Behavioral Match: Embedding Production Systems and Objects

Jacques Bouaud[†] and Robert Voyer[‡]

[†]DIAM, Service d'Informatique Médicale AP-HP
& Dép. de Biomathématiques, CHU Paris 6
91, boulevard de l'Hôpital, F-75634 Paris CEDEX 13 - France

[‡]INT & LAFORIA, Université Paris 6
9, rue Charles Fourier, F-91011 Evry CEDEX - France

{*bouaud@biomath.jussieu.fr, voyer@int-evry.fr*}

Abstract

Attempts to merge production systems and object-oriented paradigms have not yet yielded well integrated systems. In this paper, we show that standard match algorithms, such as RETE, rely on an assumption of structural match that the object-oriented paradigm basically do not respect. Considering that a neat embedding of objects and production systems should preserve both encapsulation from the former and correctness from the latter, we address this issue through the problem of behavior-based match. In this framework, we derive two implemented solutions that enable the use of standard match algorithms in a Smalltalk environment.

1 Introduction

Several attempts to merge the production system and object-oriented paradigms have been performed: objects are used for representing data and actions, whereas rules for expressing pieces of knowledge about objects. Control is taken in charge by an inference engine. But, in the case of forward chaining "first-order" production systems, the resulting systems are not satisfactory in the sense that essential properties of these paradigms are lost. Either at the integration level a data hybrid model is proposed, and the encapsulation principle is not respected (*e.g.* LOOPS [1], ART [2], etc), or at the exploitation level implemented algorithms are inefficient and even wrong (*e.g.* Opus [3], NeOpus [4]). These works point ou an apparent inadequacy between production systems and objects.

In this paper, we analyse the foundations of production system match algorithms and show that the object-oriented paradigm basically does not satisfy one of the most basic assumption of optimized structural match algorithms such as RETE [5] which is a correct α-memory update. This last problem is revisited within the issue of embedding objects and production systems. A neat integration should preserve encapsulation from the first and correctness from the second. Such an integration is based on *behavioral match* where reference to objects is made according to behavior rather than to structure. In this context and with the most inefficient naïve match algorithm as a start point, we derive two successive levels of optimization so that standard match algorithms like RETE can be reused, gaining efficiency. These solutions satisfy the constraints of behavioral match and have been tested into Smalltalk.

The paper is organized as follows. Section 2 mentions existing embedding attempts and points out the inherent problems in the retained choices. Fundamental issues of match algorithms are revisited in the framework of an object-oriented environment in section 3. Two solutions to behavioral match are presented along section 4. The last section concludes.

2 Background

More than the nature of data referred to in rules, it is the way they are referred to that differentiates systems mixing objects and rules, that is, the language for expressing the *condition elements* (CEs). In the condition part of rules, CEs express the selection criteria to apply on the pieces of data, or *working memory elements* (WMEs). In a first category of systems, CEs are structural data abstractions, or patterns, while in the second one they are message expressions in the sense of object-oriented languages.

2.1 Structural access to data

In systems in which CEs are data abstractions, WMEs are to be selected according to their external structure, *i.e.* the one the user sees. A WME matches a CE if there exists a structural match between them. The determination of such a basic match is easy to implement. However, since rule condition parts are often made of a conjunction of CEs, the whole match step becomes highly combinatorial. Dedicated efficient match algorithms have been developed in this context mainly in the AI field (RETE, TREAT [6], TREE [7], etc.). Several systems were given an "object-oriented" flavor with classes, inheritance, and methods (LOOPS, ART). In full object-oriented environments, OKS [8, 9], for instance, was developed in Smalltalk.

All these systems rely on structural match and are then able to use the standard match algorithms. Nevertheless, if such systems are efficient, the major disadvantage of this approach with objects resides in the giving up of the encapsulation principle. The internal structure of objects, usually only visible inside the class, must be known outside, in the rules. As a consequence, such systems do not constitute a complete, clean, embedding of rules and objects, since one of the essential characteristic of objects, namely encapsulation, is lost.

2.2 Behavioral access to data

Another idea is to consider objects not through their structure but according to their behavior. CEs are then message expressions. Roughly speaking, a WME matches a CE if the message sent to it returns `true`. This is the essence of behavioral match. The advantage of this approach is to retain the encapsulation principle of the object model. Few systems are based on this approach (Opus, NeOpus).

Though in these systems the object aspects are totally preserved, we observe that the usual running characteristics of production systems are not guaranteed due to their match algorithm. They are either totally inefficient or even wrong. For instance, in NeOpus, which is based on a RETE matcher, the production system behavior may be incorrect mainly because the conflict set is not as systematically updated as it should be. Some rules, theoretically valid, are not actually triggered. Once more, such attempts cannot be considered as successful occurrences of a clean embedding of both paradigms.

3 The issue of embedding objects and production systems

In our opinion, a neat integration of object and rules should satisfy two constraints. It should (*i*) enable behavioral match with any arbitrary object, and (*ii*) provide a correct operating production system. The aspects related to the efficiency of match algorithms have then a lesser significance for integration.

3.1 The production system task

The conflict set contains the valid rules instantiations obtained by projecting the rule condition parts onto WMEs. The fundamental problem to be solved by a production system, or more precisely by the match algorithm on which it relies, is *to maintain a valid conflict set across cycles*, whatever happens between cycles.

3.2 Match algorithms

The basic match algorithm that produces the conflict set from the rule condition parts is a classical backtracking search algorithm. This "naïve" algorithm is highly combinatorial for conjunctions of CEs and needs to be run at each cycle. Optimized algorithms (RETE, TREAT, TREE, etc.) have been then developed to improve production system efficiency. This goal is mainly reached by the way the conflict set is updated. Instead of recomputing the whole conflict set, the update is selective, *i.e.* incremental. Considering a given cycle and the conflict set building task, the difference with the conflict set of the previous cycle is just computed. Only new valid and no more valid rule instantiations are treated, and then added, respectively removed, from the previous valid conflict set.

Such an interesting ability to exhibit but the difference in two successive conflict set states relies on an architecture which is itself dependent on one important point: the ability to update α-memories, *i.e.* the sets of WMEs that satisfy the same CE. But, as previously noted in section 2.1, all of these match algorithms have been elaborated in the framework of structural match. In this context, selective α-memory update is obtained efficiently through selective working memory update. When a particular WME is created, removed, or modified, since the system has control over these operations and then access to the WME structure, it is easy to know which CEs are concerned and to update relevant α-memories. Such an update is known as the sift step.

3.3 Why structural match algorithms are basically not suitable for objects

Aiming at a neat integration of objects and rules with behavioral match, any restrictive assumption on the object language should be avoided. Some attempts, *e.g.* NeOpus, followed the above principle while taking advantage of standard match algorithms such as RETE. However, the inferential properties of the resulting systems are not correct, causing lots of troubles when one tries to use them [10].

The initial reason for that is rather due to the inadequacy of standard match algorithms to behavioral match than to some specific properties of the latter. Though the initial contents of α-memories can be computed, the fundamental assumption of α-memory update for match algorithms is not basically satisfied in object-oriented environments with behavioral match. Since the behavior of an object may depend on many other invisible objects, on the grounds of encapsulation, it is not straightforward to determine how object behaviors have been changed after any modification in memory. This explains why there are many reasons for the sift step to be incorrect and, furthermore, for the whole match process.

Noting this apparent mismatch between behavioral match of objects and structural match algorithms, the issue is then to study to which extent behavioral match can be dealt with correctly.

4 Solutions to behavioral match

4.1 The naïve solution

The first point we would like to highlight is that the use of a basic "naïve" exponential algorithm perfectly satisfies the functional aspects of a clean embedding of objects and production systems. Each conflict set can be easily, though inefficiently, produced for each cycle by computing all current valid rule instantiations. The result is correct and behavioral match is used without breaking encapsulation. Although this solution is adequate, it would lead in practice to a combinatory explosion.

The above analysis (cf. 3.3) however provides clues for optimizations. As α-memory update is indeed fundamental to selective one of conflict set, the keystone of an efficient and full integration of objects and rules consists in solving the former problem. The consequence is that if an α-memory update is performed, then the efficient structural match algorithms can be reused, with all their optimizations (hash tables, node sharing), for the so called join step.[1] But, this sub-problem must be tackled for behavioral match. We propose below two solutions to α-memory update in this context.

Let us recall that an α-memory contains the set of objects that satisfy a CE. Thus, the issue of α-memory update is to exhibit the difference between two successive states of an α-memory. In the framework of behavioral match, objects contained by an α-memory have to behave identically in response to a message transmission. This message corresponds to the CE.

[1]This result relies on the assumption that join tests are equality tests. If not, the problem is slightly more complex and attention must be paid to the careful update of β-memories.

4.2 A first optimization: naïve α-memory update

In an object-oriented environment, nothing is known at the user level about object changes and, furthermore, about their possible effects on their behaviors after any rule firing. To realize α-memory update, an immediate first solution is to use the naïve algorithm to solely recompute α-memories. Their new content is systematically produced, and, by comparison with their old one, the difference is returned. From then on, the α-memory differences can be transfered to an optimized algorithm like RETE or TREAT.

On the complexity side, it must be noted that this naïve selective α-memory update is only linear in the numbers of WMEs and α-memories, while the combinatorial aspects of conflict set update are taken into account efficiently by a standard match algorithm.

While an optimization of the basic naïve match algorithm, this solution implies that every α-memory content be fully computed at each cycle. The following proposal consists in optimizing this naïve α-memory update.

4.3 A second optimization: selective α-memory update

A careful analysis of the sift step of structural matchers shows that the selective α-memory update is obtained by a selective working memory update, in which, (i), structurally modified WMEs are detected, and then, (ii), α-memories concerned by these modifications are appropriately updated. In order to optimize the previous model, we address these two problems in the behavioral match context. Hence, the first point must be reformulated. It becomes: *behaviorally modified objects must be detected.* From now on this becomes the main issue.

The proposed optimization relies on caching structures [11] which are object dependency graphs with a behavior similar to the Smalltalk dependency model. From a structural modification of an object, they enable the focus on every object in working memory whose behavior may be affected by such a modification.

4.3.1 Detection of possible structural object modifications

Noting that only the actions of fired rules may modify the working memory between two cycles, we collect all the objects that either received a message expression or were created. Only these objects may have been modified in their internal structure. But, insofar as behavioral match is the issue, we are interested among them in those whose behavior has changed.

4.3.2 From structural modification to possible behavioral modification

To know whether the behavior of a given object has been modified, we keep updated a dependency link between each object considered as a WME and all the objects upon which it directly depends. These objects are the contents of the variables that are accessible from its methods and that control its behavior (instance variables, class variables, global variables and SharedPools). These *supporting objects* are recorded in a state vector indexed by the *dependent object* in a dictionary SupportDictionary which constitutes the first caching structure. At the end of a cycle, a new state vector is computed for each possibly modified object. If the state vector is unchanged, the object behavior is necessarily unchanged at this object level. Otherwise, the object behavior may have changed and the object is actually considered as possibly modified.

However, we know that an object modification is likely to modify the behavior of many other objects. The problem now is to deduce from these recorded object changes all other behavioral changes.

4.3.3 Detection of all possible behavioral object modifications

To derive the complete set of modified objects from an initial set, we use a second caching structure, DependentDictionary. This dictionary contains the reverse information of the previous one: each WME object is associated with the set of objects it supports and which themselves depend on the current state vector of the object. Both dictionaries are updated accordingly. A recursive propagation in this structure yields the set of objects possibly (behaviorally) modified by the initial behavioral modifications.

4.3.4 Update of candidate α-memories

Now at the end of a cycle, we are provided with the set of objects whose behavior may have been modified. All other objects are not behaviorally changed and should not be considered. Only the modified objects are candidate for α-memory update. According to their class which provides candidate CEs, their behavior is actually tested to realize the selective α-memory update. The α-memory difference is returned, and then, like in the first solution, classical match algorithms can be reused.

It must be noticed that even if behavioral modification detection is based here on structural modification detection, the encapsulation principle is not violated since all these modifications are detected automatically; the user need not know anything about object internal structures.

This "state-saving" solution allows a finer selective, though not exact,[2] α-memory update than our first solution does. As to efficiency, complexity results for this sift step are not direct since they depend on the caching structures, which themselves depend mostly on the program structure.

5 Conclusion

We addressed in this paper the issue of behavioral match as the key to a neat embedding of objects and production systems. Firstly, we showed that, besides its inefficiency, the basic naïve match algorithms is a solution to this issue. Secondly, we identified the problem of α-memory update as the key to the reusability of standard optimized match algorithms, such as RETE. Thirdly, we proposed two solutions for this problem that preserve object encapsulation and production system algorithm correctness. One, naïve α-memory update, is based on a systematic α-memory re-evaluation, the other, selective α-memory update, on caching structures that permit to derive automatically all possible behavioral modifications from structural modifications. These solutions have been implemented in Smalltalk above a RETE network and constitute full integrations of objects and rules. Though incomplete, first experiments about efficiency are quite promising.

[2] In fact, this method does not guarantee that the object behaviors are actually modified.

We are currently working on other optimizations of the detection of behaviorally modified objects. In particular, we attempt to set up a finer indexing scheme for the caching structures. The ultimate goal is to obtain an exact selective α-memory update. We also study the possibility to use other match algorithms such as TREAT and TREE.

References

[1] D. G. Bobrow and M. Stefik. *The LOOPS manual.* Xerox Corporation, December 1983.

[2] B. D. Clayton. *Inference ART. Programming Tutorial version 3.0.* Inference Corporation, 1987.

[3] R. Atkinson and J. Larsen. Opus: a Smalltalk production system. In *Proceedings of OOPSLA'87*, pages 377–387, 1987.

[4] F. Pachet. Reasonning with objects: the NeOpus environment. In *Proceedings of East-EurOOpe*, Bratislava, Tchequoslovaquia, 1991.

[5] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[6] D. P. Miranker. *TREAT: a New and Efficient Match Algorithm for AI Production Systems.* Pitman; Morgan Kaufmann Publishers, 1990.

[7] J. Bouaud. TREE: the heuristic join strategy of a RETE-like matcher. In *Proceedings of the 13 th International Joint Conference on Artificial Intelligence*, pages 496–502, Chambéry, France, August, 29 – September, 3 1993. International Joint Committee on Artificial Intelligence.

[8] R. Voyer. Implémentation d'architectures efficaces pour la représentation des connaissances application aux langages Loopsiris et Oks. PhD thesis, Université Paris 6, 16 février 1989.

[9] R. Voyer. A simulation system of knowledge representation models in Smalltalk80. In *Proceedings of Western Simulation Multiconference: Object Oriented Simulation*, pages 177–184, Anaheim, January 1991.

[10] F. Pachet. Représentation des connaissances par objets et règles : le système NéOpus. PhD thesis, Université Paris 6, 25 septembre 1992.

[11] M. Perlin. Call-graph caching: Transforming programs into networks. In *Proceedings of the 11 th International Joint Conference on Artificial Intelligence*, pages 122–128, Detroit, MI, August 20–25, 1989. International Joint Committee on Artificial Intelligence.