N96-12943

Executing CLIPS Expert Systems in a Distributed Environment

James Taylor
IntelliCorp, Mountain View, California, USA

Leonard Myers
CAD Research Unit, California Polytechnic State University
San Luis Obispo, California, USA

This paper describes a framework for running cooperating agents in a distributed environment to support the Intelligent Computer Aided Design System (ICADS), a project in progress at the CAD Research Unit of the Design Institute at the California Polytechnic State University. Currently, the system aids an architectural designer in creating a floor plan that satisfies some general architectural constraints and project specific requirements. At the core of ICADS is the Blackboard Control System. Connected to the blackboard are any number of domain experts called Intelligent Design Tools (IDT). The Blackboard Control System monitors the evolving design as it is being drawn and helps resolve conflicts from the domain experts. The user serves as a partner in this system by manipulating the floor plan in the CAD system and validating recommendations made by the domain experts.

The primary components of the Blackboard Control System are two expert systems executed by a modified CLIPS shell. The first is the Message Handler. The second is the Conflict Resolver. The Conflict Resolver synthesizes the suggestions made by domain experts, which can be either CLIPS expert systems, or compiled C programs. In DEMO1 [1], the current ICADS prototype, the CLIPS domain expert systems are Acoustics, Lighting, Structural, and Thermal; the compiled C domain experts are the CAD system and the User Interface.

COMMUNICATION FRAMEWORK

The communications framework supports multiple hierarchies of connections among both C and CLIPS processes. Each connection provides an independent two-way stream communication path between processes using UNIX sockets [2]. The current network of connections demonstrates some of the possibilities (Fig. 1). From the point of view of the Blackboard Message Handler (MH), the Conflict Resolver consists of a single connected component. However, to increase performance, the rule set of the Conflict Resolver was divided into three independent rule sets and distributed as separate processes across the network. The User Interface has also been divided into two processes to take advantage of the organizational power of the Rete Network in CLIPS and the graphical display capabilities of the X Windows Tool Box.

MESSAGE HANDLER

The part of the Blackboard called the Message Handler (MH) is a CLIPS expert system with additional functions for message passing. The MH has two primary functions. First, it initializes the system by starting each IDT. Second, it distributes modified values to IDTs that request them. The MH initializes the system in two phases. During the first phase, the MH establishes a connection with the IDT to allow message passing, and receives the input requests specifying the blackboard values the IDT needs to produce its results. During the second phase, the MH builds a hash table and transmits it to each IDT to reduce future message sizes. An important prerequisite in this framework is that all system components use the same naming convention. Without a consistent naming convention, too much time would be spent converting between different representations. This common naming scheme is provided by a frame-based representation developed as part of the ICADS project [3].

REPRESENTATION

The particular-frame based representation used in ICADS is implemented as a set of CLIPS facts. A frame is a collection of information about a class or object. The information is represented in CLIPS with a frame header fact and any number of slot facts. Slots can define a particular value of the class or identify a "has-a" relation to another class.

A frame header is a fact of the form:

> (FRAME <class> <instance>) where
> FRAME is a keyword,
> <class> is the name of the class of this frame, and
> <instance> is the frame identification number.

The FRAME header is useful in performing operations on the entire frame (ie. deleting the frame), but is not needed to access the slots within the frame.

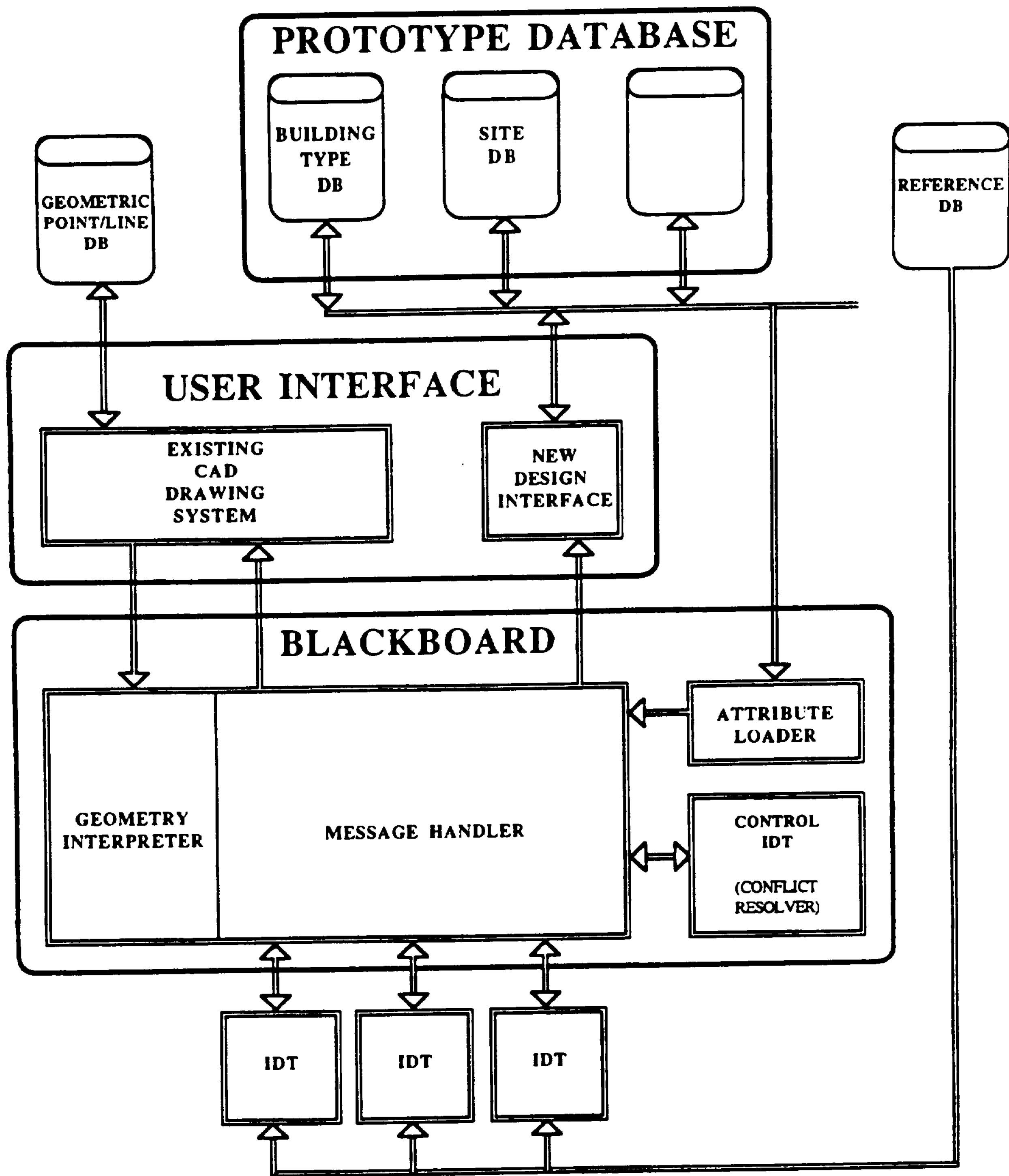

Figure 1: ICADS System Diagram

A value slot is a fact of the form:

```
(VALUE <class> <attribute> <instance> <value>) where
VALUE is a keyword,
<class> and <instance> are the same as in the frame header,
<attribute> is the slot name or attribute, and
<value> is the actual value of the slot.
```

The <value> field is one or more values, depending on the nature of the slot. For example, a slot for the coordinate of a point would have two values, whereas a slot for the length of a wall would only have one value.

A relation slot is a fact of the form:

```
(RELATION <class1> <class2> <instance1> <instance2>) where
RELATION is a keyword,
<class1> and <class2> are the names of classes, and
<instance1> and <instance2> are the frame identification
numbers of <class1> and <class2> respectively.
```

An example of an architectural object is the room or space object. Shown below is an instance of the class 'space' with an id number of 15, a name of LOBBY, a center coordinate of (128, 384), a perimeter of 108 feet, and four walls:

```
(FRAME space 15)
(VALUE space name 15 LOBBY)
(VALUE space center 15 128 384)
(VALUE space perimeter 15 108)
(RELATION space wall 15 1)
(RELATION space wall 15 2)
(RELATION space wall 15 3)
(RELATION space wall 15 4)
```

Changes to existing frames are made by inserting an action as the first field of the slot. Slots can be added, deleted, and modified using the keywords ADD, DELETE, and MODIFY. The ADD action asserts the slot. The DELETE action retracts the slot, and the MODIFY action retracts the existing slot and asserts the new slot. For example, if the above instance of a 'space' class exists and (MODIFY VALUE space area 5 216) is asserted, then the following actions occur:

```
retract (VALUE space area 5 108)
assert (VALUE space area 5 216)
retract (MODIFY space area 5 216)
```

When the DELETE action is asserted with the frame header, the entire frame (ie. all slots and the header) is retracted.

EXTERNAL FUNCTIONS

The external functions added to CLIPS to implement message passing are divided into two categories -- initialization and transmission. Messages are composed of any number of slots (ie. CLIPS facts), and are received explicitly with an external function that asserts the slots in the message. Messages are built with commands that have been added to the standard CLIPS command set and have the same syntax as the CLIPS 'assert' command.

INITIALIZATION FUNCTIONS

The functions used during initialization are briefly described below:

(new_server <name of process>):
Called by the MH and IDTs to create a server to allow future connection. Returns zero if no errors occurred.

(connect_bb [<name of message handler>]):
Called by an IDT to establish a two way connection between the IDT and the MH. Returns IDT identification number. If no argument is present, the IDT identification number is returned.

(accept_idt):
Called by the MH to establish a two way connection between the MH and an IDT. Returns IDT identification number.

(unaccept_idt <IDT id number>):
Called by the MH to terminate the connection between the MH and the IDT specified. Returns zero if no errors occurred.

(insert_hstring <field1> <field2> ...)
Called by the MH and IDTs to add a string composed of the concatenated fields to the hash table. Returns zero if no errors occurred.

TRANSMISSION FUNCTIONS

The functions used during the transmission of facts are briefly described below:

(receive_message [<IDT id number>]):
Called by MH and IDTs to receive a message in FIFO order and assert the facts in the message. Receives a message from only the MH, if zero is supplied as the IDT id number. Receives a message from only the IDT specified, if IDT id number is supplied. Returns zero if no errors occurred.

(bb_assert (<fact 1>) [(<fact 2>) ...]):
Called by IDTs to add facts to the message buffer. Uses the same syntax as the CLIPS 'assert' command. Returns zero if no errors occurred.

(bb_end_message):
Called by IDTs to send the message buffer built with the bb_assert command to the MH. Returns zero if no errors occurred.

(idt_assert <IDT id number> (<fact 1>) [(<fact 2>) ...]):
Called by MH to add facts to the message buffer of the IDT specified. Separate message buffers are maintained to allow messages for different IDTs to be built simultaneously. Returns zero if no errors occurred.

(idt_end_message <IDT id number>):
Called by MH to send the message buffer built with the idt_assert command to the IDT specified. Returns zero if no errors occurred.

INITIALIZATION

The Message Handler (MH) has two phases of initialization. In the first phase, it starts each IDT, establishes a connection to allow message passing, and receives input requests specifying the slots an IDT requires as input. Each IDT sends its input requests as its first message in the form of 'input' value slots in an 'idt' frame. The following example demonstrates the actions performed by the MH and two IDTs during the first phase:

```
        MESSAGE HANDLER
(new_server "mhandler")
(system "sound.start")
(receive_message (accept_idt))
(system "light.start")
(receive_message (accept_idt))
```

```
        SOUND IDT
(new_server "sound")
(bind ?no (connect_bb "mhandler"))
(bb_assert
        (ADD FRAME idt ?no)
        (ADD VALUE idt input ?no FRAME space)
        (ADD VALUE idt input ?no FRAME space name)
        (ADD VALUE idt input ?no FRAME space area))
(bb_end_message)
```

```
        LIGHT IDT
(new_server "light")
(bind ?no (connect_bb "mhandler"))
(bb_assert
        (ADD FRAME idt ?no)
        (ADD VALUE idt input ?no FRAME wall)
        (ADD VALUE idt input ?no VALUE wall length)
```

```
        (ADD VALUE idt input ?no RELATION wall window))
(bb_end_message)
```

As shown above, an optional argument is supplied to receive_message to specify that the next message be received only from the most recently started IDT. This prevents messages sent by previously started IDTs from being mistakenly received and interpreted as the input requests for the most recently started IDT.

In the second phase of initialization, the MH builds a hash table to decrease the percentage of time spent transmitting messages by reducing the amount of information sent across the network. This technique reduces message sizes by a factor of four or five. The MH builds the hash table from the input requests of the IDTs. The keyword and class name fields of the input request slots are concatenated into a string and entered into a hash table. Then, when an instance of that slot is added to the message buffer with bb_assert or idt_assert, the string of consecutive words starting with the second field is converted to a hash code, transmitted across the network as an integer, and then converted back to the original string of words upon receipt. If the string cannot be found in the hash table, each field is transmitted as a sequence of separate words. To insure that the hash code is correctly converted back to the original fields, the MH and all IDTs must have identical hash tables. Thus, even though an IDT may never receive a particular slot, the slot name is still contained in the hash table of the IDT.

Using the example from Phase I, the following strings would be entered into the hash table of the MH, the sound IDT, and the light IDT:

```
(insert_hstring FRAME space)
(insert_hstring VALUE space name)
(insert_hstring VALUE space area)
(insert_hstring FRAME wall)
(insert_hstring VALUE wall length)
(insert_hstring RELATION wall window)
```

When the slot shown below is added to the message buffer, the second, third, and fourth fields (ie. VALUE space name) are converted to a single integer hash code, sent across the network, and converted back to the original three fields upon receipt of the message.

```
(bb_assert (MODIFY VALUE space name 5 RECEPTION))
```

DISTRIBUTION

After initialization, the basic loop of the MH receives the next available message, distributes the slots of the message to the IDTs that request them, and then retracts the slots. The following rules accomplish this for VALUE slots:

```
(defrule receive-message
        (declare (salience 40))
        ?f <- (RECEIVE)
        =>
        (retract ?f)
        (receive_message)
)



(defrule build-message
        (declare (salience 30))
        (VALUE idt input ?no VALUE ?class ?attribute)
        (?action VALUE ?class ?attribute ?instance $?value)
        =>
        (idt_assert ?no (?action VALUE ?class ?attribute ?instance $?value))
        (assert (SEND FRAME idt ?no))
)

(defrule send-message
        (declare (salience 20))
        ?f <- (SEND FRAME idt ?no)
        =>
        (retract ?f)
        (idt_end_message ?no)
)

(defrule loop-rule
        (declare (salience 10))
        (not (RECEIVE))
        =>
        (assert (RECEIVE))
)
```

Similar rules send the FRAME header and RELATION slots.

Assertion of (DELETE FRAME idt <IDT id number>) causes the MH to retract the frame and terminate the connection of the IDT specified. This fact must be asserted for an IDT to exit prior to receipt of (KILL) without causing an error. Assertion of (KILL) causes the MH to distribute this fact to all of the connected IDTs and then exit. The IDTs exit upon receipt of this fact.

COMMUNICATION ARCHITECTURE

There are three levels of C modules below the actual IDT in the communication architecture (Fig. 2).

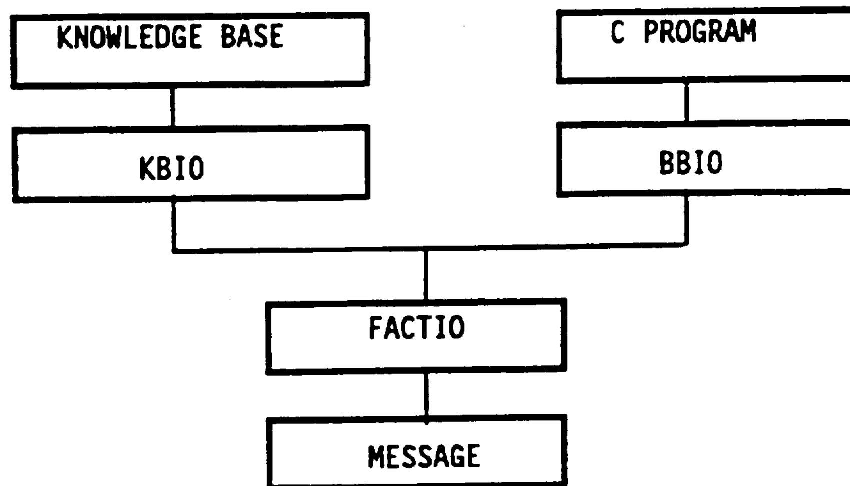

Figure 2: Levels of C Modules in Communication Hierarchy

At the lowest level in the hierarchy is the MESSAGE module which implements transmission of information between distributed processes using UNIX sockets. This module takes care of mapping the logical name supplied by a process into a network address, creating and binding the socket to this address, establishing multiple connections to a single socket, and receiving facts from distributed processes in first-in-first-out order. The next level in the hierarchy is the FACTIO module which implements reading and writing of the elements in a CLIPS facts. This module hides the representation and means of transmission of the fact. The next level in the hierarchy depends on the language in which the IDT is written. CLIPS knowledge bases use KBIO, while C programs (ie. CAD system, User Interface) use BBIO. Both modules implement establishing a two way connection between the MH and an IDT, and the hashing and unhashing of the static fields of frame slots. The KBIO module allows facts to be transmitted using the same syntax as the CLIPS 'assert' command. The BBIO module allows facts in the frame format to be transmitted with a single C function call.

CONCLUSION

ICADS DEMO1 is currently very stable. However, for the system to become usable in a professional setting, the response time needs to be much faster. Presently, the response time is slow because of the large size of the knowledge bases. The response time could be increased by dividing the large IDTs into multiple rule sets, and adding an expert system to coordinate them. The communications framework supports this creation of multiple hierarchies of expert systems.

An IDT should be divided into rule sets that are as independent of each other as possible. This will minimize the transmission and subsequent

assertion of local facts between the sub-IDTs. In addition, one slow sub-IDT will not affect the calculation of results from the other sub-IDTs. Optimumly, the facts produced by the sub-IDTs will be blackboard values to be passed directly from the coordinating IDT back to the Blackboard Message Handler.

The IDT would control its sub-IDTs using the same technique as the Blackboard Message Handler. The multiple rule sets would be coordinated by their own message handler. All communication among the rule sets would go through this message handler. Only this message handler would be connected to the Blackboard Message Handler, allowing the IDT to continue to be treated as a single connected component.

Based on run-time profiles of ICADS DEMO1, the percentage of time spent in communication (5 percent) is insignificant compared to the percentage of time spent managing expert system execution (75 percent). The functions which are taking the highest percentage of time are join_compute, find_id, and request_block. The execution time of all these functions would decrease with smaller rule sets. The savings gained from dividing large knowledge bases outweighs the added overhead for the necessary communication.

The slowest and thus the most logical system to divide is the Conflict Resolver. This knowledge base is the largest with over 250 rules. It would be divided into three relatively independent rule sets: no conflict, direct conflict, and indirect conflict. The no conflict division would have rules to post a blackboard value which only one IDT produces. The direct conflict division would have rules to decide the blackboard value based on suggestions for that value from more than one IDT. The indirect conflict division would have rules to infer a blackboard value from a set of other blackboard values. The coordinating expert system for these divisions would be implemented using the same rules contained in the Blackboard Message Handler.

The Conflict Resolver is the largest and most complex knowledge base, and thus would need to be divided first. However, in the future, each IDT will be expanded to produce more in depth analysis and simulation, and thus become larger and slower. When this time comes, these expanded IDTs will also need to be divided.

REFERENCES

1. Pohl, J., L. Myers, A. Chapman, J. Cotton (1989); ICADS: Working Model Version 1; Tech. Report, CADRU-03-89, CAD Research Unit, Design Institute, Cal Poly, San Luis Obispo, California.

2. Kernighan, B. and R. Pike (1984); The UNIX Programming Environment; PrenticeHall.

3. Assal, H. and L. Myers (1990); An Implementation of a Framebased Representation in CLIPS; Proc. First CLIPS Users Conference, Houston, Texas.