N91-20699

ADVANCED CLIPS CAPABILITIES

Mr. Gary Riley
Mr. Brian L. Donnell
Software Technology Branch
NASA Johnson Space Center
Mail Stop PT4
Houston, TX 77058

ABSTRACT

The 'C' Language Integrated Production System (CLIPS) is a forward chaining rule based language developed by NASA at the Johnson Space Center. CLIPS was designed specifically to provide high portability, low cost, and easy integration with external systems. The current release of CLIPS, version 4.3, is being used by over 2,500 users throughout the public and private community. The primary addition to the next release of CLIPS, version 5.0, will be the CLIPS Object-Oriented Language (COOL). The major capabilities of COOL are: class definitions with multiple inheritance and no restrictions on the number, types, or cardinality of slots; message passing which allows procedural code bundled with an object to be executed; and query functions which allow groups of instances to be examined and manipulated. In addition to COOL, numerous other enhancements have been added to CLIPS including: generic functions (which allow different pieces of procedural code to be executed depending upon the types or classes of the arguments), integer and double precision data type support, multiple conflict resolution strategies, global variables, logical dependencies, type checking on facts, full ANSI compiler support, and incremental reset for rules.

INTRODUCTION

The 'C' Language Integrated Production System (CLIPS) is a forward chaining rule-based production system developed by the Software Technology Branch at NASA/Johnson Space Center [1,2,3]. Version 4.3 of CLIPS has capabilities similar to those of OPS5 (Official Production System) and is syntactically similar to ART (Automated Reasoning Tool) [4,5,6]. Version 5.0 of CLIPS introduces several enhancements to version 4.3 which will be discussed in this paper.

CLIPS OBJECT-ORIENTED LANGUAGE

The primary addition to version 5.0 of CLIPS is the CLIPS Object-Oriented Language (COOL). COOL is a hybrid system incorporating various ideas from other Object-Oriented Programming (OOP) systems such as Smalltalk and the Common Lisp Object System (CLOS) [7,8,9,10]. Since other constructs within CLIPS (defrule, deffacts, etc.) were not originally developed in an object-oriented manner, no attempt was made to rewrite CLIPS to develop a completely object-oriented system. Thus, OOP features that have been added to CLIPS are extensions rather than fundamental changes to the entirety of CLIPS. For example, no attempt is made to let all CLIPS constructs be treated as objects, such as rules being instances of the rule class [10]. Instead, an imaginary dividing line was drawn with the class construct; once instances of a user-defined class are created, they may only be handled in an object-oriented manner, i.e. via messages. However, other elements of CLIPS, such as rules and facts, are still manipulated in the same (non-OOP) manner as they were in previous versions.

The primary features of COOL are: classes with multiple inheritance, instances, message-passing constructs [10], and a query system for determining and/or iterating an action over a set of instances which satisfy user-defined criteria. The message-passing constructs consist of around, before, primary, and after message-handlers as well as slot-accessor message-handlers and slot-daemons.

Objects

An object in CLIPS is defined to be one of the following: an integer or floating-point number, a symbol, a string, a multifield value, or an instance of a user-defined class. Objects may be used anywhere within CLIPS: expressions, facts, rule patterns, and so on. Objects are manipulated by sending them mes-

sages. Instances of a user-defined class can only be manipulated with messages, but other objects can be handled in a non-OOP manner as well. For example, two integers can still be added by calling the '+' function directly; sending a message to one of the integers with the other as an argument (as one would in Smalltalk) is unnecessary [9].

Instances are created with a special function called *make-instance*, which allocates the memory for an instance and then sends it the *init* message. All operations on objects which are instances of user-defined classes are done with messages. The message-passing concept used by COOL is similar to that of Smalltalk [9].

Classes

A class is a special construct in CLIPS, similar to rules. CLIPS does not support metaclasses (classes of classes) [7,8,9,10], since classes are not objects. Classes must be manipulated with special functions like other CLIPS constructs. For example, to print a rule, the function pprule is used, and, similarly, ppclass is used to print a class.

Classes in COOL are defined much in the same way as they are in CLOS. Full multiple inheritance is supported using the rules found in CLOS [7,8]. Classes can have any number of slots, and slots can have a list of facets selected from a predefined set. Some of the slot facets available are: single and multi-valued cardinality, static and dynamic default values, shared and local storage (similar to class and instance variables respectively in Smalltalk), and access restrictions.

Messages

Messages are implemented by pieces of procedural code written in CLIPS called message-handlers. These handlers are bundled with the class definitions, and thus inheritance relationships may be used to determine to which messages an instance can respond. The implementation of a message can be further subdivided into handler types: around, before, primary, and after. This notion is borrowed from generic function methods in CLOS [7,8]. Around handlers are meant to set up an environment within which the other handlers may execute. Before and after handlers perform auxiliary work outside the scope of the primary handler. The primary handler is intended to do the core of the work of the message. Within the body of a message-handler is the only place where the slots of an object can be directly accessed without using messages. However, an object may send other messages (including ones to itself) in the course of executing a message. The declarative flow of execution for COOL message-handlers is similar to the standard method combination type in CLOS [7,8]. COOL also provides imperative control by allowing handlers to explicitly call other handlers that they are shadowing.

Slot-daemons

For every slot in an instance, two implicit primary message-handlers are defined: one for reading the slot and one for the writing the slot. Users must use these messages to access explicitly the object's slots. Slot-daemons may easily be defined by defining around, before, or after message-handlers which correspond to these messages.

Instance-Set Queries and Distributed Actions

At present, only facts can be pattern-matched on the left-hand side of rules; pattern-matching against the state of an instance of a user-defined class is not possible. However, COOL does provide a useful query-system for determining and performing actions on sets of instances that meet certain user-defined criteria. This query-system can be used with control facts to accomplish a brute-force instance pattern-match. (Control facts and slot-daemons may also be used to this end.)

An instance-set is an ordered collection of instances. Each member of this set is an instance of a set of classes defined by the user. The set of classes can be different for each instance in the instance-set. For example, one instance-set definition might be the ordered pairs of men or boys and women or girls. If there is one instance of each of these four classes, then there would be four instance-sets which satisfy the definition: (Man-1,Woman-1), (Man-1,girl-1), (boy-1,Woman-1), and (boy-1,girl-1).

A query is a user-defined boolean expression applied to an instance-set to determine if the instance-set meets further user-defined restrictions. Continuing the example above, one query might be that the two instances in an ordered pair have the same age.

A distributed action is a user-defined expression evaluated for each instance-set which satisfies a query. Continuing the example above, one distributed action might be to simply print out the ordered pair to the screen.

Several different functions are provided in this system: determine if there are any instance-sets which satisfy the query, group and return all

instance-sets which satisfy the query, perform an action for all instance-sets which satisfy the query, and others.

GENERIC FUNCTIONS

In addition to the object system itself, CLIPS 5.0 also supports generic functions [7,8,10]. Generic functions are groups of procedural code written in CLIPS that can later be called like any other CLIPS function. Different methods can be defined for generic functions that do different things depending on what the classes of the generic function arguments are. This allows new generic functions as well as standard CLIPS system functions to be overloaded. Although generic functions are not part of COOL (and can be used independently of it), they will utilize the full inheritance information of the classes of their arguments.

Generic functions in COOL are quite similar to generic functions in CLOS [7,8]. One difference is that COOL supports only primary methods, whereas CLOS has around, before, and after methods. This notion of splitting tasks into around, before, primary, and after parts was moved to messages and taken away from generic functions in COOL because it was felt intuitively that, for a particular set of arguments, a generic function should only execute one piece of code. However, it seemed reasonable that the implementation of a message might in fact be comprised of many different pieces of code.

CLIPS system functions which are not overloaded by generic functions completely bypass the generic dispatch mechanism. Thus, previous CLIPS programs will not pay any performance penalties simply as a result of the generic dispatch being available.

The argument restrictions which are used to determine the applicability of a method to a particular generic function call are somewhat more powerful than what is found in CLOS [7,8]. The user can specify that a restriction be any one of a list of classes, whereas CLOS only lets the user specify one class. Also, in COOL, the user may also specify an arbitrary boolean expression that the argument must satisfy for the method to be applicable. This is more powerful than CLOS individual methods, for they only allow the user to restrict the specific object rather than allowing any boolean expression. As a result of these enhancements, the precedence determination between methods in COOL is slightly more complicated than it is in CLOS.

To define new non-overloaded functions in CLIPS, the deffunction construct can be used in place of generic functions. Deffunctions allow a piece of procedural code to be written and used in the CLIPS language without any coding in an external language such as C. In previous versions of CLIPS, to add a new function, the user had to write it in C (or another language such as FORTRAN or Ada) and compile it, then relink CLIPS with the new function.

GLOBAL VARIABLES

The defglobal construct allows global variables to be defined which may then be accessed or set by rules, generic functions, and other constructs. Global variables allow information to be stored outside of facts (thus avoiding potentially unwanted pattern matching). For example, a global variable could be used to count the number of facts of a particular type. Incrementing the global variable from a counting rule would not reactivate the counting rule, whereas incrementing a value from a fact storing the count would retrigger the rule since the count fact would have to be matched against in the antecedent of the rule.

INTEGER DATA TYPE SUPPORT

CLIPS now supports an integer data type (represented internally as a C long integer). Floating-point numbers are now represented internally as C double precision numbers for greater accuracy. Previously, CLIPS stored all numbers as single precision floating-point numbers. Arithmetic functions such as addition, subtraction, multiplication, and division support mixed mode operations on integers and floats.

CONFLICT RESOLUTION STRATEGIES

Past versions of CLIPS supported a single conflict resolution strategy [4,11]. The order of rules to be executed on the agenda (the list of rules that have their conditions satisfied) was determined by the salience of the rule (a numerical value between -10,000 and 10,000) and the order of activation of the rule. Rules with higher salience are executed before rules with lower salience. Among rules of equal salience, the rule last activated is executed first (a "depth-first" or "stack" strategy).

CLIPS 5.0 now supports seven different conflict resolution strategies: depth, breadth, LEX, MEA, simplicity, complexity, and random. These resolution strategies are used to determine placement of an activation of a rule on the agenda between rules of equal salience. The depth strategy implements the "stack" placement (last-activated, first-executed) of activations found in previous versions of CLIPS. The breadth strategy implements a "queue" placement

(first-activated, first-executed) of activations. The LEX and MEA strategies are similar to the OPS5 strategies of the same name [4,5]. The simplicity strategy executes activations of rules with simple antecedents before rules with complex antecedents. The complexity strategy works in a directly opposite manner to simplicity strategy. The complexity of the antecedent of a rule is determined by manner factors including the number of patterns, the number of constant comparisons, and the number of variable comparisons. The random strategy randomly determines the order of activations of equal salience. The conflict resolution strategy can be dynamically changed and the agenda will be updated to reflect the new strategy.

SALIENCE EXTENSIONS

The salience declaration within a rule is no longer limited to strictly integer constants. The declared salience for a rule can be an expression which references global variables as well as calling system and/or user defined functions. In addition, evaluation of salience values can now occur at several different times: when a rule is defined, when an activation is placed on the agenda, and every cycle of execution. The user also has the ability to refresh the salience values of activations on the agenda at any time.

DEFTEMPLATE FIELD CHECKING

The deftemplate construct introduce in version 4.3 of CLIPS provided a method for structuring facts by tagging each field of the fact with a name. This provided CLIPS with a record structure similar to procedural programming languages. Optional fields in the deftemplate construct allowed type, value, and range restrictions to be specified. However, only the CLIPS Cross Reference, Style and Verification (CRSV) utility tool was able to make use of this information. CLIPS 5.0 now supports type, range, and value checking for deftemplates both statically (when patterns or actions using deftemplates are loaded) and dynamically (when deftemplate facts are asserted).

LOGICAL DEPENDENCIES

Truth maintenance [4,12] is now supported in CLIPS through the use of logical dependencies. The "logical" pattern operator can be placed around the first N patterns of a rule to indicate that facts asserted by this rule are dependent upon the existence of the facts matching the logical patterns (or non-existence of facts matching negated logical patterns). A fact asserted from a rule with logical patterns is logically supported by that rule. A fact may have multiple logical support from the same or different rules. A fact asserted from a source other than a rule with logical patterns is unconditionally supported (and cannot be retracted as a result of truth maintenance). Whenever a fact is retracted that matched a logical pattern of a rule (or a fact is asserted that matched a negated logical pattern), the logical support from that rule for any fact asserted by that rule is removed. Any fact that loses all of its logical support is automatically retracted.

INCREMENTAL RESET

In previous versions of CLIPS, newly defined rules were not activated on currently existing facts. That is, rules were only activated based on facts that were added after the rule was defined. Thus it was not possible to load new rules into the system and have these rules activated on previously asserted facts without somehow reasserting those facts. Newly defined rules in CLIPS 5.0, however, are fully activated by currently existing facts. This makes it possible to dynamically load new rules and have them automatically updated based on the current set of facts.

The build function allows construct to be dynamically created during execution. This makes it possible for a rule to create new rules (which would be incrementally reset based on the currently existing facts). It is also possible to safely delete rules as one of the actions of the consequent of a rule. It is even allowed to delete the currently executing rule and have the actions of its consequent execute to completion.

ANSI COMPILER SUPPORT

In addition to the numerous features added to CLIPS 5.0, the source code has been modified to be ANSI C conformant wherever discrepancies occurred between "K&R C standards" and ANSI C standards. Function prototypes have also been used for all functions to increase the maintainability of the code. ANSI C features not compatible with K&R C compilers can be removed through the use of compiler directive flags.

CONCLUSION

Version 5.0 of CLIPS provides several new capabilities which significantly increase the usefulness of the tool. Among these capabilities are: object-oriented programming, generic functions, global variables, integer data type support, additional conflict resolution strategies, salience extensions, type, range and value checking for deftemplates, incremental reset, and logical dependencies.

REFERENCES

1. Culbert, C., *CLIPS Reference Manual.* NASA document JSC-22948, July 1989.

2. Riley, G., *CLIPS Architecture Manual.* NASA document JSC-23047, May 1989.

3. Giarratano, J., *CLIPS User's Guide.* NASA document, August 1989.

4. Brownston, L., Farrell, R., Kant, E. and Martin, N., *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming,* Addison-Wesley, 1985.

5. Forgy, C., *OPS5 User's Manual.* Department of Computer Science document CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, PA. 1981.

6. *ART Reference Manual,* Inference Corporation, Los Angeles, CA. 1986.

7. Keene, S., *Object-Oriented Programming in Common Lisp.* Symbolics, Inc., 1989

8. Bobrow, D., DeMichiel, L., Gabriel, R. , Keene, S., Kiczales, G. and Moon, D., *Common Lisp Object System Specification.* X3J13 document 88-002R, June 1988.

9. Pinson, L. and Wiener, R., An Introduction to Object-Oriented Programming and Smalltalk. Addison-Wesley, 1988.

10. Tello, E., Object-Oriented Programming for Artificial Intelligence. Addision-Wesley, 1989.

11. Cohen, P. and Feigenbaum, E. (ed.), The Handbook of Artificial Intelligence, Vol. I, William Kaufmann, Inc., 1982.

12. Cohen, P. and Feigenbaum, E. (ed.), The Handbook of Artificial Intelligence, Vol. II, William Kaufmann, Inc., 1982.