Integrating an Object System into CLIPS: Language Design and Implementation Issues

Mark Auburn

Inference Corporation
5300 W. Century Blvd.
Los Angeles, CA 90045

Abstract

This paper describes the reasons why an object system with integrated pattern-matching and object-oriented programming facilities is desirable for CLIPS, and how it is possible to integrate such a system into CLIPS while maintaining the run-time performance and the low memory usage for which CLIPS is known. The requirements for an object system in CLIPS that includes object-oriented programming and integrated pattern-matching are discussed, and various techniques for optimizing the object system and its integration with the pattern-matcher are presented.

1. Introduction

As CLIPS, and CLIPS-like production systems, gain widespread usage and acceptance, and as the number of CLIPS applications increases, the limitations of the main CLIPS data representation, the fact, become more evident. Although facts, and the n-ary relations they represent, are a powerful and flexible method for representing arbitrary relationships between data, the lack of explicit relationships between individual facts and their lack of internal structure inhibit the representation of large, complex knowledge bases.

Object representations, such as embodied in the object-oriented programming languages of Smalltalk, CLOS and C++, and in the experimental languages KL-ONE, are a natural data extension to CLIPS's facts. Object-oriented programming languages that include the capability of pattern-matching on objects represent a combination of two separate lines of research: research on representing objects and representing the actions associated with those objects, and research on the most efficient general methods of matching on data. It is apparent that both of these lines have matured, in the form of efficient commercial object-oriented programming languages (e.g. Classic-Ada [8]) and efficient commercial production systems.

In the first section, the specific advantages of an object system will be discussed, followed by a presentation of what requirements are necessary for an object system that would maximally increase the utility of CLIPS programming and the various tools built around the basic production system component of CLIPS.

These issues will be illustrated using the example of ART-IM (Automated Reasoning Tool for Information Management) [5], a tool from Inference Corporation for development of expert systems, which shares a common syntax and many implementation strategies with CLIPS, and may be logically viewed as an extension of CLIPS.

In the second section, issues of object system implementation are examined, concentrating on the integration into CLIPS's pattern and join networks necessary to achieve the desired efficiency of pattern-matching. Although it is possible to match against an object's slots and values just as is done for facts, the nature of an object system allows for an additional degree of optimization based on knowledge of the object hierarchy and assumptions about the rate of change of various parts of the hierarchy. Just as assumptions about the frequency of working-memory change lead the implementation of a fact pattern-matcher to use the Rete algorithm, assumptions about the usage of the object system lead to additional optimization techniques. This paper discusses those

assumptions and several of the techniques used by the ART-IM object system to reduce object system overhead.

Finally, some future directions for object system enhancement are sketched.

2. Language Design

2.1. Advantages of an Object System

Although fact-based data storage and retrieval, including fact-based pattern matching, provides a wide range of desirable functionality for the developer of expert systems, there remain many expert system applications whose data representation cannot be adequately represented in facts. The working-memory model, made popular by OPS5 [1] and implemented as facts in CLIPS, implicitly subdivides and flattens data down to a level comparable to a database record or a record in a conventional programming language. However, there are many problems such as classification and diagnosis for which an inheritance hierarchy is both closer to a natural understanding of the domain and more economical in expressing data. Although an inheritance hierarchy does not expand the class of possible applications beyond that of the working-memory model, in many cases it can provide a more natural, economical and maintainable representation. An object system offers the following advantages over a working-memory model:

- An explicit hierarchy.
- Explicit inheritance (along with the ability to override it).
- Explicit internal structure that can be declaratively described.
- Easier to maintain, since it corresponds better to the user's model.

There are, of course, disadvantages. Typically object systems, in exchange for these advantages, require more memory and more processing time than an equivalent fact representation. However, due to the decreased maintenance cost of a more explicit representation, the total software lifetime cost may be lower.

Once an object representation is in place, it is also possible to enhance the inheritance hierarchy with procedural attributes to achieve object-oriented programming. Although rules can be used to duplicate any procedural activity, it is often simpler, in cases where the control flow is predefined, to write procedural code. Procedural code will typically be faster than an equivalent rule version, since the overhead for control flow determination implicit in a rule implementation lacks. Object-oriented programming can be used to achieve some of the same goals of rule-based programming, in that by increasing the locality between data and the operations on that data the ease of maintenance is increased.

An object data representation also offers a finer granularity of update recalculation over the working-memory model in that a data change can be performed, and pattern-matching updated, on a change to an object's slot value, rather than only on the assertion or retraction of an entire fact. In large applications this can have a significant impact on performance.

2.2. Requirements for an Object System in CLIPS

The utility of an object system for CLIPS depends directly on the degree of integration with CLIPS, and its subsidiary features, achieved by the object system. The main requirement, of course, is that it integrates with the pattern-matcher. Object patterns must be provided that offer the same sophisticated pattern-matching available to fact patterns.

The object patterns need to be able to:

- Test for the existence of an object.
- Test the class membership of an object.
- Test for the existence of a specific attribute on an object.
- Test for the values of a specific attribute on an object.

Binding variables to various attributes and values, and comparing those variables to other attributes and values in the same object, and to other variables

bound in other object and fact patterns, is also an important consideration.

The object system needs to be completely dynamic, as with facts, and to enjoy a full procedural interface for changes during execution. Object-oriented programming, while perhaps not a necessity given the availability of the powerful rules of CLIPS, is certainly desirable. Essential to the programming ease of the object system is full integration into all debugging features and into all programming utilities, such as those for verification and validation, truth maintenance and explanation generators.

ART-IM, as an example CLIPS extension, provides an integrated object system with inheritance and three types of links: subclass, class member and user-defined relations. The attributes of the objects are defined using the object system itself, and they and their values are inherited by children nodes. Object-oriented programming is also provided and consists of attaching methods to atttributes of the appropriate object. The ART-IM object system is also integrated with ART-IM's explanation-generation subsystem and with its justification-based truth-maintenance system.

3. Implementation

Although the features provided by an object system are desirable, it is clear that in a production system designed for speed and low memory usage like CLIPS an inefficient implementation of the object system features would severely restrict the usage of the object system. In particular, without the deep integration between the object hierarchy and the pattern-matcher, such as exists between the fact database and the pattern-matcher, the efficiency of rules that matched on objects would be much less than that of those rules that matched on facts, and therefore of little use in a real-world CLIPS application.

ART-IM incorporates a variety of implementation techniques to increase the efficiency of the object system, and some of these techniques are discussed below. It is possible, in some cases, for the efficiency of matching on objects to exceed the efficiency of matching on equivalent facts, using these implementation techniques.

In particular, three techniques for optimization are discussed below:

- Representing class membership with the use of bit vectors.
- Canonicalizing attribute order.
- Precomputing valid object patterns for a particular segment of the object hierarchy.

The second technique, although useful for reducing the storage requirements of a large and multilayered object base, is crucial to ensuring the success of the third and is primarily useful in that context.

This paper will not touch on the various techniques for optimizing method selection on objects in object-oriented programming. In general, since pattern-matching is the most important constraint in most CLIPS applications and in most production systems, the integration with pattern-matching is viewed as the most important efficiency topic.

3.1. Representing Inheritance Information

Since the test for class membership is performed often in an object system (and replaces the fact equivalent of testing for a particular value in a particular position on a fact), optimizing this test would appear to yield significant benefits.

There are at least two commonly used methods for deciding which classes an object belongs to:

- Explicitly passing class information down from each class to all of its children.
- Requiring the system to search upward from an object to its immediate parents, repeating the search until all of the parent classes have been discovered.

The processing time for such class membership determination is conserved in the first, while storage space is conserved in the second. Due to multiple inheritance and deep inheritance hierarchies, the first method can become prohibitively expensive in terms of space when implemented by representing class membership by attribute values. On the other hand, searching upward from an object to all of its classes can consume large amounts of processing time, especially if the results of the search are not cached

for future use.

A technique used in ART-IM to reduce the space consumption of the first method while preserving its fast class comparison test is that of encoding inheritance chains into bit vectors. Encoding the class structure of each object into a binary vector has two desirable properties: it consumes little space (in ART-IM, one byte per ancestor link), and the test of whether or not an object belongs to a specific class is reduced to the quick test of whether or not a binary value is contained as a prefix in the vector of the object.

Of course, the encoding of inheritance values costs processing time, but the cost of the processing is on the same order as that of directly passing class information as attribute values down to the object's children, and the space consumption is approximately an order of magnitude less. The membership test itself is again only slightly more complex than the search for a particular attribute value.

3.2. Canonicalization of Attribute Combinations

A typical implementation for a fully dynamic object system (one that allows the creation and destruction of all classes, subclasses and class members, along with the creation and destruction of object attributes, during execution) of the attributes of objects is as a linked list. As attributes are added to an object, or deleted, they are inserted into or removed from the object's attribute list. In order to add or substract values from an attribute, it is necessary to search the list looking for the attribute, and then insert the value into the value list of that particular attribute.

The advantages of this representation are:

- The implementation is straightforward.
- Dynamic addition and deletion of attributes is a simple list operation.

The disadvantages are:

- Inserting or deleting a value requires a full search of the attribute linked list.
- Each attribute requires at least two words of memory, no matter how static the inheritance hierarchy is.

The linked list representation is certainly the most efficient implementation when attributes are dynamically added and deleted to objects with a high frequency. However, as the frequency of attribute changes decreases, the most efficient representation converges on an implementation which is the analog of a structure (or record) in a conventional programming language: a contiguous segment of memory with implicit positioning of attributes.

In order to allocate contiguous segments of memory (erasing the need for the link field and the attribute name per attribute), and still allow for dynamic changes, it is necessary to create a parallel data structure which represents the attribute combinations present in the object system. By creating a canonical ordering for all attributes in the system, the space consumed by this parallel structure can be reduced.

As objects are created, their attributes are sorted into canonical ordering. The attributes are then stored in an array that does not include either a link field or the name of the attribute itself. In order to determine which element of the array belongs to which attribute, a pointer is attached to the object which points at a parallel attribute-combination hierarchy. Each node in this hierarchy contains a specific combination of attributes, and the growth of the hierarchy is dependent on the canonical order of the attributes contained in each node. This hierarchy is more efficient than representing the attributes directly in the objects because many objects will share specific attribute combinations, but requires some additional time for attribute lookup. However, the time for attribute lookup can also be less than the list implementation, depending on the hardware, as an array lookup is often implemented in hardware, whereas a list lookup is not.

This canonical ordering of slots is also an essential prerequisite to the pattern precompilation technique discussed in the following sections, which further reduces the cost of matching the attributes of an object to the attributes required by a particular pattern.

3.3. Pattern Matching Technology for Record Data Types

Production systems, the software tools that have refined the technology of pattern-matching the farthest, have traditionally used either simple variables or records as their data representation. Data types called "working memory elements", which are similar to the records of data bases or traditional programming languages, have been used most frequently in systems such as OPS5. Efficient algorithms for pattern-matching on these working memory elements have been developed, including Rete [2] and TREAT [6]. Variants on these algorithms, in particular for parallel machines [3] [4], have been designed, and comparisons have been performed [7]. These algorithms, however, have typically only been tested and designed for the working-memory model.

These algorithms make several assumptions:

- That the set of patterns to match on is constant.
- That the knowledge base (the collection of working memory elements) is large.
- That the change in the knowledge base over the interval of time between each match is small.

The goal of these algorithms is to reduce the time required for deriving the matches by storing partial results for the matches, and updating the partial results as the knowledge base changes. Otherwise, the N times M comparison necessary for full derivation of the matches of a set of patterns, where N is the number of knowledge base items and M is the number of patterns, is far too computationally expensive to obtain whenever the matches are desired.

In a pattern that consists of references to several working memory elements, for example, the Rete algorithm will store two types of data for all matches: pointers to all working memory elements that match an individual reference in the pattern (a condition), and partial matches for successive subsets of the conditions in the entire pattern. As changes in the knowledge base occur, they are percolated down to a network created by the Rete algorithm which determines how to update the stored partial results based on the changes. Since the time required for obtaining the matches is dependent only on the number of changes in the knowledge base since the last pattern-matching point and the number of patterns which are affected by those changes, and not on the total number of patterns or knowledge base objects, it typically reduces the pattern-matching time by a significant factor.

As the form of data representation has migrated from records, in the form of working memory elements, to objects as the representation of choice, due to their economy of representation (from inheritance) and flexibility, the Rete and TREAT algorithms were adapted in a straightforward manner to match on objects. Objects and their attributes and values were transformed into object-attribute-value triplets, and these triplets handled exactly like simple working memory elements. As objects changed, modified triplets were sent to the pattern-matcher for updates. Although this method for object integration is straightforward and allows for the reuse of code developed for fact pattern-matching, it does not exploit the wide range of optimization possibilities inherently present in an object system. The following two sections discuss some of the features available for optimization in the object system, and one technique for exploiting some of these features.

However, since comparing bound variables across various objects allows for the same implementation as the identical comparison in the fact pattern-matcher, that comparison will not be discussed in this paper. Object systems do not present additional problems or opportunities in the inter-condition comparison, as opposed to the intra-condition case.

3.4. Object System Features Relevant for Pattern Matching

As in the case of knowledge bases constructed using working memory elements, it is possible to construct a set of assumptions about object-based knowledge bases in addition to the assumptions stated above:

- That each object may have a large set of different attributes.
- That each pattern may refer to a limited group of attributes of an object.

- That the inheritance hierarchy changes slowly, if at all.

- That many objects will be instances of classes, as opposed to representations of subclasses.

Like all assumptions, these may be violated in any particular application, but should hold in general. Based on those assumptions, it would seem desirable to implement pattern-matching on an object system such that:

- Matching on an instance of a class is highly efficient, even if the set of instances and their values change relatively rapidly.

- Each pattern need only inspect those attributes of an object that are used in the match.

- Inheritance and class information is incorporated as much as possible, given that patterns may refer to that information and that it changes slowly.

These assumptions form the basis for the next section, which describes a particular method for utilizing these apparent features. However, it is important to note that there exist many different methods for exploiting these assumptions, just as with working-memory element pattern-matchers, and that the one described below is only one of several possibilities.

3.5. An Inheritance Hierarchy for Pattern Matching Correlations

Once the pattern and join networks (or alpha and beta nodes, to use the terminology of [2]) for a set of fact patterns have been created, the process of matching a new fact to the existing patterns is described by testing the fact against the entire set of application patterns, and producing matches for those patterns which the fact successfully matched against.

Using the features of the object system described in a previous section, it is possible to reduce the size of the set of patterns considered in the matching process. By using structural characteristics of the patterns (such as which classes they address or the attributes they contain), it is possible to substantially reduce the set of patterns considered, which depending on the cost of examining each pattern for applicability can reduce the processing time required for pattern-matching considerably.

Once the parallel attribute-combination hierarchy described in an earlier section has been created for an object system, each pattern is attached to exactly one node in that attribute-combination hierarchy. Each pattern is attached to that attribute-combination node which contains exactly those attributes used in the pattern. As objects are created, then, in addition to the cost of searching for the appropriate attribute-combination node, pattern-matching information is attached to the object, derived from the nodes in the attribute-combination hierarchy that the object traverses. The pattern-matching information will apply to that class and to its subclasses. Attaching pattern-matching information to the object hierarchy, and updating it as the hierarchy and the objects contained it change, does impose overhead on changes to the object system. Based on the assumptions above, the relative infrequency of changes to the object hierarchy will compensate for the expense of those changes.

When pattern-matching occurs, preselection of those objects that are relevant to a pattern has already been accomplished, so that patterns that couldn't fulfill a particular object (e.g., they belong to a different class or do not contain the attributes required by the pattern) are not considered in the pattern-matching process. For class instances, in particular, this can bring a substantial performance improvement, as they need only use the pattern-matching information of their class in deriving the appropriate patterns. The repetitive class membership tests and the attribute presence tests required in patterns can be performed once, for the class, and amortized over the entire set of class instances.

4. Conclusions

This paper has presented several reasons for integrating an object system into CLIPS, as well as some techniques for optimizing that integration. The optimization techniques, although implemented for a production system, are applicable to other object-based processing methodologies that use pattern-matching.

There are other ideas that have not been implemented but deserve active consideration.

It would be quite desirable to introduce the capability to partition the knowledge base, and indeed individual attributes on objects, into items appropriate for pattern-matching and items upon which pattern-matching will not be performed. Since pattern-matching imposes an overhead on objects and their attributes, reducing this overhead by confining it to specified areas could greatly improve efficiency. In addition, developing protocols for passing information between a pattern-matcher and an object system that are independent on the object used, or indeed on the implementation of the pattern-matcher, would be of interest. This would allow the creation of object-oriented data bases with integrated pattern-matching, with the advantage of efficient storage of large number of objects on disk.

Taking such a protocol and enhancing it for distributed communications would present the interesting possibility of distributed expert systems communicating through a general object metaprotocol, as well as allowing for a flexible, transparent external data interface that would communicate with data from such diverse sources as databases, windowing interfaces and process monitors.

Allowing type and value restrictions on object attribute values, and being able to specify an internal structure for those values, is also a desirable addition.

References

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

2. Forgy, C.L. "RETE: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem". *Artificial Intelligence 19* (1982).

3. Gupta, A.. *Parallelism in Production Systems*. Pitman Publishing, 1987.

4. Gupta A. et. al. Results of Parallel Implementation of OPS5 on the Encore Multiprocessor. CMU-CS-87-146, Carnegie-Mellon University, Department of Computer Science, August, 1987.

5. Inference Corporation. *ART-IM/MS-DOS 1.5 Reference Manual*. Inference Corporation, 1988.

6. Miranker, D.P. TREAT: A New and Efficient Algorithm for AI Production Systems. Phd thesis, Columbia University, 1987.

7. Schor, M.I., Daly, T.P., Lee, H.S., Tibbitts, B.R. Advances in Rete Pattern Matching. Proceedings of the National Conference on Artificial Intelligence, AAAI, 1986.

8. Software Productivity Solutions, Inc. *Classic-Ada User Manual*. Software Productivity Solutions, Inc, 1988.