Expert systems; software engineering environment

Artificial intelligence began on the far-out fringe of computer science and has been progressing more and more into the mainstream of software engineering through the use of AI components, such as inference engines, that can be embedded in programs written in more traditional languages. Eclipse takes this progression one step further by allowing the use of Lisp in combination with rules and traditional C programming. We are currently making use of Eclipse in our work on a team software development environment.
1 Introduction

Eclipse, the subject of this paper, is one component of an object-based software development environment (SDE), that is being developed at Hewlett-Packard Labs in Palo Alto. This SDE is designed to support teams of developers building large software systems. In addition to providing a shared object base of project-wide information, the environment provides rule-based support for managing the software development process. These rules create an active database that responds to user actions, checks constraints and automates repetitive tasks. A typical constraint rule might require that a software module pass certain tests before being included in a release.

For reasons of efficiency, the rules are attached not to the persistent, project-wide object base but rather to a local cache of objects in the virtual memory of each developer's workstation. This active cache of objects is stored in Eclipse, which is a fusion of XLISP [Bet89] and CLIPS [CLI88]. XLISP is a small, fast and powerful Lisp interpreter written in C, with easy linkage to external routines written in C. CLIPS (C Language Production System) is a forward-chaining inference system developed at NASA's Johnson Space Center. We have integrated XLISP and CLIPS and added some additional features as well.

Eclipse has been developed as a component of the SDE, but we think that it could be useful in a wide variety of knowledge-based applications. The rest of this paper describes Eclipse in more detail.

2 Inference Engine

2.1 RETE Algorithm

The information in a CLIPS database is stored in the form of facts (simple tuples). Like OPS5 [BFK85] and ART (Automatic Reasoning Tool) [Cla87], CLIPS uses the RETE algorithm [For82] to ensure that matches between rules and facts are performed incrementally. Following this algorithm, the system saves partial matches between facts and rule patterns in a network (the pattern net) and recomputes them incrementally, as needed. It also saves partial matches between facts and combinations of patterns (in the join net). For greater efficiency, patterns and joins common to more than one rule (and the match information associated with them) are shared, not duplicated. The RETE algorithm supports forward chaining (event-driven actions), although some RETE-based systems use forward chaining to simulate backward chaining (goal-directed queries). CLIPS does not support backward chaining, but that feature is a possible future enhancement of Eclipse.

The RETE approach is similar in spirit to the incremental computation model of spreadsheets, in which information is saved, links are maintained and the results of changes are propagated automatically. Some language-based editors also adopt the incremental computation paradigm [RT87]. These editors, taking advantage of the grammatical structure of the language they are supporting, attach automatically updatable attributes to the nodes of the language's abstract syntax tree (the attribute grammar model [Knu68]). Recent research has sought to extend the features of attribute grammars to general graph structures, but it is not clear how well the update algorithms applicable to attribute grammars can be extended to general graphs. The RETE algorithm supports an arbitrary dependency graph in a way that tries to balance efficiency and generality.
CLIPS (version 4.2 from NASA) supports RETE-based forward chaining, but it does not provide the features listed below. We have enhanced the version of CLIPS contained in Eclipse to support these additional features.

2.2 Logical Dependencies

Logical dependencies provide a simple form of nonmonotonic reasoning (or truth maintenance), in which the arrival of new knowledge may cause existing knowledge to be retracted [Hal87]. For example, the assertion that Clyde can fly depends on the assumptions that Clyde is a bird and that all birds can fly. If we discover that Clyde is not a bird or that not all birds can fly, the assertion that Clyde can fly no longer holds. If the inference engine supports logical dependencies, as Eclipse does, the dependent fact (Clyde can fly) will be retracted automatically. Similarly, an assertion may be logically dependent on a negated condition: if Clyde is not female, the he's male. If we discover that Clyde is female, then the fact that Clyde is male will be retracted automatically.

2.3 Incremental Rule Compilation

In the RETE algorithm, rule compilation is incremental in the sense that existing matches (and sets of matches) are saved and new matches are computed as facts are added to or removed from the knowledge base. However, one cannot add new rules to the system without recompiling the entire network. We have revised the model to allow users to add rules incrementally. The system will compute new matches and match sets as needed while doing as little recomputation of existing matches as possible.

2.4 Multiple Agendas

CLIPS allows users to attach a numerical salience (priority) to rules. Rules with higher priorities are placed ahead of ones with lower priorities on the agenda. (The agenda is the queue of rules whose conditions have been satisfied and which are ready to be executed.) In addition to supporting rule priorities, Eclipse permits users to separate rules into disjoint rule sets. Rules from different rule sets are placed on different agendas. (Under the current conflict resolution strategy, the highest priority rule on each agenda is executed in turn, in round-robin fashion.)

3 Interpreted Lisp Environment

3.1 Tests and Actions in Rules

As indicated above, CLIPS is written in C. This makes it smaller and more portable than Lisp-based inference engines. However, Lisp-based systems, besides being easier to develop, generally allow users to define new functions and use them in rules (for left-hand-side test or right-hand-side actions) without recompiling the system. CLIPS allows user-defined functions, written in C, to be used in rules, but arguments must be handled in a special way and the functions must be compiled and linked into the CLIPS executable.

To provide the benefits of an interpreted environment, we have integrated CLIPS into
a small Lisp interpreter called XLISP. XLISP was developed by David Betz and is now available in the public domain. The Lisp primitive functions are written in C and compiled, and new primitives (as well as interpreted functions written in Lisp) can be added.

The merger of XLISP and CLIPS makes it much easier for users to define functions for use in rules. There is no special argument-passing mechanism, and the system does not have to be recompiled simply to define a new action on the right-hand side of a rule.

The simple control structure of rule-based programs promotes incremental development since individual rules can generally be added or deleted without a major restructuring of the program. The RETE algorithm, supplemented by incremental rule compilation, uses incremental computation as an implementation model. In combination with a Lisp interpreter, development and computation is even more incremental.

### 3.2 Data Representation

The original CLIPS from NASA uses a single hash table to store the symbols and strings used in facts and rule patterns. XLISP has its own separate symbol and string tables. To save space, and to save time translating between two different representations, Eclipse uses the XLISP representation for strings and symbols throughout, both in Lisp code and in facts and rules.

Although Eclipse use tuples as a simple and flexible fact representation, Eclipse also provides some support for complex, structured objects (record-like structures with named fields/attributes/slots whose values may be other objects; this is the form a data visible to the user in the SDE). There are Eclipse commands for adding, removing and retrieving the slot values of objects. Objects in Eclipse are represented in two different ways simultaneously. First of all, they are represented as sets of facts, one fact corresponding to each object-slot-value triple. The fact representation meets the pattern-matching needs of the inference engine. But objects are also represented by Lisp symbols: the property lists of those symbols contain the slot values of the objects. The property list representation permits more efficient access to slot values in contexts that require direct modification rather than pattern matching.

Of course a simple property list, with slot names as keys, is not as efficient a way to store associatively indexed data (slot values) as a vector for which the keys have been compiled into integer offsets. Moreover, a property list provides no built-in support for typing and inheritance. But a property list is flexible. Objects can have attributes indicating the class(es) they belong to, and classes (which are also objects) can have attributes indicating their superclass(es). Inheritance can be implemented by rules. The system can easily accommodate changes in class structure, in the set of slots associated with a class, or in the inheritance scheme.

---

1. The SDE provides higher-level versions of these commands as well as commands for defining/creating classes, slots and objects.
4 Implementation Notes

4.1 Logical Dependencies

In Eclipse, declaring a rule *logical* makes any facts asserted by the actions on the rule's right-hand-side logically dependent on the set of facts matching the combination of left-hand-side patterns. Fact B will automatically be retracted if a fact it depends on, call it A, is retracted. In contrast to ART, which allows you to restrict the logical declaration to one or more consecutive patterns at the beginning of a rule's left-hand side, Eclipse makes the entire rule "logical".

Logical dependency information is stored in a rule's *terminator* join. A join node normally connects a given rule pattern to the previous patterns (already joined) of the rule. The terminator join connects the last ordinary join (which connect previous patterns to the rule's last pattern) to the actions on the right-hand side of the rule. A join node's *beta memory* contains the sets of facts that match the previous combination of patterns (driven down from the previous join). But the terminator join's beta memory is normally empty, with the set of facts corresponding to a given activation of the rule stored on the activation itself (on an agenda). If the rule is declared logical, the set of facts corresponding to the activation is also copied to the terminator join's beta memory.

If a new fact is asserted when the rule is fired, that fact is linked to the new fact in the terminator join, and the new fact's support count is set to 1. (If the fact was already asserted, its support count is incremented.) The retraction of any fact causes match information to be updated throughout the pattern and join nets. If one of the facts in the logical match set listed at a terminator join is retracted, the dependent facts' support counts are decremented. If a fact's support count reaches zero the fact is retracted. Similarly, if a new assertion causes a negated condition to become true, the token corresponding to the negation will be removed from the logical match set and the dependent facts' support counts will be decremented in the same way.

4.2 Incremental Rule Compilation

When new facts are asserted, they are sorted in the pattern net (matched against rule patterns), and the match information is propagated through the join net. The conventional RETE algorithm, as in the original CLIPS, requires that all rules be defined (and the RETE net constructed) before any facts are asserted. Incremental rule compilation permits rules to be added *after* facts. Adding new rules may cause new pattern and/or join nodes to be added to the network, although new rules may share old patterns and/or joins with existing rules. The incremental match algorithm should determine (and propagate) matches for new patterns and joins while avoiding recomputation of old matches.

A pattern node's *alpha memory* contains the facts that match the pattern. In the Eclipse model of incremental rule compilation, when a new join is created, the match sets for the previous join (the merger of facts in the previous pattern's alpha memory and the facts in the previous join's beta memory) are propagated down to the new join. (Note that matches are not sent down to sibling joins since their match information is already up to date.) If the previous pattern is negated and no facts match it (the alpha memory is empty), then a match for a negated pattern is propagated. (A special pseudo-fact token
representing the satisfied not pattern is added to the join's match set and driven to the new join node.)

In CLIPS (and Eclipse), an "entry" join with an empty beta memory is attached to the first pattern in a rule. If the new join is an entry join, there is no previous pattern and no previous join. In that case, there are no matches to drive from above. However, matches for the new entry join will be sent down to the join at the next level when that join is created. (The match set will not be empty if the new join is attached to an old, shared pattern or if the new pattern is negated.) In addition, matches for all new entry patterns will be computed and propagated after the join net is complete.

In fact, after the extension of the RETE net and the propagation of match information to the new joins, the existing facts are matched against all the new patterns in the pattern net$^2$. (Some new joins may be connected to old, shared pattern, but all new patterns are connected to new joins.) Facts that match the new patterns are then merged with the sets of facts in the beta memories of the corresponding (new) joins. The resulting match sets are propagated down to succeeding join nodes.

This entire process updates the match information stored in the RETE net and creates new rule activations as appropriate.

### 4.3 Lisp/CLIPS Interaction

Eclipse contains primitive Lisp functions for accessing the CLIPS inference engine. Primitives are provided to create and destroy both rules and facts, as well as to print information about the rules and facts and invoke the inference process. Invoking the forward-chaining engine will activate any rules pending on the agenda and execute the actions on the right-hand sides. The right-hand side action can be any body of Lisp code. When a rule is entered into Eclipse using the `defrule` primitive, a Lisp function is constructed containing the code provided on the right-hand side of the rule and taking as arguments the bindings for variables that will be made on the left-hand side. When the inference engine fires a rule it takes the variable bindings from the patterns matching the left-hand side and calls the function containing the right-hand side actions.

### 4.4 Memory Management

XLISP contains a garbage collector used to reclaim storage no longer needed by Lisp. Since in Eclipse the inference component (CLIPS) shares the data structures for Lisp symbols and strings, the Lisp garbage collector must be made aware of which structures are being used by CLIPS so that they are not prematurely discarded. For symbols this is easy, since once a symbol is entered into the symbol table it is never discarded. CLIPS maintains a list of the Lisp strings it is currently using, which the Lisp garbage collector looks at during the marking phase of garbage collection.

CLIPS manages its own storage for facts and for rules; Lisp is never allowed access to them. When a fact must be passed from CLIPS to Lisp it is first converted into a Lisp list built from cons cells.

---

$^2$Actually, all facts are re-sorted in the new, augmented pattern net and only new matches are recorded. Some redundant matching does take place.
5 Issues for Future Consideration

5.1 Efficiency of the Inference Engine

The pattern matching during the first phase of RETE and the join tests during the second phase are currently the most expensive parts of the inference engine in terms of execution time. When running the prototype of the SDE much more time is spent doing this than in actually running the Lisp code triggered by the rules.

One way to speed up matching in the pattern net may be to index patterns by the hash value of their first element (usually a symbol). The pattern matcher would sort facts by hashing their first elements and finding the patterns that match. Current the fact is compared to all of the patterns until of is found whose first element matches.

The key feature of the RETE algorithm is that it saves partial matches between facts and rules. A fairly radical technique to improve the efficiency of the join net would be to save matches for patterns (alpha memories) but discard match information at join nodes (beta memories). Under this method, known as the TREAT algorithm [Mir87], matches for joins would have to be computed and propagated every time a newly asserted fact matched a pattern. However, the system would not have to update all the join nodes when a fact was retracted. The developers of the algorithm claim that, for most applications, there would be a net decrease in the amount of work performed and in the amount of space required to store intermediate results.

5.2 Backward Chaining

An inefficiency in forward chaining arises when work is done to deduce facts that are never used, thus it is desirable to focus the work of the inference engine toward known targets. This is frequently possible by structuring the rules properly, but there are cases where it is not convenient (or even possible) to do so. One way to solve this problem is to use backward chaining and to trigger it only when the results are needed, allowing only required answers to be generated and only on demand. We are looking into ways that backward chaining can be integrated into the forward chaining Eclipse inference engine.

6 Conclusion

Eclipse is based on the RETE algorithm, written in C, coupled with a C-based Lisp interpreter, and equipped with useful features. Our goal in developing it is to create a small, efficient, but reasonably powerful inference engine. Although it is a component of the SDE, we think that it could be useful in a variety of knowledge-based applications.

7 References


