# CLIPS, AppleEvents, and AppleScript: Integrating CLIPS with Commercial Software



Michael M. Compton
*compton@ptolemy.arc.nasa.gov*
(415) 604-6776

Shawn R. Wolfe
*shawn@ptolemy.arc.nasa.gov*
(415) 604-4760



AI Research Branch / Recom Technologies, Inc.
NASA Ames Research Center
Moffett Field, CA 94035-1000

*Abstract:*

*Many of today's intelligent systems are comprised of several software modules, perhaps written in different tools and languages, that together help solve the users' problem. These systems often employ a knowledge-based component that is not accessed directly by the user, but instead operates "in the background" offering assistance to the user as necessary. In these types of modular systems, an efficient, flexible, and easy-to-use mechanism for sharing data between programs is crucial. To help permit transparent integration of CLIPS with other Macintosh applications, the AI Research Branch at NASA Ames Research Center has extended CLIPS to allow it to communicate transparently with other applications through two popular data-sharing mechanisms provided by the Macintosh operating system; Apple Events (a "high-level" event mechanism for program-to-program communication), and AppleScript, a recently-released scripting language for the Macintosh. This capability permits other applications (running on either the same or a remote machine) to send a command to CLIPS, which then responds as if the command were typed into the CLIPS dialog window. Any result returned by the command is then automatically returned to the program that sent it. Likewise, CLIPS can send several types of Apple Events directly to other local or remote applications. This CLIPS system has been successfully integrated with a variety of commercial applications, including data collection programs, electronic forms packages, DBMSs, and email programs. These mechanisms can permit transparent user access to the knowledge base from within a commercial application, and allow a single copy of the knowledge base to service multiple users in a networked environment.*

## Introduction

Over the past few years there has been a noticeable change in the way that "intelligent applications" have been developed and fielded. In the early and mid-1980s, these systems were typically large, monolithic systems, implemented in LISP or PROLOG, in which the inference engine and knowledge base were at the core of the system. In these "KB-centric" systems, other parts of the system (such as the user interface, file system routines, etc.) were added on once the inferential capabilities of the system were developed. These systems were often deployed as multi-megabyte "SYSOUTs" on special-purpose hardware that was difficult if not impossible to integrate into existing user environments.

Today, intelligent systems are more commonly being deployed as a collection of interacting software modules. In these systems, the knowledge base and inference engine serve as a subordinate and often "faceless" component that interacts with the user indirectly, if at all.

Several such systems have been deployed recently by the Artificial Intelligence Research Branch at NASA's Ames Research Center:

The Superfluid Helium On-Orbit Transfer system (or SHOOT) was a modular expert system that monitored and helped control a Space-Shuttle-based helium transfer experiment which flew

onboard the Endeavor orbiter during mission STS-57 in June, 1993. In that system, a CLIPS-based knowledge system running on a PC laptop on the Shuttle's Aft Flight Deck was embedded in a custom-developed C program that helped crew members (and ground controllers) perform transfer of cryogenic liquids in microgravity.

The Astronaut Science Advisor (also known as "PI-in-a-Box", or simply "[PI]") was another modular system that flew onboard the Space Shuttle in 1993 ([3]). [PI] ran on a Macintosh PowerBook 170 laptop computer and helped crew members perform an experiment in vestibular physiology in Spacelab. This system was comprised of a CLIPS-based knowledge system that interacted with HyperCard (a commercial user interface tool from Claris Corporation and Apple Computer) and LabVIEW (a commercial data acquisition tool from National Instruments) to monitor data being collected by the crew and offer advice on how to refine the experiment protocol based on collected data and, when necessary, troubleshoot the experiment apparatus. In that system, CLIPS ran as a separate application and communicated with HyperCard via specially-written external commands (XCMDs) that were linked into the HyperCard application.

The Prototype Electronic Purchase Request (PEPR) system ([1] and [2]) is yet another modular system that motivated the enhancements being described in this paper. The PEPR system uses a CLIPS-based knowledge system to validate and generate electronic routing slips for a variety of electronic forms used frequently at NASA Ames. A very important part of the PEPR system's capabilities is that it is able to work "seamlessly" with several commercial applications.

Interestingly, these applications represent a progression of integration techniques in which the knowledge-based component interacts with custom-developed programs. This progression starts with tightly-coupled, "linked" integration of components (in SHOOT), to integration via custom-built extensions to a specific commercial product (in [PI]), and finally to a general-purpose integration mechanism that can be used with a variety of commercial products (in PEPR).

## Requirements for the PEPR system

A brief description of the PEPR system will help explain how we determined our approach to the problem of integrating CLIPS with other commercial applications.

Our primary goal in the development of the PEPR system was to demonstrate that knowledge-based techniques can improve the usability of automated workflow systems by helping validate and route electronic forms. In order to do this, we needed to develop a prototype system that was both usable and provided value in the context of a real-world problem. This meant that in addition to developing the knowledge base and tuning the inference engine, we also needed to provide the users with other tools. These included software to fill out electronic forms and send them between users, as well as mechanisms that could assure recipients of these forms that the senders were really who they appeared to be, and that the data hadn't been altered, either accidentally or maliciously, in transit. We also wanted to provide users with a way by which to find out the status of previously-sent forms, in terms of who had seen them and who hadn't.

Of course, we didn't want to have to develop all of these other tools ourselves. Thankfully, commercial solutions were emerging to help meet these additional needs, and we tried to use these commercial tools wherever possible. Our challenge became, then, how to best integrate the CLIPS-based knowledge system component with these other commercial tools.

## Our first try at component integration: Keyboard macros

Our initial integration effort was aimed at developing a "seamless" interface between the knowledge system and the electronic forms package (which of course was the primary user interface). Our

first try was to use a popular keyboard macro package to simulate the keystrokes and mouse clicks that were necessary to transfer the data from the forms program to CLIPS. The keyboard macro would, with a single keystroke, export the data on the form to a disk file, switch to the CLIPS application, and then "type" the `(reset)` and `(run)` commands (and the requisite carriage returns) into the CLIPS dialog window. This would then cause CLIPS to read in and parse the data from the exported file.

This worked fairly well, and was reasonably easy to implement (that is, it did not require modifying any software). However, the solution had several serious drawbacks. First, it required that the user have the keyboard macro software (a commercial product) running on his or her machine. Also, the macro simply manipulated the programs' user interface, and was therefore very distracting for the user because of the flurry of pull down menus and dialog boxes that would automatically appear and disappear when the macro was run. Most importantly, however, was that the keyboard macro approach required each user to have their own copy of CLIPS and the knowledge base running locally on their machine. This, of course, would have presented numerous configuration and maintenance problems; the "loadup" files would have to be edited to run properly on each user's machine, and fixing bugs would have involved distributing new versions of the software to all users (and making sure they used it). Another drawback was that we found that the keyboard macros would often "break" midway through their execution and not complete the operation. This would, of course, have also been very frustrating for users. As a result, this early system, although somewhat useful for demonstrations, was never put into production.

## Our second try: Apple Events

Our next attempt at integrating the electronic forms package with CLIPS was motivated by a demonstration we saw that showed that it was possible to integrate the forms package we had selected with a commercial DBMS. In this demo, a user was able to bring up an electronic order form, and enter information such as, say, a vendor number. When the user tabbed out of the Vendor Number field, the forms software automatically looked up the corresponding information in a "vendor table" in a commercial DB running on a remote machine. The vendor's name and address automatically appeared in other fields on the form.

Of course, this sort of data sharing is quite common in many data processing environments. However, we found this capability exciting because it involved two Macintosh applications (including our forms package of choice), from different vendors, cooperating across an AppleTalk network, with virtually no assistance from the user. This was exactly the sort of "seamless" behavior we were seeking.

This functionality was provided by what are known as "Apple Events", a "high-level event mechanism" that is built into "System 7", the current major release of the Macintosh operating system. We were encouraged by the fact that the forms package we had selected supported the use of Apple Events to communicate with external programs. We thought it might be possible to implement our desired inter-program interface by making CLIPS mimic the DBMS application with respect to the sending and receiving of Apple Events. However, the particular set of Apple Events supported by that version of the forms software was limited to interaction that particular DBMS product. This was problematic for several reasons. First, the set of Apple Events being used were tailored to interaction with this particular data base product, and it wasn't clear how to map the various DB constructs they supported into CLIPS. Second, it would have required a considerable amount of programming effort to develop what would have been the interface between a specific forms product and a specific data base product. We wanted to minimize our dependence on a particular product, in case something better came along in the way of a forms package or a DBMS.

So, we ended up not implementing the DB-specific Apple Event mechanism. However, we were still convinced that the Apple Event mechanism was one of the keys to our integration goals, and *did* implement several more general Apple Event handlers, which are described below.

## What are Apple Events?

The following provides a brief description of the Apple Event mechanism. As mentioned above, Apple Events are a means of sharing data between to applications, the support for which is included in System 7. An Apple Event has four main parts. The first two components are a 4-character alphanumeric "event class" and a 4-character "event ID". The third component identifies the "target application" to which the event is to be sent. The actual data to be sent is a collection of keyword/data element pairs, and can range from a simple string to a complex, nested data structure. (In addition to these, there are several other parts of the event record which specify whether a reply is expected, a reply time-out value, and whether the target application can interact with a user.) From a programmer's perspective, these events are created, sent, received, and processed using a collection of so-called "ToolBox" calls that are available through system subroutines.

Although Apple Events are a "general purpose" mechanism for sharing data between programs, programs that exchange data must follow a pre-defined protocol that describes the format of the events. These protocols and record formats are described in the *Apple Event Registry*, published periodically by Apple Computer.

```
Event Class: MISC
   Event ID: DOSC
Target App: <ProcessID>
Event Data: "----"
            "(load \"MYKB.CLP\")"
```

**Figure 1: An Example Apple Event**

Figure 1 shows an example of a "do script" event that might be sent to CLIPS. The event class is "MISC", and the event ID is "dosc". The 4-dash "parameter keyword" identifies the string as a "direct object" parameter, and is used by the receiving program to extract the data in the event record. The protocol associated with the "do script" event in the Apple Event Registry calls for a single string that contains the "script" to be executed by the target application. In this example, it's a CLIPS command for loading a knowledge base file. This example shows the simplest data format. More complex Apple Events may require numerous (and even embedded) keyword/data pairs.

The Apple Event mechanism is very powerful. However, we wanted a more flexible solution to our needs of integrating CLIPS with the commercial forms application. It would have required considerable effort to implement a Apple-Event based programmatic interface between CLIPS and the other applications with which we needed to share data. That is, we didn't want to have to code

specific Apple Event handling routines for every other application we were using. This would have made it very difficult to "swap in" different applications should we have needed to. What we needed was a way by which we could use the power of Apple Events to communicate between CLIPS and other programs, but make it easier to code the actual events for a given program.

## Our third try: AppleScript

Just as we were trying to figure out the best way to link CLIPS with the other programs we wanted to use, we learned about a new scripting technology for the Mac called AppleScript. This new scripting language provides most of the benefits of Apple Events (and is in fact based on the Apple Event mechanism) such as control of and interaction with Macintosh applications running either locally or remotely. In addition, it offers some other benefits that pure Apple Event programming does not. For example, using AppleScript, one is able to control and share data with an ever-growing array of commercial applications, without having to understand the details of the application's Apple Event protocol. AppleScript comes with a reasonably simple Script Editor that can be used to compose, check, run, and save scripts. In addition to providing all the constructs that one might expect in any programming language (variables, control, I/O primitives) it is also extendible and can even be embedded in many applications.

The thing that made AppleScript particularly appealing for our use was that it utilized the Apple Event handlers that we had already added to CLIPS. All that was necessary to permit the "scriptability" we desired was the addition of a new "Apple Event Terminology Extension" resource to our already-modified CLIPS application. This AETE resource simply provided the AppleScript Editor (and other applications) with a "dictionary" of commands that CLIPS could understand, and the underlying Apple Events that the AppleScript should generate.

Another very appealing aspect of integrating programs with AppleScript is more and more commercial software products are supporting AppleScript by becoming scriptable. That of course makes it much easier to take advantage of new software products as they come along. For example, we recently upgraded the forms tracking data base used by the PEPR system. We were able to replace a "flat-file" data base package with a more-powerful relational DBMS product with only minor modifications to the AppleScript code linking the DB to the other applications. This would have been far more difficult (if even possible) had we relied solely on integration with AppleEvents.

The main disadvantage to using AppleScript rather than Apple Events is that AppleScript is somewhat slower than sending Apple Events directly. However, the increased flexibility and power of AppleScript more than compensates for the comparative lack of speed.

```
tell application "CLIPS" of machine "My Mac"
      of zone "Engineering"
   do script "(reset)"
   do script "(run)"
   set myResult to evaluate "(send [wing-1] get-weight)"
   display dialog "Wing weight is " & myResult
end tell
```

**Figure 2: An example AppleScript program**

Figure 2 shows an example script that could be used to control CLIPS remotely. There are several things to note in this example. First, the commands are passed to CLIPS and executed as though they had been entered into the Dialog Window. The example shows both the "do script" command (which does not return a result) and the "evaluate" command (which does). The example also shows a "display dialog" command which is built in to AppleScript and displays the result in a modal dialog box. Of particular interest is that the CLIPS application is running on another Macintosh, which is even in another AppleTalk zone.

## Specific CLIPS Extensions

The following paragraphs describe the actual CLIPS extensions that have been implemented to support the functionality described above. Note that some of these extensions were actually implemented by Robert Dominy, formerly of NASA Goddard Space Flight Center.

## Receiving Apple Events

It's now possible to send two types of Apple Events to CLIPS. Each takes a string that is interpreted by CLIPS as though it were a command typed into the Dialog Window. The format of these Apple Events is dictated by the *Apple Event Registry*, and they are also supported by a variety of other applications. Note that CLIPS doesn't currently return any syntax or execution errors to the program that sent the Apple Events, so it is the sender's responsibility to ensure that the commands sent to CLIPS are syntactically correct.

## The "do script" Event

The "do script" event (event class = MISC, event ID=DOSC) passes its data as a string which CLIPS interprets as if it were a command that were typed into the Dialog Window. It returns no value to the sending program.

## The "evaluate" Event

The "evaluate" event (event class = MISC, event ID=EVAL) is very similar to the do script event, and also passes its data as a string which CLIPS interprets as if it were a command that were typed into the Dialog Window. However, it does return a value to the sending program. This value is always a string, and can be up to 1024 bytes in length.

## Sending Apple Events from CLIPS

The two Apple Events described above can also be sent by CLIPS from within a knowledge base. Of course, the application to which the events are sent must support the events or an error will occur. However, as mentioned above, the "do script" and "evaluate" events are very common and supported by many Mac applications.

## SendAEScript command

The SendAEScript command sends a "do script" event and can appear in a CLIPS function or in the right-hand-side of a rule. The syntax of the SendAEScript command is as follows:

```
(SendAEScript <target app> <command>)
```

In the above prototype, `<target app>` is an "application specification" and `<command>` is a valid command understandable by the target application. An application specification can have one of three forms; a simple application name, a combination application name, machine name and

AppleTalk Zone name, or a process identifier (as returned by `PPCBrowser`, described below). The `SendAEScript` command returns zero if the command is successfully sent to the remote application, and a variety of error codes if it was not. Note that a return code of zero does *not* guarantee that the command was successfully executed by the remote application; only that it was sent successfully .

The following examples show each of the application specification types.

```
CLIPS>(SendAEScript "HyperCard" "put \"hello\" into msg")
0
CLIPS>
```

The above example sends a "do script" Apple Event to HyperCard running on the local machine, and causes it to put "hello" into the HyperCard message box.

```
CLIPS>(SendAEScript "HyperCard" "John's Mac" "R&D" "put \"hello\" into msg")
0
CLIPS>
```

The above example sends a similar "do script" Apple Event to HyperCard running on a computer called "John's Mac" in an AppleTalk zone named "R&D". Note that it is necessary to "escape" the quote characters surrounding the string "hello" to avoid them being interpreted by the CLIPS reader.

## SendAEEval command

The SendAEEval command is very similar to the SendAEScript command, differing only in that it returns the value that results from the target application evaluating the command.

```
(SendAEEval <target app> <command>)
```

The following examples show CLIPS sending a simple command to HyperCard running on the local machine:

```
CLIPS> (SendAEEval  "HyperCard"  "word 2 of \"my dog has fleas\" ")
"dog"
CLIPS>
```

Note that the result returned by SendAEEval is always a string, e.g.:

```
CLIPS> (SendAEEval  "HyperCard"  "3 + 6")
"9"
CLIPS>
```

The `SendAEEval` command does not currently support commands that require the target application to interact with its user. For example, one could not use `SendAEEval` to send an "ask" command to HyperCard.

## PPCBrowser command

The `PPCBrowser` function permits the CLIPS user to select an AppleEvent-aware program that is currently running locally or on a remote Mac. This command brings up a dialog box from which the user can click on various AppleTalk zones, machine names and "high-level event aware" applications. It returns a pointer to a "process ID" which can be bound to a CLIPS variable and used in the previously-described "send" commands.

```
CLIPS>(defglobal ?*myapp* = (PPCBrowser))
CLIPS> ?*myapp*
<Pointer: 00FF85E8>
```

The above example doesn't show the user's interaction with the dialog box.

## GetAEAddress command

The `GetAEAddress` function is similar to `PPCBrowser` in that it returns a pointer to a high-level aware application that can then be bound to a variable that's used to specify the target of one of the "SendAE" commands described earlier. Rather than presenting a dialog box to the user, however, it instead takes a "target app" parameter similar to that described above.

```
(GetAEAddress <target app>)
```

The following example shows the GetAEAddress function being used to specify the target of a `SendAEEval` function call.

```
CLIPS> (defglobal ?*myapp* = (GetAEAddress "HyperCard" "Jack's Mac" "R&D"))
CLIPS> (SendAEEval ?*myapp*  "8 + 9")
"17"
CLIPS>
```

## TimeStamp command

Another extension we've made is unrelated to inter-program communication. We have added a `TimeStamp` command to CLIPS. It returns the current system date and time as a string:

```
CLIPS>(TimeStamp)
"Wed Sep 7 12:34:56 1994"
CLIPS>
```

## Possible Future Extensions

In addition to the CLIPS extensions described above, we are also looking into the possibility of implementing several other enhancements. First, we want to generalize the current Apple Event sending mechanisms to permit the CLIPS programmer to specify the event class and event ID of the events to be sent. This is a relatively straightforward extension if we limit the event data to a string passed as the "direct object" parameter. It would be somewhat harder to allow the CLIPS programmer to specify more complex data structures, because we would have to design and implement a mechanism that allows the CLIPS programmer to construct these more complex combinations of keywords, parameters, and attributes. We will probably implement these extensions in stages.

Another extension we're considering is to make CLIPS "attachable". This would permit the CLIPS programmer to include pieces of AppleScript code in the knowledge base itself. This would significantly enhance the power of CLIPS, as it would make it possible to compose, compile, and execute AppleScript programs from within the CLIPS environment, and save these programs as part of a CLIPS knowledge base.

## Acknowledgments

Some the extensions described in this paper were designed and implemented by Robert Dominy, formerly of NASA's Goddard Space Flight Facility in Greenbelt, Maryland.

Also, Jeff Shapiro, formerly of Ames, ported many of the enhancements described herein to CLIPS version 6.0.

## References

[1] Compton, M., Wolfe, S. 1993 *Intelligent Validation and Routing of Electronic Forms in a Distributed Workflow Environment.*. Proceedings of the Tenth IEEE Conference on AI and Applications.

[2] Compton, M., Wolfe, S. 1994 *AI and Workflow Automation: The Prototype Electronic Purchase Request System.* Proceedings of the Third Conference on CLIPS.

[3] Frainier, R., Groleau, N., Hazelton, L., Colombano, S., Compton, M., Statler, I., Szolovits, P., and Young, L., 1994 *PI-in-a-Box, A knowledge-based system for space science experimentation*, AI Magazine, Volume 15, No. 1, Spring 1994, pp. 39-51.