HyperCLIPS: A HyperCard Interface to CLIPS

Brad Pickering
Randall W. Hill, Jr

Jet Propulsion Laboratory
4800 Oak Grove Drive MS 125-123
Pasadena, CA. 91109

Introduction:

HyperCLIPS combines the intuitive, interactive user interface of the Apple Macintosh®* with the powerful symbolic computation of an expert system interpreter. HyperCard® is an excellent environment for quickly developing the front end of an application with buttons, dialogs, and pictures, while the CLIPS interpreter provides a powerful inference engine for complex problem solving and analysis. By integrating HyperCard and CLIPS the advantages and uses of both packages are made available for a wide range of uses: rapid prototyping of knowledge-based expert systems, interactive simulations of physical systems, and intelligent control of hypertext processes, to name a few.

Interfacing HyperCard and CLIPS is natural. HyperCard was designed to be extended through the use of external commands (XCMDs), and CLIPS was designed to be embedded through the use of the I/O router facilities and callable interface routines. With the exception of some technical difficulties which will be discussed later, HyperCLIPS implements this interface in a straight forward manner, using the facilities provided. An XCMD called "ClipsX" was added to HyperCard to give access to the CLIPS routines: clear, load, reset, and run. And an I/O router was added to CLIPS to handle the communication of data between CLIPS and HyperCard.

Programming in HyperCLIPS:

Programming HyperCLIPS is only slightly more difficult than programming HyperCard and CLIPS separately. The three extra issues that one needs to understand are: how to use the "ClipsX" XCMD; how to use the I/O commands from CLIPS to get information to and from HyperCard; and when and how to pass control of the Macintosh between the CLIPS and HyperCard. The following examples should clarify these issues.

The ClipsX XCMD:

---

* Apple, Macintosh, and HyperCard are registered trademarks of Apple Computer, Inc.

Example 1: The use of clear.

```
-- in a HyperCard script
  ClipsX "clear"
  get the result
  if char 1 to 3 of it is not "V4." then
    -- this is probably an error
    -- so handle the error and then exit
  end if
  -- continue setting up CLIPS program
```

The "ClipsX" command handles four sub-commands as specified by the first parameter. The first of these commands is "clear". It is used to clear the CLIPS environment. This should be the first CLIPS command called from a HyperCLIPS application stack, so that any other CLIPS program in the interpreter will be excised. If the CLIPS interpreter has not been loaded then it will be loaded at this time. Many things can go wrong while loading the CLIPS interpreter: memory may become full; the file containing the interpreter may not be found; or an incompatible version of the interpreter may be loaded. So it is important to check for these errors. Any data from CLIPS may be retrieved using the HyperTalk function "the result". If everything executes as it should then the first line of the data return will be the version information. This example checks that version four has been loaded.

Example 2: The load and reset commands.

```
-- in a HyperCard script
-- assumes card field "program" contains
-- the following CLIPS program
-- (defrule start
--    (initial-fact)
--    =>
--    (fprintout t "Hello world." crlf))
  ClipsX "load",card field "program"
  ClipsX "reset"
  -- continue setting up CLIPS program
```

The second command typically used is "load". It takes a second parameter which is the text of the CLIPS program to load. The next command is "reset" which sets up the initial facts and activations in the CLIPS environment. Because of how the IO router system is set up, these routines return may return information about which rules were compiled, which facts were asserted, and which rules were activated. But this information is not usually of interest in a HyperCLIPS application so this example does not make use of the data return through "the result". It simply loads a program and makes it ready to run, assuming no errors will occur.

Example 3: The run command.

```
-- in a HyperCard script
-- assumes the CLIPS program from the previous example
-- has been loaded and is ready to run.
  ClipsX "run",empty
  get the result
  -- process the results returned from CLIPS
  get line 1 of it
  answer it with "OK"
```

The last of the four sub-commands to "ClipsX" is "run". This is the most often used command because it passes data and control to CLIPS. It takes a second parameter which is the text of the data you wish to make available to the running CLIPS program. This example passes "empty" as its second parameter because the program that is loaded does not need any extra data to do its computation. The "run" command starts the CLIPS intepreter which does not return until an error occurs or it runs out of rules to fire. In this case the interpreter will fire just the one rule and then return control back to HyperCard. Because of the way the IO router is set up, the message "Hello world." will be returned as the first line of the data returned through "the result". Processing the results usually involves parsing the data and presenting it in an appropriate fashion to the user. This example displays the message in a dialog box. The last line of the data passed back from CLIPS should say how many rules were fired. This information may be useful for debugging purposes but is of little use in the final version of an application.

The I/O router:

Example 1: Sending data back to HyperCard.

```
; in a CLIPS program
(defrule start
  (initial-fact)
  =>
  (fprintout t "Hello world." crlf))
```

This is the example that was used above and you probably already understand what happens, but it will now be explained in greater detail. The I/O router facilities of CLIPS allow the redirection of I/O from one physical location to another. In standard CLIPS, any data written to any of the logical names "stdout", "werror", or "wdisplay" will probably be written to the terminal. Whereas in a windowing version of CLIPS the data will probably be written to three different windows. This is managed by routing data sent to these logical names to different locations in each case. The HyperCLIPS I/O router handles data written to all of the standard logical names by collecting and buffering it and then passing it back to HyperCard as "the result" when the CLIPS interpreter returns. This means that in the example above the fprintout statement, which writes a message to "stdout", will make the message "Hello world." available to HyperCard when the run command completes.

Example 2: Receiving data from HyperCard

```
; in a CLIPS program
; assumes a HyperCard call such as ClipsX "run","broken"
; also assumes that this rule is on the activation list so
; that it will be fired when the run command is called
(defrule get_engine_state
  ?fact <- (get_state)
  =>
  (retract ?fact)
  (bind ?state (read))
  (assert (engine_state ?state)))
```

Receiving data from HyperCard is also handled through the I/O router system. The standard version of CLIPS normally reads data from the terminal. The HyperCLIPS I/O router reroutes reads from the "stdin" logical name (the default read location) to get characters from a memory buffer instead of the terminal. When the "ClipsX" "run"

command is called the second parameter is used to fill in this buffer. This example will read the word "broken" from the buffer and then assert the fact "engine_state broken".

Passing control between CLIPS and HyperCard:

Example 0: Passing control to CLIPS

```
-- no example needed
```

HyperCard and CLIPS do not execute concurrently. Control must be explicitly passed between the two whenever either of them needs the functionality of the other. Control is usually passed to CLIPS when HyperCard needs a computation performed. This is done with the "run" command. The CLIPS program, though, must be ready to accept control. This means that there are rules on the activation list ready to fire. Initially rules are put on the activation list by the "reset" command, but there is another method to get CLIPS ready to accept control which will be explained next.

Example 1: Passing control to HyperCard

```
; in a CLIPS program
(defrule get_data
  ?f < - (phase get_data)
  =>
  (retract ?f)
  (fprintout t "need data" crlf)
  (assert (get_data_continue))
  (halt))

(defrule get_data_continue
  ?f <- (get_data_continue)
  =>
  (retract ?f)
  (bind ?data (read))
  (assert (data ?data)))
```

Control is usually passed back to HyperCard for one of two reasons: the computation is finished; or more data is needed to complete the computation. If the computation is finished then passing control back to HyperCard is trivial: there will be no more rules to fire so CLIPS will return automatically. The case of needing more data, though, is more complex. This example shows how to give control back to HyperCard while making sure that the rule that reads the data will be ready to fire when HyperCard eventually returns control back to CLIPS. The important CLIPS function is "halt". It causes an error within CLIPS so that the interpreter will return to HyperCard, but it does not alter the activation list so that any rule that was ready to fire before the "halt" command will still be ready to fire after the "halt" command. In this way the CLIPS program is ready to accept control when HyperCard calls the "run" command with the data needed to continue the computation.

Technical difficulties implementing HyperCLIPS

Although HyperCard and CLIPS seem easily integrated through the use of their built in hooks for such reasons, there are some technical problems which make this task more difficult that it would appear. The problem is on the HyperCard side. HyperCard allows the addition of functionality in the form of XCMDs, but XCMDs have severe limitations: an XCMD cannot be larger than 32K bytes, and an XCMD cannot have global data. CLIPS breaks both of these rules and cannot, therefore, be implemented as a normal XMCD.

Both of these limitations are the result of the architecture of the Macintosh. A Macintosh application uses register A5 of the Motorola 68000 to point to the area of memory that contains the global variables and the jump table. The jump table is used to support intra-segment calls which are necessary because segments are limited to 32K and any application larger than this must be divided into multiple segments. Segments are limited in size by the longest possible branch instruction, which on the 68000 is +/- 32K. Jump instructions could be used to allow farther branches and larger segments, but this would make the code non-relocatable which is contrary to the Macintosh memory management strategy. While HyperCard is in control, register A5 points to HyperCard's global data and jump table. XCMDs cannot use this jump table or global data area, this leads to the limitations mentioned above.

The way to get around the two limitations mentioned above is obvious but tricky to implement: let the XCMD have its own jump table and globals area and make A5 point to this area while the XCMD is running. The difficulty in this is in setting up the jump table. This process is usually handled by the Segment Loader facility in the Macintosh Operating System. It interprets the information in CODE resource 0 of the application to form the jump table and globals area and then starts the program by jumping to the first entry in the jump table.

The implementation of HyperCLIPS is divided into two parts: an XCMD that duplicates the functionality of the Segment Loader and takes care of setting up the A5 register before calling the CLIPS interpreter; and a modified CLIPS interpreter stored in the format of an application file where the XCMD can find it. The only modifications to CLIPS are the code to handle the function dispatching and the I/O router to handle the communication of data.

Conclusion:

We have used HyperCLIPS to develop prototypes for device simulation and knowledge based training systems. In our experience we have found development time to be very fast. The CLIPS side of an application can be developed and debugged in the usual CLIPS environment and later be integrated with a HyperCard user interface. This final stage of integration is a little awkward because of the lack of tools for modifying CLIPS programs from within HyperCard, but we are adopting methodologies to make this step easier. Because HyperCard and CLIPS are interpreted languages, execution time for HyperCLIPS applications is rather slow. In the case of CLIPS, the results may be worth the wait, but HyperCard may need to replaced by a more efficient user interface engine in production quality applications. If a faster interface becomes necessary though, the substitution should be transparent to the CLIPS side of the application. Our future plans include looking for such an interface engine, possibly on a more powerful workstation.