the Switching Assembly to connect the designated CPU and prompts the user to verify successful completion of the switching activity. Finally, X-Switcher interacts with its CLIPS Gateway to reply to OPERA that reconfiguration of the specified CPU succeeded or failed. This process is triggered when CLIPS executes an X-Switcher rule containing a consequent clause of the form *(gw-return result)*. The CLIPS Gateway converts *result* into a DMS fact, which is transmitted to the OPERA KEE Gateway. This Agent asserts this fact as an update value in a subsystem status slot in the Configuration Table KEE unit. Finally, the OPERA Gateway formulates an error report, which is dispatched in a message to the Oracle Gateway Agent, which updates the simulated PRACA Problem-Tracking Database.

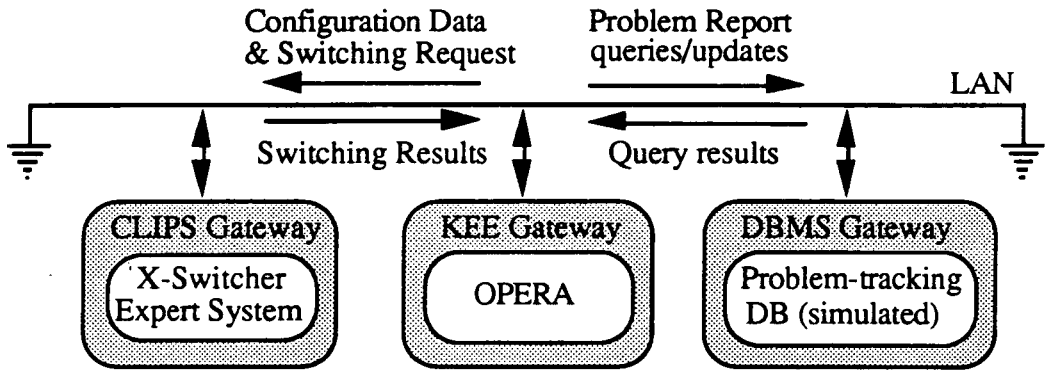

Figure 5. CLIPS Application configured as a Server

Coordinating Independent Systems to Enhance Fault Diagnosis Capabilities

The second distributed application illustrates the use of a SOCIAL CLIPS Gateway Agent in a client role (Adler 1991). GPC-X is a prototype expert system for isolating faults in the Shuttle vehicle's on-board computer systems, or GPCs. GPC-X was implemented using CLIPS V4.3 on a Sun workstation. One type of memory hardware fault in GPC computers manifests itself during switchovers of Launch Data Buses. These buses connect GPCs to Firing Room Console computers until just prior to launch, when communications are transferred to telemetry links. Unfortunately, the data stream that supplies the GPC-X expert system does not provide any visibility into the occurrence of Launch Data Bus switchovers (or the health of the GPC Console Firing Room computer). Thus, GPC-X can propose but not test certain fault hypotheses about GPC problems, which seriously restricts the expert system's overall diagnostic capabilities.

However, Launch Data Bus switchover events are monitored automatically by the LPS Operating System, which triggers warning messages that are detected and processed by the OPERA system discussed above. CLIPS and KEE Gateway Agents were used to integrate GPC-X and OPERA, as before. A SOCIAL Manager Agent was used to mediates interactions between these application Gateway Agents to coordinate their independent fault isolation and test activities.

Specifically, GPC-X, at the appropriate point in its rule-based fault isolation activities, issues a request via its embedding Agent to check for Launch Data Bus switchovers to the Manager. The request is initiated by a gw-return consequent clause in the CLIPS rule that proposes the memory fault hypothesis. When this rule fires, CLIPS executes the gw-return function, which sends a message to the GPC-X CLIPS Gateway Agent. This Agent formulates a message to the Manager which contains a Manager API task request for the LDB-Switchover-Check service.

The Manager searches its directory for an appropriate server Agent for LDB-Switchover-Check, reformulates the task data into a suitable DMS-based message, and passes it to the OPERA KEE Gateway Agent. The :process-request method for this application Agent performs a search of the knowledge base used by OPERA to store interpreted LPS Operating System error messages. The objective is to locate error messages, represented as KEE units, indicative of LDB switchover events. The OPERA Gateway :out-filter uses the Manager API to translate search results into a suitable DMS structure, which is posted back to the Manager. In this situation, the OPERA Gateway Agent contains *all* of the request processing logic: OPERA itself is a passive participant that continues its monitoring and fault isolation activities without significant interruption.

Next, the Manager returns the results of the LDB-Switchover-Check request back to GPC-X's CLIPS Gateway Agent. The Agent :in-filter executes the :process-response method, which transparently converts the Manager DMS object into a CLIPS fact that is asserted into the GPC-X fact base. Finally, the GPC-X Agent re-activates the CLIPS rule engine to complete GPC fault diagnosis. Obviously, new rules had to be added to GPC-X to exploit the newly available hypothesis test data. However, all of the basic integration and coordination logic is supplied by the embedding GPC-X Gateway Agent or the HDC-Manager.

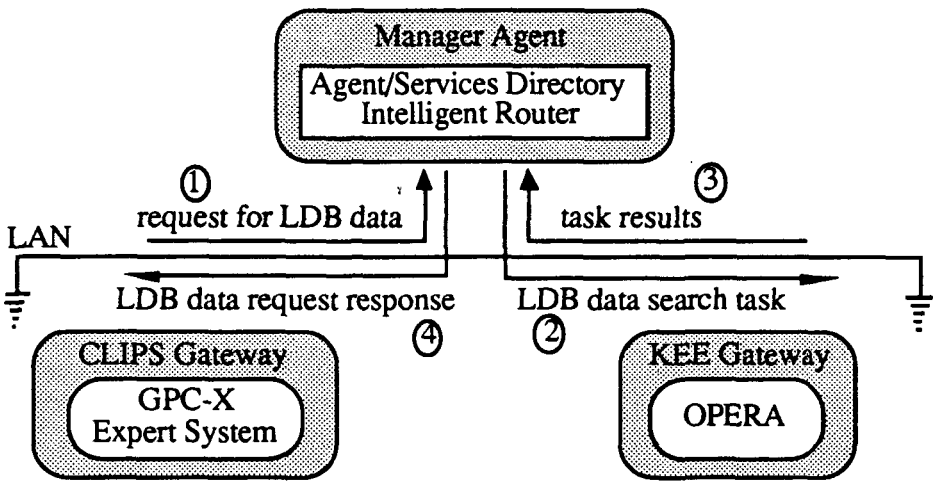

Figure 6. CLIPS Gateway configured as a client

This SOCIAL prototype demonstrates non-intrusive *system-level* coordination of distributed applications that solve problems at the subsystem level. Neither application is capable of full diagnosis individually. GPC-X can generate GPC fault candidates, but lacks data concerning other LPS subsystems that is necessary for testing these hypotheses. OPERA automatically detects LPS error messages that are relevant to GPC-X's candidate test requirements. However, it lacks the contextual knowledge about GPC computers, and awareness of GPC-X's capabilities and current activities, to recognize the potential significance of specific LPS data as a test for GPC-X fault hypotheses. Gateway Agents integrate the two systems, supplying communication and data mapping capabilities. The Manager establishes the logical connections required to combine and utilize the fragmented subsystem-specific knowledge of the two applications to enhance diagnostic capabilities. This coordination architecture is non-intrusive in that neither system was modified to include direct knowledge of the other, its interfaces, knowledge model, or platform. The Manager directory and routing capabilities introduce an isolating layer of abstraction, enhancing the "plug-compatible character of the integration architecture.

RELATED WORK

The most closely related research to SOCIAL CLIPS Gateway Agents is the AI Bus (Schultz 1990), a framework for integrating rule-based CLIPS applications in a distributed environment. SOCIAL and AI Bus both rely on modular message-based communications. AI Bus uses a client-server model based on remote procedural calls that is currently restricted to Unix hosts. SOCIAL's MetaCourier layer supports a fully peer-to-peer model that is transparent across diverse platforms. SOCIAL and AI Bus integrate applications using Agents, whose API functionality are roughly comparable. Each Agent has a dedicated message control module and can communicate directly with one another. Indirect interactions are mediated by a dedicated organizational Agent, the SOCIAL Manager or the AI Bus Blackboard. It appears that AI Bus Agents are currently restricted primarily to CLIPS-based knowledge sources, while SOCIAL Gateways provide broader support for KEE, CLIPS, and other tool-based and custom applications.

Other tools for developing heterogeneous distributed intelligent systems include GBB (Corkill 1986), ERASMUS (Jagannathan 1988), MACE (Gasser 1987), and ABE (Hayes-Roth 1988). These systems lack SOCIAL's modular, layered, architecture, and are considerably less extensible below the top-level developer interfaces. GBB and ERASMUS impose a blackboard control architecture for integrating distributed applications. ABE allows multiple kinds of interaction models (e.g., transaction, data flow, blackboard), but it is not clear how easily these can be combined within a single system. MACE provides few organizational building blocks for developing complex architecture beyond a relatively simple routing Agent. None of these frameworks provide a predefined integration interface to CLIPS, although ABE and GBB include simple "black box" tools such as external or foreign function call passing to build one.

STATUS AND FUTURE DEVELOPMENT

The original CLIPS Gateway Agent was implemented for CLIPS V4.3 in Franz Common Lisp, using a foreign function interface to the CLIPS API, which is written in C. Within the next year, we intend to develop a full C implementation of the Agent. This Agent will also be extended to reflect enhancements in CLIPS V5.0, most notably, procedural programming and the CLIPS Object-Oriented Language.

CONCLUSIONS

CLIPS was designed to facilitate embedding intelligent applications within more complex systems. However, lacking built-in support for distributed communications capability, applications implemented with CLIPS are generally "hardwired" directly to other software systems residing either on the same platform or on a parallel multi-processor. Moreover, CLIPS integration interfaces are typically custom-built, by systems level programmers who are experienced with the mechanics of interprocess communication. SOCIAL CLIPS Gateway Agents provide a generalized, high-level approach to integrating CLIPS applications with other intelligent and conventional programs across heterogeneous hardware and software platforms. Gateways exploit object-oriented inheritance to partition generic distributed communication and control capabilities, shell-specific APIs, and application-specific functionality. Developers need only learn the high-level APIs to integrate CLIPS applications with other application Agents. SOCIAL's modular and extensible integration technologies promote a uniform, "plug compatible" model for non-intrusive, peer-to-peer interactions among heterogeneous distributed systems.

ACKNOWLEDGMENTS

Development of SOCIAL, including the CLIPS Gateway Agent, was sponsored by the NASA Kennedy Space Center under contract NAS10-11606. MetaCourier was developed by Robert Paslay, Bruce Nilo, and Robert Silva, with funding support from the U.S. Army Signals Warfare

center under Contract DAAB10-87-C-0053. Rick Wood designed and implemented the C portions of the CLIPS Gateway API. Bruce Cottman developed the prototypes for SOCIAL's data management tools and the Oracle Gateway Agent.

REFERENCES

Adler, R.M. (1991). A Hierarchical Distributed Control Model for Coordinating Intelligent Systems. *Proceedings, 1991 Goddard Conference on Space Applications of Artificial Intelligence.* NASA CP-3110. pp. 183-198.

Adler, R.M. and Cottman, B.H. (1990). EXODUS: Integrating Intelligent Systems for Launch Operations Support. *Proceedings, Fourth Annual Workshop on Space Operations, Applications, and Research Symposium (SOAR'90).* NASA CP-3103. pp. 324-330.

Adler, R.M., Heard, A., and Hosken, R.B. (1989). OPERA - An Expert Operations Analyst for A Distributed Computer Network. *Proceedings, Annual AI Systems in Government Conference.* IEEE Computer Society Press. Washington, DC. pp. 179-185.

CLIPS Reference Manual. (1989).Version 4.3. Artificial Intelligent Section, Johnson Space Center, Houston, TX.

CLIPS Reference Manual. (1991). Version 5.0. Software Technology Branch, Johnson Space Center, Houston, TX.

Corkill, D.D., Gallagher, K.Q., and Murray, K. (1986). GBB: A Generic Blackboard Development System. *Proceedings Fifth National Conference on Artificial Intelligence.* pp. 1008-1014.

Gasser, L., Braganza, C., and Herman, N. (1987). MACE: A Flexible Testbed for Distributed AI Research. in *Distributed Artificial Intelligence Vol. 1.* M. Huhns (Ed.) Morgan Kaufmann. Los Altos, California, 1987.

Hayes-Roth, F., Erman, L.D., Fouse, S., Lark, J.S., and Davidson, J. (1988). ABE: A Cooperative Operating System and Development Environment. in A. H. Bond and L. Gasser, (Eds.) *Readings in Distributed Artificial Intelligence.* Morgan-Kaufmann. Los Altos, CA.

Heard, A.E. (1987). The Launch Processing System with a Future Look to OPERA. *Acta Astronautica,* IAF-87-215.

Jagannathan, V., Dodhiawala, R., and Baum, L. (1988). The Boeing Blackboard System: The Erasmus Version. *International Journal of Intelligent Systems.* vol. 3. no. 3. pp. 281-294.

Schultz, R.D., and Stobie, I.C. (1990). Building Distributed Rule-Based systems Using the AI Bus. *First CLIPS Conference Proceedings.* NASA CP-10049. pp.676-685.

Symbiotics, Inc. (1990). Object-Oriented Heterogeneous Distributed Computing with MetaCourier. Technical Report. Symbiotics, Inc. Cambridge, MA.