S21-82
34681
p_11

N95-19646

Expert System Technologies for Space Shuttle Decision Support: Two Case Studies

Christopher J. Ortiz
Workstations Branch (PT3)
Johnson Space Center
Houston, TX 77058
chris.ortiz@jsc.nasa.gov

David A. Hasan
LinCom Corporation
1020 Bay Area Blvd., Suite 200
Houston, TX 77058
hasan@gothamcity.jsc.nasa.gov

Abstract

This paper addresses the issue of integrating the C Language Integrated Production System (CLIPS) into distributed data acquisition environments. In particular, it presents preliminary results of some ongoing software development projects aimed at exploiting CLIPS technology in the new Mission Control Center (MCC) being built at NASA Johnson Space Center. One interesting aspect of the control center is its distributed architecture; it consists of networked workstations which acquire and share data through the NASA/JSC-developed Information Sharing Protocol (ISP). This paper outlines some approaches taken to integrate CLIPS and ISP in order to permit the development of intelligent data analysis applications which can be used in the MCC.

Three approaches to CLIPS/ISP integration are discussed. The initial approach involves clearly separating CLIPS from ISP using *user-defined functions* for gathering and sending data to and from a local storage buffer. Memory and performance drawbacks of this design are summarized. The second approach involves taking full advantage of CLIPS and the CLIPS Object-Oriented Language (COOL) by using *objects* to directly transmit data and state changes from ISP to COOL. Any changes within the object slots eliminate the need for both a data structure and external function call thus taking advantage of the object matching capabilities within CLIPS 6.0. The final approach is to treat CLIPS and ISP as *peer toolkits*. Neither is embedded in the other, rather the application interweaves calls to each directly in the application source code.

Introduction

A new control center is being built at the NASA Johnson Space Center. The consolidated Mission Control Center (MCC) will eventually replace the existing control center which has been the location of manned spaceflight flight control operations since the Gemini program in the 1960s. This paper presents some preliminary results of projects aimed at incorporating knowledge-based applications into the MCC. In particular, it discusses different approaches being taken to integrate CLIPS with the MCC Information Sharing Protocol (ISP) system service.

MCC and ISP

The new control center architecture differs significantly from the current one. The most prominent difference is its departure from the mainframe-based design of the existing control center. The new MCC architecture is aggressively distributed. It consists of UNIX workstations (primarily DEC Alpha/OSF1 machines at present) connected by a set of networks running TCP/IP protocols. Exploitation of commercially available products which will be relatively easy to replace and upgrade has been a prime motivating factor in this change. Particular attention has been paid to the use of standard hardware, and commercial off-the-shelf (COTS) software is being used wherever possible.

With a mainframe computer at the center of all flight support computation, the process of presenting telemetry and computational results to flight controllers was really a matter of "pushing" the relevant data out of the mainframe onto the appropriate flight control terminals. In recent years, the task of acquiring telemetry on the system of MCC upgrade workstations has been a matter of requesting the telemetry data from the mainframe. The central computer has served as the one true broker for telemetry data and computations.

In the distributed MCC design, the notion of a central telemetry and computation broker has disappeared. In fact, terminals have disappeared. The flight control consoles consist of UNIX workstations which have access to telemetry streams on the network but must share data instead of relying on the mainframe for common computations.

Software applications running on the MCC workstations will obtain telemetry data from a system service called the Information Sharing Protocol (ISP). ISP is a client/server service. Distributed clients may request ISP support from a set of "peer" ISP servers. The servers are responsible for extracting data from the network. Client applications needing those data initiate a session with the servers (possibly from different machines), using the ISP client application program interface (API). These clients *subscribe* to data from the ISP servers, which deliver the data asynchronously to the clients as the data change. Parenthetically, the ISP API provides *data source independence.* Thus, ISP client applications may be driven by test data for validation and verification, playback data for training or live telemetry for flight support.

It is the intent of the new MCC architecture that hardware (workstations, routers, network cabling, etc...) will take a backseat to the "software platform". ISP is a prominent element of this platform, since it is the data sharing component of the system services in addition to providing a telemetry acquisition service. Indeed, the architecture presupposes that many of the functions previously handled in the mainframe program (e.g., display management, limit sensing, fault summary messages, "comm fault" detection) will now be carved up into smaller, easier to maintain application programs.

ISP supports the integration of these applications by allowing clients to receive data that have been *published* by other clients. These shared data will in many instances be computations which were previously handled internally to the mainframe, e.g., spacecraft trajectories and attitude data, but with greater frequency, the shared data are expected to be "higher order information" derived from analyses of telemetry data.

Some of the data analysis necessary for the generation of this higher order information will be derived from rule-based pattern matching of telemetry. One example of this is the Bus Loss Smart System which is used by EGIL flight controllers to identify power bus failures. Another example discussed later in this paper is the Operational Instrumentation Monitor (oimon) which assesses the health of electronic components involved in the transmission of Shuttle sensor data down to Earth.

Rule-based applications can capture the often heuristic procedures used by flight controllers to perform their duties. A number of such applications are currently under development by flight control groups where the knowledge-based approach contributes to getting the job done "better, faster and cheaper". CLIPS is being used in a number of these. In the discussion that follows, methods for combining the telemetry acquisition and data sharing functionality of ISP with the pattern matching capabilities of CLIPS are discussed.

An Embedded Approach

CLIPS is a data-driven language because it uses data in the form of facts to activate and fire if-then rules which result in a change of execution state for the computer. CLIPS was designed to be embedded in other applications thus allowing them the ability to perform complex tasks without the use of complex programming on the part of the rule developer. Since ISP is a transport protocol for both receiving and transmitting information, it only makes sense to find a way to integrate the two toolkits to provide CLIPS developers with a reliable transport mechanism for data.

In our work using ISP and CLIPS we have taken two approaches in integrating the two toolkits. The first was to directly embed ISP within CLIPS and provide a simple set of user-defined functions which allow the CLIPS developer to communicate via ISP. The second approach was to use both toolkits as peers allowing the developer to have complete control over the integration.

The first method of embedding ISP into CLIPS grew out of three separate attempts to balance data, speed and ease of use concerns for the application developer. The first attempt at integration (Figure 1) consisted of creating a separate *data layer* with which ISP and CLIPS could communicate. This layer provides a storage location for all the change only data that ISP receives as well as hiding the low level ISP implementation. This approach provided several challenges in how ISP and CLIPS would communicate with the data layer.

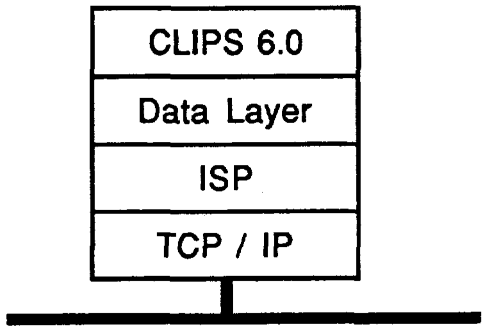

Figure 1: The first attempt at integration

A simple set of commands was added to CLIPS to aid in the communication with ISP.

```
(connect-server [CLIPS_TRUE / CLIPS_FALSE] )
(subscribe-symbol [symbol_name] [CLIPS_TRUE / CLIPS_FALSE] )
(enable-symbols [CLIPS_TRUE / CLIPS_FALSE] )
```

The ***connect-server*** command establishes or disconnects a link with the ISP server and registers the application as requesting ISP data. Likewise, ***subscribe-symbol***, informs the ISP server of which data elements are requested or no longer needed by the application. The ***enable-symbols*** command begins and ends the flow of data to the application.

Other API calls were added to CLIPS to enable an application to publish data to the ISP server for use in other CLIPS or ISP only applications.

```
(publish-symbol [symbol-name] [EXCLUSIVE])
(publish [VALUE/LIMIT/STATUS/Message] [symbol-name] [data]
[time])
(unpublish-symbol [symbol-name] )
```

The ***publish-symbol*** command informs the ISP server that the CLIPS application wants permission to send data under a given symbol-name. The optional `EXCLUSIVE` parameter informs the server that the requesting application should be the only application allowed to change the value.

The final set of APIs allows ISP to communicate with CLIPS in the form of facts.

```
(want-isp-event [CYCLE/LIMIT] [CLIPS_TRUE/CLIPS_FALSE])
```

The ***want-isp-event*** command will activate or deactivate cycle or limit facts to be published in the fact list.

This first attempt provided a solid foundation for communication between the two tool kits. However, the use of a separate data layer proved to be cumbersome to implement. The data layer needed to have a dynamic array and a quick table look-up mechanism as well as a set of functions to provide CLIPS with the ability to check values. Each time a rule needed a data value, an external function would have to be called. This proved to be a costly option in the amount of time needed to call the external function. A better method was clearly needed.

Our second attempt at integration ISP and CLIPS consisted of the elimination of the data layer (Figure 2). All of the ISP data would be fed to CLIPS via facts. This method had several advantages. First, it reused the ISP functions developed earlier. Second, it allowed CLIPS to store all ISP data as facts. Finally, CLIPS application developers could do direct pattern matching on the facts to retrieve the data.

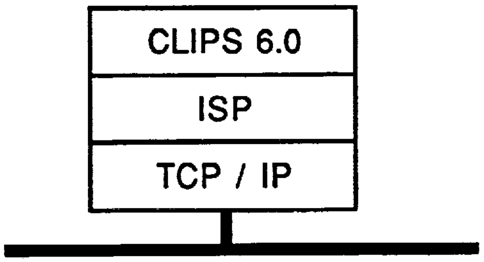

Figure 2: Removal of the data layer

The use of facts provided a few smaller challenges in the update and management of the fact list. There needed to be a way to find an old fact that had not been used and retract it when new data arrived. This could be done at the rule level or in the integration level. At the rule level, the application programmer would be required to create new rules to seek out and remove obsolete facts. At the integration level, time would be spent looking for matching facts with similar IDs to be removed.

The final approach at integration of ISP with CLIPS involved the use of the CLIPS Object-Oriented Language (COOL). Each data packet would be described as an instance of a class called MSID. The MSID class would provide a data storage mechanism for storing the data name, value, status, time tag, and server acceptance information.

```
(defclass MSID
   (is-a USER)
   (role concrete)
   (pattern-match reactive)
   (slot value ( create-accessor read-write) (default 0.0))
   (slot status ( create-accessor read-write)(default 0.0))
   (slot time (create-accessor read-write)(default 0.0))
   (slot accepted (create-accessor read-write)
     (default CLIPS_TRUE))
)
```

A second advantage of using the object implementation consists of inheriting constructors and destructors. When an instance of the MSID class is created, a constructor is activated and the symbol is automatically subscribed. On the other hand, when an symbol is no longer needed a destructor is activated and the symbol is automatically unsubscribed. Constructors and destructors free the application programmer from worrying about calling the appropriate ISP APIs for creating and deleting symbols.

```
(defmessage-handler MSID init after()
   (subscribe-symbol (instance-name ?self) CLIPS_TRUE)
)

(defmessage-handler MSID delete before()
   (enable-symbols CLIPS_FALSE)
   (subscribe-symbol (instance-name ?self) CLIPS_FALSE)
   (enable-symbols CLIPS_TRUE)
)

( definstances MSIDS
   ( S02K6405Y of MSID )
   ( S02K6205Y of MSID )
   ( S02K6026Y of MSID )
   •••
   ( S02K6078Y of MSID )
)
```

Creating and subscribing symbols are automatically handled by COOL. One of the only ISP implementation details that the application programmer needs to be concerned is to enable the symbols and to schedule which ISP events are to be handled.

```
(defrule connect
   ?fact<- (initial-fact)
=>
   (retract ?fact)
   (enable-symbols CLIPS_TRUE )
   (want-isp-event LIMIT CLIPS_FALSE )
   (want-isp-event CYCLE CLIPS_TRUE )
)
```

Another clear advantage of using objects, like facts, is the ability to do direct pattern matching. As the ISP data changes, a low level routine updates the value in the affected slot. CLIPS could then activate any rule which needed data from the changed slot and work with this information on.

```
(defrule ValueChanges
   ?msid <- (object ( is-a MSID ) ( value ?value))
=>
   (update-interface ( instance-name-to-symbol (
      instance-name ?msid)) ?value )
)
```

After working with the integration of CLIPS and ISP there is an advantage that CLIPS/COOL bring to bear on the ease of use for linking CLIPS with external 'real time' data. One such application that used this integrated technology was a prototype to display switch positions from on-board systems to Space Shuttle ground controllers. The prototype was up and running within three days. The display technology had already been developed as part of a training tool to help astronauts learn procedures for the Space Habitation Module. The display technology was then reused and combined with CLIPS and ISP to monitor telemetry and react whenever subscribed data were detected. CLIPS was needed deduce single switch settings based on the downlinked telemetry data. For example, several parameters may contain measurements of pressure across a line. If most of the pressure sensors begin to fall low, then a pressure switch might have been turned off.

An Open Toolkit Approach

So far, the discussion has focused on integration of CLIPS and ISP by embedding ISP into CLIPS. This has the advantage of hiding the details of the ISP client API from developers of CLIPS applications; however, it is easy to imagine applications which are no more "CLIPS applications" per se than they are "ISP clients". CLIPS and ISP provide distinct services, so it is not immediately obvious which ought to be embedded in the other, or whether embedding is even necessary.

The third approach we used to integrating CLIPS and ISP was implemented into the oimon application, discussed below. Instead of embedding ISP inside CLIPS, the two APIs are treated as "peers" within the application. Consider the following definitions:

- **open system**: a system which makes its services available through a callable API,
- **open toolkit**: an open system which permits the caller to always remain in control of execution.

The primary distinction between these two is that open systems may require that the caller turn over complete control of the application (e.g., by calling a `MainLoop()` function).

Open toolkits permit the caller to exploit the systems' functionality while maintaining control of the application; it real-time applications, this can be critical. Examples of open toolkits include the X-toolkit, the ISIS distributed communications system, and Tcl/Tk. Figure 3 depicts a number of the CLIPS and ISP API functions. Although both have functions which will take control of the application (i.e., `Run(-1)` and `ItMainLoop()`), the use of these functions is not required; lower level primitives are available to fine-tune execution of the two systems (i.e., `Run(1)` and `ItNextEvent()`, `ItDispatchEvent()`). By the definitions above, both CLIPS and ISP are open toolkits. As a result, they may both be embedded in a single application which chooses how and when to dispatch to each.

```
•   ISP:
-       ItInitialize()
-       ItPublish(), ItSubscribe()
-       ItConnectServer(), ItDisconnectServer()
-       ItNextEvent(), ItDispatchEvent()
•   CLIPS:
-       InitializeCLIPS()
-       AssertString()
-       Reset()
-       Run()
```

Figure 3: Elements of the CLIPS and ISP appication program interfaces

OIMON

The Operational Instrumentation Monitor (oimon) is being developed as a MCC application run at Instrumentation/Integrated Communications Officer (INCO) console workstations. The program is useful to other non-INCO flight controllers, since it publishes (via ISP) the status of certain electronic components which may affect the validity of data in the telemetry stream. The paragraphs which follow outline oimon and discuss how ISP and CLIPS have been integrated into it as peer open toolkits.

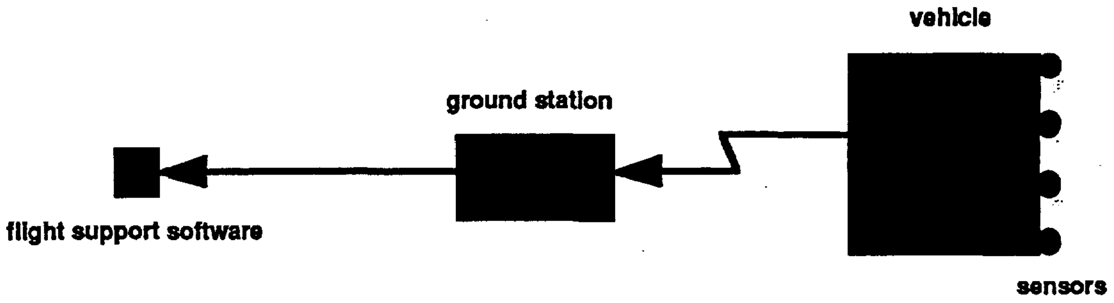

Figure 4: Shuttle sensor "channelization"

Figure 4 presents a summary of the data flow originating at sensors on board the Shuttle (e.g., temperatures, pressures, voltages, current) and ending up on a flight controller's display or in some computation used by a flight control application. The significant aspect of this figure is that there are a number of black boxes (multiplexer/demultiplexers -- MDMs and discrete signal conditioners -- DSCs) which sit in the data path from sensor to flight controller. A number of these black boxes (the so-called operational instrumentation (OI) MDMs and DSCs) are the responsibility of the INCO flight control discipline. There are other black boxes managed by other disciplines, for example the "flight critical" MDMs. The oimon application is concerned with the OI black boxes.

Failure of an OI MDM or DSC can corrupt the telemetry data for a number of on-board sensors. As a result, most flight controllers and many flight control applications are interested in the status of MDMs and DSCs. Instead of requiring that each consumer of telemetry data individually implement the logic necessary to assess MDM/DSC status, the INCO discipline will run oimon as an ISP client which publishes the status of the OI MDMs and DSCs. Any consumer of data affected by a particular OI black box may subscribe to its status through ISP and thus effectively defer OI MDM/DSC to the INCO discipline. This deferral of responsibility to the appropriate discipline is one of the benefits of a software architecture which promotes data sharing between different applications on different workstations.

The oimon application is a C-language program which makes calls to the ISP API to obtain its input data, the CLIPS API to execute the pattern matching necessary to infer the MDM/DSC statuses, and the ISP API to publish its conclusions. There are no explicit parameters in the downlist which unambiguously indicate OI black box status, and this is the reason CLIPS is needed. The major elements of oimon of relevance here are

- •a set of CLIPS rules which implement the pattern matching,
- •callback functions invoked whenever a telemetry event occurs,
- •assertions of CLIPS facts from within the callback functions, and
- •a main loop which coordinates CLIPS rule firing and ISP event dispatching.

A template of oimon is shown in Figure 5.

```
isp_event_callback(){
    ...
    sprintf( fact, ...);
    AssertString(fact);
    ...
}

main(){
    ...
    ItAddCallback( ...isp_events... );
    while(True){
            while( moreISP() ){
                    ItNextEvent( ... );
                    ItDispatchEvent(...);
            }

            while( moreCLIPS() ){
                    Run(1);
            }
    }
}
```

Figure 5: oimon code template

The CLIPS rule base is constructed so as to implement a number of tests currently used by INCO flight controllers to manually assess MDM/DSC status and to enable/disable some tests based on the results of others. In particular, these tests are (1) the MDM wrap test, (2) MDM and DSC built-in test equipment tests, (3) power bus assessment, and (4) a heuristic test developed by INCO flight controllers to deduce OI DSC status based on a handful of telemetry parameters which are connected to the DSCs through each of the DSC cards and channels.

The oimon application is still under development. However, preliminary experience with it suggests that the integration of CLIPS and ISP as peer toolkits called from a main application is not only feasible but easily implemented. Preliminary experience with this approach to CLIPS/ISP integration has revealed one advantage of it over embedding ISP in CLIPS. Under certain circumstances, the invocation of CLIPS functions can invoke the CLIPS "periodic action". When CLIPS is embedded in ISP, this can cause ISP events to "interrupt" the execution of the consequent of an ISP rule. In the peer toolkit approach, ISP functionality is not invoked using the CLIPS periodic action and thus this behavior does not exist. Subsequent testing of oimon will focus on tuning the event-dispatching/rule-firing balance to ensure that oimon is neither starved of telemetry nor prevented from reasoning due to high data rates.

Summary

This paper has outlined three approaches we took to integrating the CLIPS inference engine and the ISP client API into single applications. A summary of a "data layer" approach was given, but this approach was not actually implemented. A similar method was also described in which ISP API calls are embedded in CLIPS, and ISP event processing is handled as an ISP "periodic action". The CLIPS syntax for this approach was presented. A quick prototype was developed based on this second approach, and the prototype demonstrates the soundness of the technique. In particular, it permitted very rapid development of the application. Unlike the first two approaches, the third approach we discussed did not embed ISP in CLIPS. Rather the CLIPS and ISP APIs are invoked as "peer toolkits" in the C-based oimon application. This application is currently being tested against STS-68 flight data, but additional development is expected. Preliminary results from oimon suggest that the peer toolkit approach is also sound. The possibility of ISP events interrupting the firing of ISP rules is eliminated in the oimon approach, since ISP is invoked directly from the application instead of being called as a CLIPS periodic action.

References

Paul J. Asente and Ralph R. Swick, **X Window System Toolkit**, Digital Press, 1990.

Joseph C. Giarratano and Gary Riley, **Expert Systems Principles and Programming**, PWS Pub. Co.

Joseph C. Giarratano , **CLIPS User's Guide**, NASA JSC-25013

John K. Ousterhout, **Tcl and the Tk Toolkit**, Addison-Wesley, 1994.

G. Riley, *CLIPS: An Expert System Building Tool*, Proceedings of the Technology 2001 Conference, San Jose, CA, December 1991.

The ISIS Distributed Toolkit Version 3.0 User Reference Manual, Isis Distributed Systems, 1992.