CLIPS: AN EXPERT SYSTEM BUILDING TOOL

Gary Riley
Software Technology Branch
NASA Johnson Space Center
Mail Stop PT4
Houston, TX 77058

ABSTRACT

Expert systems are computer programs which emulate human expertise in well defined problem domains. The potential payoff from expert systems is high: valuable expertise can be captured and preserved, repetitive and/or mundane tasks requiring human expertise can be automated, and uniformity can be applied in decision making processes. The C Language Integrated Production System (CLIPS) is an expert system building tool, developed at the Johnson Space Center, which provides a complete environment for the development and delivery of rule and/or object based expert systems. CLIPS was specifically designed to provide a low cost option for developing and deploying expert system applications across a wide range of hardware platforms. The commercial potential of CLIPS is vast. Currently, CLIPS is being used by over 3,300 individuals throughout the public and private sector. Because the CLIPS source code is readily available, numerous groups have used CLIPS as the basis for their own expert system tools. To date, three commercially available tools have been derived from CLIPS. In general, the development of CLIPS has helped to improve the ability to deliver expert system technology throughout the public and private sectors for a wide range of applications and diverse computing environments.

INTRODUCTION

Conventional programming languages, such as FORTRAN and C, are designed and optimized for the procedural manipulation of data (such as numbers and arrays). Humans, however, often solve complex problems using very abstract, symbolic approaches which are not well suited for implementation in conventional languages. Although abstract information can be modeled in these languages, considerable programming effort is required to transform the information to a format usable with procedural programming paradigms.

One of the results of research in the area of artificial intelligence has been the development of techniques which allow the modeling of information at higher levels of abstraction. These techniques are embodied in languages or tools which allow programs to be built that closely resemble human logic in their implementation and are therefore easier to develop and maintain. These programs, which emulate human expertise in well defined problem domains, are called expert systems. The availability of expert system tools has greatly reduced the effort and cost involved in developing an expert system.

The C Language Integrated Production System (CLIPS) [1, 2] is an expert system tool developed by the Software Technology Branch at NASA's Johnson Space Center. The prototype of CLIPS, version 1.0, was developed in the spring of 1985 in a UNIX environment. Subsequent development of CLIPS greatly improved its portability, performance, and functionality. The first release of CLIPS, version 3.0, was in July of 1986. The latest version of CLIPS, version 5.1, was released in October of 1991. A version of CLIPS written entirely in Ada, CLIPS/Ada, has also been developed. CLIPS is currently available to the general public through the Computer Software Management and Information Center (see appendix).

KEY FEATURES OF CLIPS

CLIPS was designed to address several issues key to NASA. Among these were the ability to run on a wide variety of conventional hardware platforms, the ability to be integrated with and embedded within conventional software systems, and the ability to provide low cost options for the development and delivery of expert systems.

CLIPS is written in C for portability and speed and has been installed on many different computers without changes to the source code. At the time of its original development, CLIPS was one of the few tools that was written in C and capable of running on a wide variety of conventional platforms. CLIPS can be ported

to any system which has an ANSI compliant C compiler including personal computers (IBM PC compatibles, Macintosh, Amiga), workstations (Sun, Apollo, NeXT), minicomputers (VAX 11/780, HP9000-500), Mainframes (IBM/370), and supercomputers (CRAY).

Figure 1. CLIPS is Easily Ported From One Environment to Another

To maintain portability, CLIPS utilizes the concept of a portable kernel. The kernel represents a section of code which utilizes no machine dependent features (see Figure 2). The inference engine contains the key functionality of CLIPS and is used to execute an expert system. Access functions allow CLIPS to be embedded within other systems. This allows an expert system to be called as a subroutine (representing perhaps only one small part of a much larger program). In addition, information stored in CLIPS can be accessed and used by other programs. Integration protocols allow CLIPS to utilize programs written in other languages such as C, FORTRAN, and Ada. Integration guarantees that an expert system does not have to be relegated to performing tasks better left to conventional procedural languages. It also allows existing conventional code to be utilized. The CLIPS language can also be easily extended by a user through the use of the integration protocols.

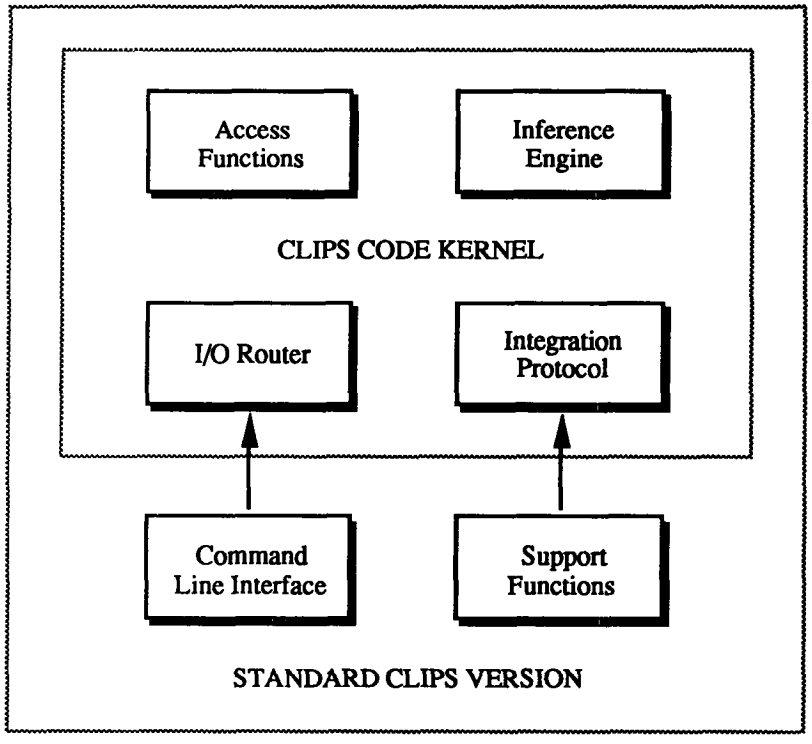

Figure 2. The CLIPS Code Kernel

To provide machine dependent features, such as windowed interfaces or graphics editors, CLIPS provides fully documented software hooks which allow machine dependent features to be integrated with the kernel. The I/O router system allows interfaces to be layered on top of CLIPS without making changes to the CLIPS kernel. The standard interface for CLIPS is a simple, text-oriented, command prompt. However, three interfaces are also provided with CLIPS that make use of the I/O router system and integration protocols to provide machine specific interfaces. These interfaces are provided for Apple Macintosh systems, IBM PC MS-DOS compatible systems, and X Window systems. Figure 3 shows the CLIPS interface for the Macintosh computer.

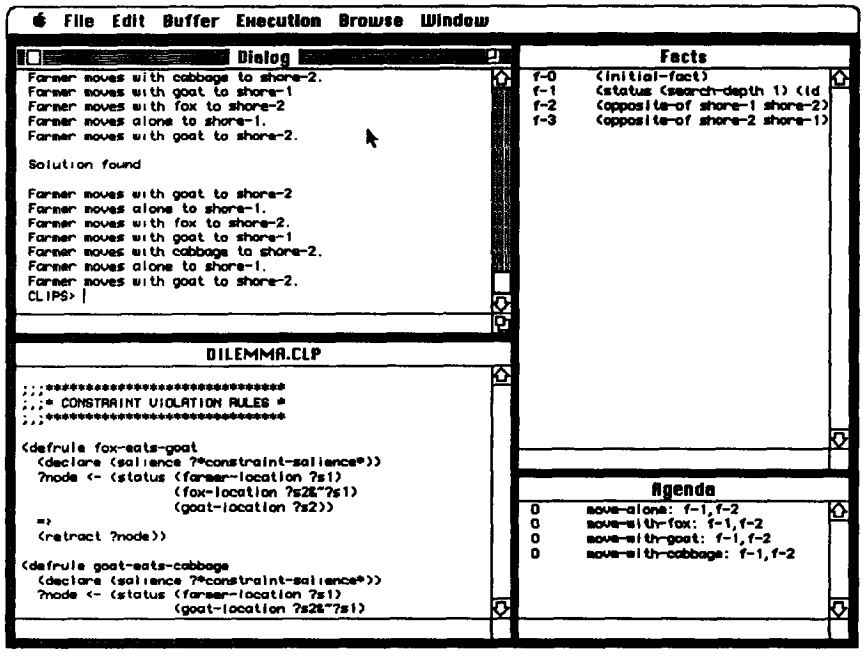

Figure 3. CLIPS Macintosh Interface

One of the key appeals of the CLIPS language results from the availability of the approximately 40,000 lines of CLIPS source code. Because the development of an expert system tool can require many man-years, the benefits of using CLIPS as a starting point for research and the creation of special purpose expert system tools cannot be understated. CLIPS users have enjoyed a great deal of success in adding their own language extensions to CLIPS due to the source code availability and its open architecture [3, 4, 5, 6, 7, 8]. Many users have also developed their own interfaces and interface extensions [9, 10, 11, 12].

KNOWLEDGE REPRESENTATION

Expert system tools are designed to provide highly productive environments by allowing knowledge to be represented flexibly. A flexible representation scheme allows the application developers to try several different approaches or to use an approach best suited to their problem. CLIPS provides a cohesive tool for handling a wide variety of knowledge with support for three different programming paradigms: rule-based, object-oriented, and procedural. In addition, CLIPS also supports the concepts of iterative refinement (refining an expert system with small iterative changes) and rapid prototyping (demonstrating proof of concept) which are found in many expert system tools.

Rule-Based Programming

The first (and originally the only) programming paradigm provided by CLIPS is rule-based programming. In this programming paradigm, rules are used to represent heuristics, or "rules of thumb", which specify a set of actions to be performed for a given situation. A rule is composed of an *if* portion and a *then* portion. The *if* portion of a rule is a series of patterns which specify the facts (or data) which cause the rule to be applicable. The process of matching facts to patterns is called pattern matching. CLIPS provides a mechanism, called the inference engine, which automatically matches facts against patterns and determines which rules are applicable. The *if* portion of a rule can actually be thought of as the *whenever* portion of a rule since pattern matching always occurs whenever changes are made to facts. The *then* portion of a rule is the set of actions to be executed when the rule is applicable. The actions of applicable rules are executed when the CLIPS inference engine is instructed to begin execution. The inference engine selects a rule and then the actions of the selected rule are executed (which may affect the list of applicable rules by adding or removing facts). The inference engine then selects another rule and executes its actions. This process, illustrated by Figure 4, continues until no applicable rules remain.

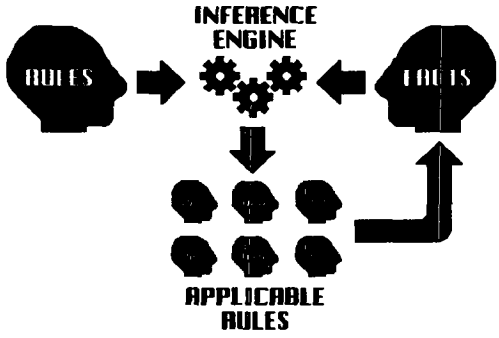

Figure 4. Execution of a Rule-Based Program

To illustrate the advantages of rule-based programming, consider the problem of monitoring a series of sensors. The following example program written in the C programming language illustrates how these sensors could be monitored using a procedural programming paradigm to determine if any two of the sensors have *bad* values (which a hypothetical expert indicates represents an overheated device).

```
#define BAD 0
#define GOOD 1
#define DEVICE_OVERHEATED 0
#define DEVICE_NORMAL 1

int CheckSensors(sensorValues,numberOfSensors)
  int sensorValues[];
  int numberOfSensors;
  {
   int firstSensor, secondSensor;

   for (firstSensor = 1;
        firstSensor <= numberOfSensors;
        firstSensor++)
      {
       for (secondSensor = 1;
            secondSensor <= numberOfSensors;
            secondSensor++)
         {
          if ((firstSensor != secondSensor) &&
              (sensorValues[firstSensor] == BAD) &&
              (sensorValues[secondSensor] == BAD))
            { return(DEVICE_OVERHEATED); }
         }
      }

    return(DEVICE_NORMAL);
   }
```

The *CheckSensors* function is implemented by storing the values of the sensors as integers in an array and then using two *for* loops to compare all combinations to determine if any two sensors have *bad* values. This function is relatively efficient if the sensors only need to be checked once. However, if this check is performed each time a sensor's value is changed, then all possible combinations are rechecked which is inefficient. In addition, the programmer has the responsibility for calling this function whenever an update is made to a sensor's value. An additional function could be written to check only one sensor against all other sensors, however, this increases the burden on the programmer. For contrast, the equivalent CLIPS code for a rule which performs the same task is shown following.

```
(defrule Two-Sensors-are-Bad
   (Sensor (ID-number ?id)   (status Bad))
   (Sensor (ID-number ~?id) (status Bad))
   =>
   (assert (Device (status Overheated))))
```

The first line of the rule contains the keyword *defrule* which indicates that a rule is being defined. The symbol *Two-Sensors-are-Bad* is the name of the rule. The next two lines beginning with the symbol *Sensor* are the patterns that form the *if* portion of the rule. Essentially, the first pattern searches for any *Sensor* fact that contains a *status* value of *Bad* and the second pattern searches for another *Sensor* fact with a *status* value of *Bad* that does not have the same *ID-number* as the *Sensor* fact matching the first pattern. The => symbol serves to separate the *if* portion of the rule from the *then* portion of the rule. Finally, the *assert* command in the *then* portion of the rule creates a new fact which indicates that the device has overheated.

Because of the overhead associated with the inference engine and the generality provided through pattern matching, a rule-based program generally does not execute as quickly as a procedural program. However, significantly less code is required and the programmer does not have to explicitly check for applicable rules when sensor values are changed. Rules are always looking for new facts which satisfy their conditions. Indeed, careless implementation of pattern matching capabilities in a procedural language may result in a program which runs much less efficiently than its rule-based counterpart. CLIPS's inference engine is based on the Rete algorithm [13] which is an extremely efficient algorithm for pattern matching.

Object-Oriented Programming

The second programming paradigm provided by CLIPS is object-oriented programming. This programming paradigm allows complex systems to be modelled as modular components (which can be easily reused to model other systems or to create new components). Object-oriented programming encompasses a number of concepts including data abstraction (the ability to define complex objects using high level representations), encapsulation (the ability to hide the implementation details of an object, thereby increasing its modularity and potential for reuse), inheritance (the ability to define new classes of objects by reusing existing classes), and polymorphism (the ability of different objects to respond to the same "command" in specialized ways).

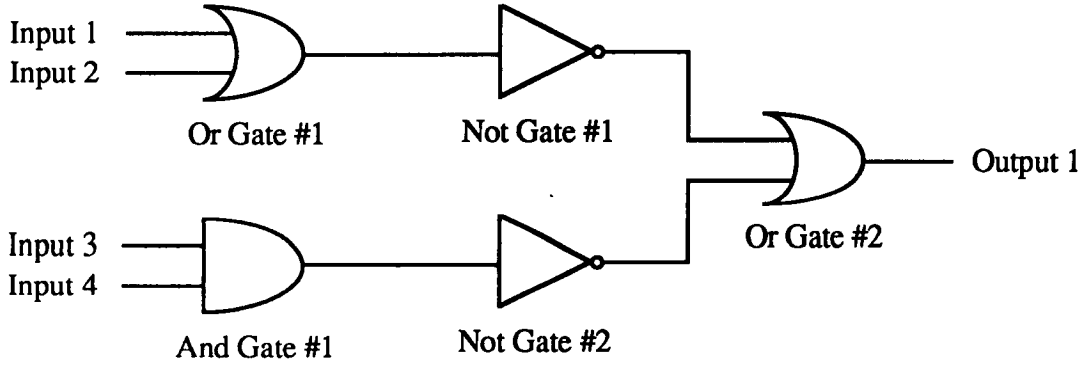

Figure 5. A Simple Electronic Circuit

Figure 5 shows a diagram of an electronic circuit consisting of *and*, *or*, and *not* gates. In electronics, a gate is a circuit that has an output dependent on some function of its input. The gates shown in Figure 5 all have boolean inputs and boolean output values. Physically, these boolean values would correspond to high and low voltages. Conceptually, these boolean values could be considered as *On* and *Off* or *True* and *False*. An *and* gate has an output value of *True* if all of its inputs are *True*, otherwise its output value is *False*. An *or* gate has an output value of *True* if any of its inputs are *True*, otherwise its output value is *False*. A *not* gate has an output value of *True* if its input value is *False* and an output value of *False* if its input value is *True*. In Figure 5, if *Input 1* and *Input 2* are both *False* and *Input 3* and *Input 4* are both *True*, then the output of *Or Gate #1* would be *False* and the output of *And Gate #1* would be *True*. The output of *Not Gate #1* would be *True* since its input (the output of *Or Gate #1*) is *False*. The output of *Not Gate #2* would be

False since its input (the output of *And Gate #1*) is *True*. Finally, *Output 1* from *Or Gate #2* would be *True* since at least one of its inputs (the output from *Not Gate #1*) is *True*.

Using object-oriented programming methodologies, it is relatively easy to model the behavior of the electronic circuit shown in Figure 5. The first step in modelling the circuit is to define classes which can be used to describe the gates used in the circuit. Since all of the gates might have some attributes in common (such as a part number), it would be useful to first define a *Gate* class. Another class, *One Input*, could be used to describe the attributes associated with a single input gate (such as a *not* gate). Since a two input gate is essentially a one input gate with an additional input, the *Two Input* class could inherit the attributes of the *One Input* class and then define additional attributes for the second input. Similarly, a *One Output* class and *Two Output* class could also be defined. Figure 6 illustrates the basic classes used to describe the gate circuits in Figure 5. The classes described illustrate the basic concepts of data abstraction and inheritance. Note that even though the circuit gates shown in Figure 5 would not need to utilize the *Two Output* class, other types of gates could utilize this class. For example, a *splitter* gate (which splits its one input into two identical outputs) could make use of this class.

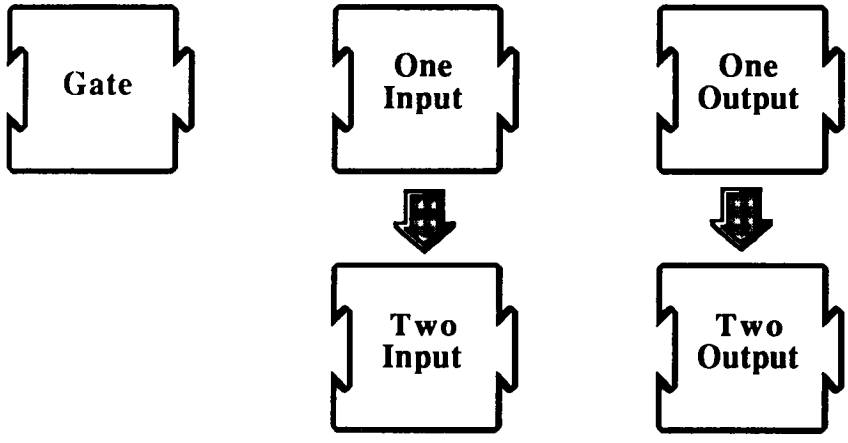

Figure 6. Classes Used to Describe Electronic Circuit Gates

Once the base classes for the gates are defined, it is possible to describe the gates in terms of these classes. Figure 7 conceptually illustrates how this could be done for the *not* gate and the *and* gate. The type of inheritance shown in Figure 7 is called multiple inheritance since a single class is inheriting attributes from more than one class. For example, the *And Gate* class inherits attributes from the *Two Input*, *One Output*, and *Gate* classes. In contrast, the inheritance shown in Figure 6 is called single inheritance since a single class inherits attributes from at most one other class (such as the *Two Input* class inheriting attributes from the *One Input* class). Some object-oriented programming languages support only single inheritance. CLIPS provides support for full multiple inheritance.

The following code shows how the gate classes could actually be defined in the CLIPS programming language (assuming the base classes described previously have been defined).

```
(defclass Not-Gate
  (is-a One-Input One-Output Gate))

(defclass And-Gate
  (is-a Two-Input One-Output Gate))

(defclass Or-Gate
  (is-a Two-Input One-Output Gate))
```

Each class definition contains the keyword *defclass* which indicates that a class is being defined. This keyword is followed by the name of the class. The next line of each class definition indicates the classes from which the class being defined will inherit attributes. Inheritance is specified using the *is-a* keyword. If

desired, additional attributes or slots of a class can be defined after the inheritance is specified. For example, the *Two Input* class might be defined as shown in the following code.

```
(defclass Two-Input
  (is-a One-Input)
  (slot Input-2))
```

Once the gate classes have been defined, it is possible to define instances (or objects) of these classes. For example, *Or Gate #1* would be a specific instance of the *Or-Gate* class as would *Or Gate #2*. It would have its own data areas for storing its input and output values. Thus a class serves as the prototypical definition which is used for creating objects belonging to that class.

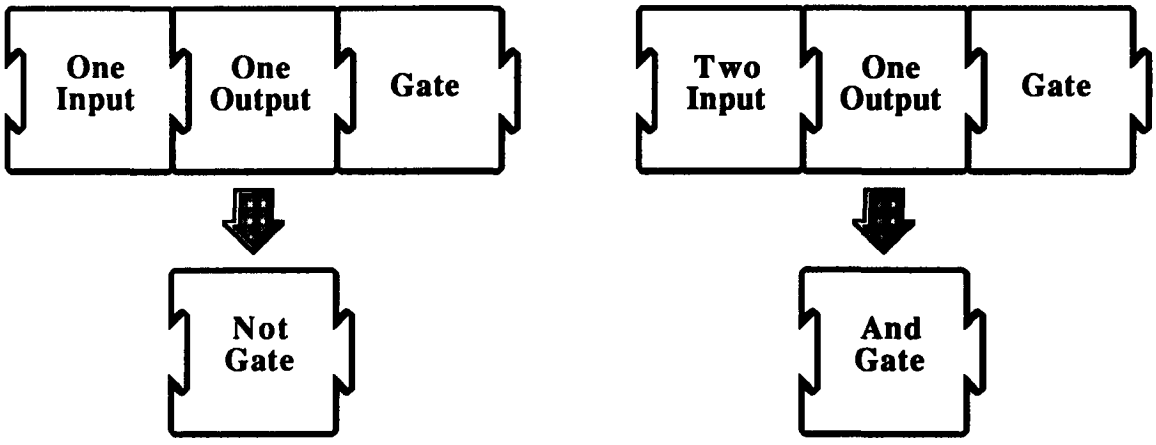

Figure 7. Building New Classes from Existing Classes

In CLIPS, objects are manipulated by sending them messages which specify an action to perform. For the circuit example, an appropriate action might be to recompute the output of a gate based upon its inputs. Notice that even though the *or* gates and *and* gates are both *Two-Input One-Output Gates*, their outputs are computed differently. In object-oriented programming, procedures as well as data can be associated with objects. Rather than writing one routine to compute the output values for all gate types given their inputs, the routines for computing outputs for objects can be encapsulated inside the classes themselves. When an *or* gate is sent a *Compute Output* message, its output is computed to be *True* if either of its inputs are *True*, otherwise its output is *False*. When an *and* gate is sent a *Compute Output* message, its output is computed to be *True* if both of its inputs are *True*, otherwise its output is *False*. Thus, both objects respond differently, yet appropriately, to the same message. This behavior is the essence of polymorphism and is illustrated by Figure 8. The procedures attached to classes are referred to as message-handlers. The following CLIPS code shows how message-handlers could be defined for the gate classes.

```
(defmessage-handler Not-Gate Compute-Output ()
   (put Output-1 (not (get Input-1))))

(defmessage-handler And-Gate Compute-Output ()
   (put Output-1 (and (get Input-1) (get Input-2))))

(defmessage-handler Not-Gate Compute-Output ()
   (put Output-1 (or (get Input-1) (get Input-2))))
```

Each message-handler definition contains the keyword *defmessage-handler* which indicates that a message-handler is being defined. This keyword is followed by the name of the class for which the message-handler is being defined and then the name of the message handled by the message-handler. The next line of each definition contains the single action performed by these message-handler. In general, message-handlers can perform as many actions as are required to complete their task. The task of computing the output requires only one action. The *put* function is used to change the value of an object's attribute.

For each of these message-handlers, the attribute being changed is the *Output-1* attribute. The value to which this attribute is changed varies for each message-handler, but is based on the inputs of the object. The *get* function is used in the message-handlers to retrieve the values of the *Input-1* and *Input-2* attributes. The message-handlers for the *Not-Gate*, *And-Gate*, and *Or-Gate* classes use the *not*, *and*, and *or* functions respectively to compute the correct output value based on their inputs. When writing CLIPS code, a prefix notation is used for calling functions that is very similar to the LISP programming language (even though CLIPS is written in the C programming language).

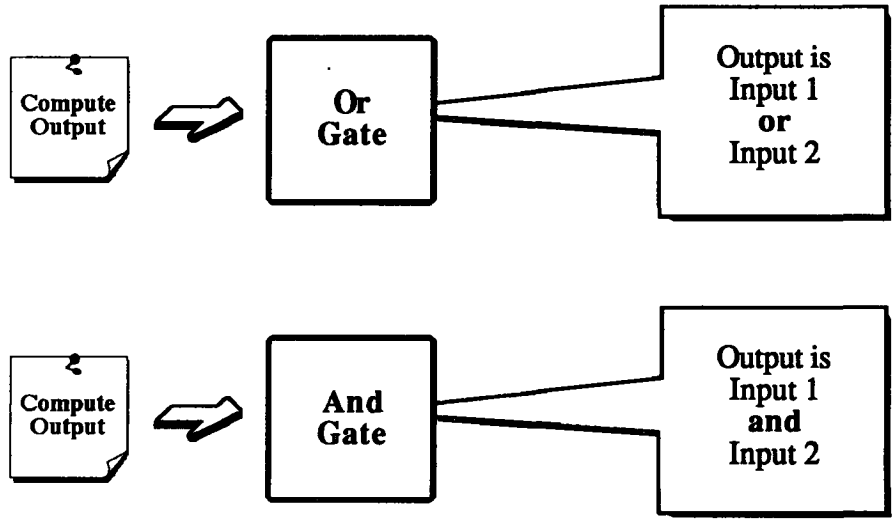

Figure 8. Two Different Objects Responding Differently to the Same Message

Procedural Programming

The third programming paradigm provided by CLIPS is procedural programming. This programming paradigm provides capabilities similar to those found in languages such as C, Pascal, Ada, and LISP. With respect to building expert systems, these are the least interesting capabilities provided by CLIPS. However, the ability to define procedural code directly within CLIPS allows new procedural capabilities to be added to CLIPS without the need of a compiler or linker. To add new capabilities to CLIPS which have been written in languages such as C, FORTRAN, or Ada, a compiler and linker are required to recompile and relink the new source code with the CLIPS source code. CLIPS allows the definition of global variables, functions, and generic functions. Generic functions are the most interesting feature of the CLIPS procedural programming language in that they allow different pieces of procedural code to be executed depending upon the arguments used when calling a function. This capability is called function overloading. As an example, the addition function could be overloaded so that numeric data types are numerically added and string data types are concatenated.

CURRENT USES

Although CLIPS was originally developed to aid in the construction of aerospace related expert systems, it has been put to widespread usage in a number of fields. CLIPS is being used by over 3,300 users throughout the public and private community including: all NASA sites and branches of the military, numerous federal bureaus, government contractors, 170 universities, and many companies. At the First and Second CLIPS Conferences held in August 1990 and September 1991 respectively, over 120 papers were presented on a diverse range of topics. In addition to aerospace and engineering applications, some other examples of CLIPS applications include: software engineering [14], network security [15], genetics [16], medicine [17], botany [18], and agriculture [19]. To date, three commercially available tools have been derived from CLIPS.

CONCLUSION

Because of its portability, extensibility, capabilities, and low-cost, CLIPS has received widespread acceptance throughout the government, industry, and academia. The development of CLIPS has helped to improve the ability to deliver expert system technology throughout the public and private sectors for a wide range of applications and diverse computing environments.

REFERENCES

1. *CLIPS Reference Manual*, Version 5.1, NASA document JSC-25012, Houston, TX., September 1991.

2. Giarratano, J., and Riley, G. *Expert Systems: Principles and Programming*, Boston, PWS-KENT, 1989.

3. Snyder, J., and Chirica, L. "An SQL Generator for CLIPS," Proceedings of the First CLIPS Conference, Houston, TX., August 1990.

4. Bhatnagar, H., Krolak, P., McGee, B., and Coleman, J. "A Neural Network Simulation Package in CLIPS," Proceedings of the First CLIPS Conference, Houston, TX., August 1990.

5. Orchard, R., and Diaz, A. "BB_CLIPS: Blackboard Extensions to CLIPS," Proceedings of the First CLIPS Conference, Houston, TX., August 1990.

6. Homeier, P., and Le, T. "ECLIPS: An Extended CLIPS for Backward Chaining and Goal-Directed Reasoning," Proceedings of the Second CLIPS Conference, Houston, TX., September 1991.

7. Adler, R. "Integrating CLIPS Applications into Heterogeneous Distributed Systems," Proceedings of the Second CLIPS Conference, Houston, TX., September 1991.

8. Boyle, C., and Schuette, J. "Debugging Expert Systems using a Dynamically Created Hypertext Network," Proceedings of the Second CLIPS Conference, Houston, TX., September 1991.

9. Pickering, B., and Hill, R. "HyperCLIPS: A HyperCard Interface to CLIPS," Proceedings of the First CLIPS Conference, Houston, TX., August 1990.

10. Callegari, A. "Integrating Commercial Off-the-shelf (COTS) Graphics and Extended Memory Packages with CLIPS," Proceedings of the First CLIPS Conference, Houston, TX., August 1990.

11. Jenkins, J., Holbrook, R., Shewhart, M., Crouse, J., and Yarost, S. "CLIPS Application User Interface for the PC," Proceedings of the Second CLIPS Conference, Houston, TX., September 1991.

12. Feagin, T. "On the Generation of Graphical Objects and Images From Within CLIPS Using XView," Proceedings of the Second CLIPS Conference, Houston, TX., September 1991.

13. Forgy, C. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," Pages 17-37, Artificial Intelligence 19, (1982).

14. Morris, K. "Automating Symbolic Analysis with CLIPS," Proceedings of the First CLIPS Conference, Houston, TX., August 1990.

15. Miller, M., Barr, S., Gryphon, C., Keegan, J., Kniker, C., and Krolak, P. "The Management and Security Expert (MASE)," Proceedings of the Second CLIPS Conference, Houston, TX., September 1991.

16. Inglehart, J., and Nelson, P. "LinkFinder: An Expert System That Constructs Phylogenic Trees," Proceedings of the Second CLIPS Conference, Houston, TX., September 1991.

17. Salzman, G., Duque, R., Braylan, R., and Stewart, C. "A CLIPS Expert System for Clinical Flow Cytometry Data Analysis," Proceedings of the First CLIPS Conference, Houston, TX., August 1990.

18. Heymans, B., Onema, J., and Kuti, J. "Application of a Rule-Based Knowledge System Using CLIPS for the Taxonomy of Selected *Opuntia* Species," Proceedings of the Second CLIPS Conference, Houston, TX., September 1991.

19. Engel, B., Jones, D., Rhykerd, R., Rhykerd, L., Rhykerd Jr., C., and Rhykerd, C. "A CLIPS Expert System for Maximizing Alfalfa (Medicago Sativa L.) Production," Proceedings of the First CLIPS Conference, Houston, TX., August 1990.

APPENDIX

CLIPS is free to NASA, USAF, and their contractors for use on NASA and USAF projects by calling the Software Technology Branch Help Desk between the hours of 9:00 AM to 4:00 PM (CST) Monday through Friday at (713) 280-2233. Government contractors should have their contract monitor call the Software Technology Branch Help desk to obtain CLIPS. Others may obtain CLIPS through the Computer Software Management and Information Center (COSMIC), which is the distribution point for NASA software. The program number is COS-10022. The program price is $350.00, and the documentation price is $140.00 (as of August 1991). The program price is for the source code. Price discounts are available to U.S. academic institutions. Further information can be obtained from

COSMIC
382 E. Broad St.
Athens, GA 30602
(404) 542-3265

An electronic bulletin board containing information regarding CLIPS can be reached 24 hours a day at (713) 280-3896 or (713) 280-3892. Communications information is 300, 1200, or 2400 baud, no parity, 8 data bits, and 1 stop bit.