Proceedings of International Conference
on Artificial Intelligence

Linear Backward Chaining with Knowledge Objects

Xiaoya Lin and Xindong Wu

Department of Software Development, Monash University
900 Dandenong Road, Melbourne, VIC 3145, Australia

Email: {lin,xindong}@insect.sd.monash.edu.au

Abstract

Rule based production systems are one of the most widely used models of knowledge representation in artificial intelligence. However, there are a number of inherent problems with existing rule based systems and tools. Most notably, they are inefficient in structural representation, and rules in general lack of software engineering devices to make them a viable choice for large programs. By applying knowledge object techniques [Wu et al. 95], this paper designs a factor-centered representation language, Factor++, which models the rule based paradigm into object-oriented (O-O) programming. Based on Factor++, a linear backward chaining algorithm, LBA, is designed to overcome the large computational requirements in rule based reasoning.

1 Introduction

Production systems are the most common form of architecture used in expert and other types of knowledge based systems. They are an important type of pattern-directed inference system. A production system consists of [Frost 86]: (1) a set of production rules or a rule base, (2) a database or working memory, and (3) a rule interpreter or inference engine. The working memory is used to store data about the problem in hand. It is the main data structure of production systems. A production rule in the rule base has a condition part called left hand side and an action or conclusion part called right hand side. The left hand side is responsible for comparing patterns associated with the left hand side with elements in the working memory. If the left hand side is satisfied by the working memory, the rule will become applicable and subject to being fired by the inference engine. Each rule represents a small chunk of knowledge relating to the given domain of expertise.

The inference engine in a production system selects and applies rules. It repeatedly applies rules to the working memory until certain stopping criteria are met. There are two basic inference methods, top-down inference or backward-chaining and bottom-up inference or forward-chaining [Bratko 90]. In the forward-chaining paradigm, rules are applicable if their condition part is satisfied by the working memory. In the backward-chaining methods, the system focuses its attention by only considering rules that are relevant to the problem in hand. There is also a third method–mixed method; forward-chaining and backward-chaining can be combined in various ways.

Production systems or rule based programming in general has many advantages, such as:

Modularity and modifiability Rules are easily coded and added to a production system. They may be added to the rule base without changing other rules.

Naturalness and simplicity The "If $\cdots$ Then $\cdots$" format of production rules is a natural and appropriate method for many kinds of human expertise. It provides an attractive simplicity for the representation of knowledge.

Knowledge intensive An expert system may be adapted for use in another problem domain by simply changing the rule base.

In the meanwhile, there are also several significant disadvantages:

Low efficiency A significant disadvantage of rule based systems is the large computational requirements to performing matching [Wu 93]. For naive production system algorithms, all but the smallest systems are computationally intractable. Even with Rete-like algorithms [Forgy 82, Miranker 87, Lee & Marshall 92], the non-polynomial complexity problem remains.

Non-structural representation Encapsulation of all relevant information of a single entity is hard with rule based programming.

Lack of software engineering devices Most rule based systems do not support models, information hiding, inheritance, and reuse to make them a viable choice for large programs [Wu et al. 95].

To avoid the second and third engineering problems, many researchers have started to integrate the rule based paradigm with object-oriented programming, a powerful technology from software engineering and the database community. Good examples are CLIPS [Donnell 94], L&O [McCabe 92] and Prolog++ [Moss 94]. This paper starts with the knowledge objects defined in [Wu et al. 95]; it models rule based programming into a factor-centered, object-oriented representation language Factor++, and designs a linear backward chaining algorithm LBA over Factor++.

2 Integration of rules and objects

Programming often involves breaking down complex problems into simpler constituents. Decomposing a problem using object-oriented programming [Booch 94, Meyer 88], or OOP, assists in the design of software, and makes the resultant software more maintainable, adaptable and recyclable. The basic idea behind OOP is the notion of classes of objects interacting with each other to accomplish some set of tasks [Meyer 88]. The objects have well-defined, self-contained behaviours. They interact with each other through the use of messages. When a task needs to be performed, an object sends off a message which specifies the task requirements. The receiving object then takes appropriate action in response to the message and responds by returning a message to the sender.

An OOP language offers at least the following facilities:

Data abstraction Abstraction extracts essential properties of a concept. It consists of data abstraction and procedural abstraction in OOP.

Inheritance The most common view of inheritance within the OOP field is that the subclass inherits all the properties and operations defined for the superclass and will probably add more [Snyder 86].

Encapsulation (or information hiding) Encapsulation combines data structures and functionality into objects. It also hides internal aspects of objects from its users and declares which features of an object are accessible.

Polymorphism Polymorphism is the concept of sending a message from one object to other objects. It means that the sending object does not need to know the receiving objects' classes.

Given the respective features of OOP and rule based programming, it is hard to say whether rule based programming or O-O languages are superior in computational strength [Wu 96]. Rule based programming expresses relationships between objects very explicitly. However, they don't express updates clearly. O-O programming is weak in inference power due to its procedural origin, but updates are defined clearly by assignments. It has the central ideas of encapsulation and reuse which encourage modular program development.

On one hand, while the O-O paradigm provides efficient facilities for encapsulation and reuse, it does not support inference engines for symbolic and heuristic computation. A clear advantage of rule based programming is that recur-

sion can be easily defined within rules while difficult in objects. On the other hand, rule based programming is very limited in structural representation and for large systems. Therefore, it would be very useful if we can integrate both of them in a seamless and natural way in order to exploit their synergism. It seems as if objects and rules are made for each other. Objects are the best way to simulate or model a problem domain. Rules can be designed to capture and encode human expertise that is applied to a problem domain. A natural way seems to be use objects for modeling the domain and rules to represent decision-making applied to the domain.

The two paradigms are both self-important and it is not appropriate to say that one should be the master and the other the slave in general, but depending on the application domains, choosing one of them as the basis and building the other on the top are necessary given that a seamless integration is not yet available and constructing one may well be very time consuming.

2.1 Incorporating rules into objects

It is argued in [Wong 90] that it is undesirable to implement objects within rule based programming, since rule based programming is not as portable as O-O programming. One way to get round this is to implement rules within objects. In Prolog++ [Moss 94], for example, an object layer is designed as an emcompassing layer for Prolog rules. In this paradigm, objects can call Prolog rules without any special annotation, and if a Prolog predicate is redefined within the Prolog++ class hierarchy, the definition will be taken by default. Rules can be used to make an object's semantics explicit and visible [Graham 93, Zhao 94]. They can also provide heuristic procedural attachment in methods. Actually methods within objects can always be implemented in the form of rules.

Rules can be defined in an independent rule base so that the methods in objects can call the corresponding predicates (rule heads), in the form of, e.g., obey statements in [Wong 90]. We can of course implement a set of rules with the same rule head in the form of objects, although some of the O-O advantages like inheritance, cannot be found from such objects.

Rules within objects can be divided into two categories [Odell 93]: constraint rules and derivation rules. The former define restrictions of object structure and behavior, such as consistency and constraints, and the latter are used to infer new data from existing data. In [Kwok & Norrie 94], for example, an object has four protocol parts: attributes, class methods, instance methods and rules. Rules can be activated by messages as methods.

2.2 Embedding objects into rules

In a rule based system, data in the working memory (or database) represents the state of the system and is used to fire rules. In an O-O system, the state is characterised by the the data items in objects. Therefore, a natural integration of objects and rules is to use objects as storage for the working memory in a rule based system, and rules execute actions depending on the values of objects in the working memory. A number of AI tools such as CLIPS [Donnell 94] have provided such facilities to embed objects in rules.

An alternative way is use O-O languages as the basis and implement rules which describe relationships of objects on the top of them. Domain expertise always relates to inter-relationships between objects, therefore a declarative query language for expressing these inter-relationships is very useful in integrated systems. This is the approach we will adopt in Section 3.

When heuristic rules are embedded within an object, the object can infer on these rules to provide heuristic answers when receiving queries from other objects. Such an object is called a *knowledge object* [Wu et al. 95]. Knowledge objects seem to fall into the category of incorporating rules into objects.

A knowledge object consists of at least three parts: data items, inheritance hierarchy, and rules. Methods can be implemented in forms of rules, or as a fourth component. Both rules and methods can be specified as public to allow global access or as private to prevent external visiting and modification.

3 Factor++: O-O modelling of rule based programming

Factor++ is new representation language based on knowledge object techniques. It models rule based programming into O-O programming, and provides facilities to represent all the information that can be represented in the rule schema + rule body language [Wu 94].

3.1 Rule schema + rule body

Rule schema + rule body [Wu 94] is an alternative representation language to rule based production systems based on an integration of rule based and numeric computations. Rule schemata in the language are used to describe the hierarchy among factors or nodes in domain reasoning networks while rule bodies, which comprise computing rules as well as inference rules, are used to express specific evaluation methods for the factors and/or the certainty factors of the factors in their corresponding rule schemata. By representing explicitly numeric computation and inexact calculus as well as inference rules, the language supports a flexible way to process procedural knowledge and uncertainty.

The rule base in rule schema + rule body consists a number of rule sets, each consisting of a rule schema with its corresponding rule body. A rule schema has a rule-like structure with the general form of: If $E_1, E_2, \ldots, E_n$ then A, where all of $E_1, E_2, \ldots, E_n$ and A are factors. A factor in rule schema + rule body a name involved in a domain expertise. It can be a logical assertion, a discrete set variable or a continuous, numeric variable.

3.2 Factor++: a factor-centered knowledge representation

A domain expertise always comprises a set of variables and the relationships between these variables. In rule schema + rule body, the variables are referred to as factors and the relationships are divided into two levels: rule schemata and rule bodies. In an object-oriented system, every entity is encapsulated into an object, a data structure combining the data properties of the entity and the procedures on the data. The data capture the attributes of the object, and the procedures capture the object's behaviours.

To integrate rules into objects by applying the knowledge objects introduced in Section 2.2, we define a *factor* in Factor++ by three characteristics: (a) the name of the factor (the factor identifier), (b) the names of variables and their value types that are relevant to the definition of the factor, and (c) the relationships between the factor and the relevant variables. Each of the relevant variables here is also a factor in the problem domain. The second characteristic relates to the rule schemata in rule schema + rule body, and the rule bodies in rule schema + rule body can be represented in the third characteristic. Inheritance is certainly an advantage of O-O modelling over rule based computation, but since the main topic of this paper is to model rules into objects, we will not address inheritance in particular.

Definition 1. A factor in Factor++ is a knowledge object. It combines a variable in a domain expertise, its data items, other relevant variables in the domain expertise that are used to define this variable, and the relationships between this variable and other variables. A factor has seven components: (a) the name of the factor, (b) the type of the factor (see Definitions 2 and 3), (c) data items of the factor, (d) a list of premiselinks (see Definition 4) which specifies relevant premise factors, (e) a set of procedures (see Definition 5) which defines the evaluation of the factor based on the the premise factors, (f) the inference status which will be described in Section 4, and (g) the certainty factor of the factor's value.

Definition 2. A *terminal factor* is an evidence factor, whose possible value is supposed to be given by the user.

Definition 3. A *goal factor* is not a terminal factor. Some procedures in the domain expertise are required to evaluate each goal factor.

Definition 4. A *premiselink* in a factor is organized as a list structure for a collection of variables. All the variables in the premiselink form a logical AND relation, and when values of these variables are all available, the factor can be evaluated.

Definition 5. A *procedure* in a factor contains the domain expertise to evaluate the value

of the factor and/or the certainty factor of the value/factor. In each procedure, there may be one or more inference rules similar to those in production systems. The inputs to the procedure must be declared in a corresponding premiselink of the factor.

A procedure in Factor++ corresponds to a rule body in rule schema + rule body. There is no procedure in terminal factors; however, there may be more than one procedure in a goal factor to define the evaluation of the factor and/or its uncertainty factor.

Rules in a traditional rule base have been divided into groups in Factor++, in a similar way as in rule schema + rule body, and have been embedded into factors. A knowledge base in Factor++ is a set of factor definitions, each of which is an independent knowledge unit.

3.2.1 Converting a rule base into Factor++

The following procedure converts a normal rule base into the Factor++ representation.

Step 1 Create terminal factors.

Step 2 If a rule contains two or more conclusion factors, split this rule into two or more, simpler rules so that each has only one conclusion factor.

Step 3 Group the rules that have the same conclusion factor into one rule set. Each rule set formed this way corresponds to a factor in Factor++.

Step 4 Set up factors by filling out their components.

3.3 Comparison between Factor++ and rule schema + rule body

A premise link in a factor by Definition 4 corresponds to a rule schema with the factor as the conclusion factor in rule schema + rule body [Wu 94]. A procedure by Definition 5 corresponds a rule body. Therefore, all information in rule schema + rule body can be represented in Factor++.

In rule schema + rule body, there may be more than one rule schema with the same conclusion factor; in Factor++, these rule schemata and their associated rule bodies are encapsulated into one knowledge object, which is the conclusion factor.

A factor in Factor++ is an object, and therefore the O-O features mentioned in Section 2 can be easily explored in Factor++.

4 LBA: Linear backward chaining on Factor++

In naive production system algorithms, the inference engine runs in 3-phase "match – conflict resolution - act" cycles. It first matches the rule base against the data in the working memory to find out a rule set, called *conflict set*, which is applicable to the current working memory, chooses a particular rule from the conflict set by some conflict resolution strategies, and fires the selected rule to change the working memory. Once the working memory is changed, a new cycle starts unless certain stopping criteria are met. There are two major problems with these naive algorithms. Firstly, the successful match of a rule with the working memory does not always mean the rule's immediate act. Some rules may be successful in matching with the working memory from the very beginning of a problem-solving process, but always fails to get the priority of act in each conflict resolution phase. When there are changes in the working memory, it also needs to be tested again and again. Secondly, a rule may fail to match with the working memory in an overall problem-solving process but it probably needs to be tested in each 3-phase cycle when the working memory is changed. These problems have caused large computational requirements to performing matching in production systems.

To avoid the large computational requirements, the rete-like algorithm [Forgy 82, Lee & Marshall 92] compile production rules in a rule base into a discrimination network and remember matched elements between cycles. Rete-like algorithms avoid iterating over the working memory elements and the rule base, but the non-exponential problems remain

[Forgy 82, Wu'93]. The LFA algorithm [Wu 93] runs in 2-phase "match - act" cycles on rule schema + rule body. It sorts the rule schemata in a rule base into a partial order once a rule base is established or updated. It matches rule schemata against the working memory one by one according to the partial order, and executes the corresponding rule body immediately if the matching of a rule schema is successful. The partial order guarantees that when a rule schema is matched, the possible values of the premise factors in the schema have been computed or collected. Therefore, no explicit conflict resolution is needed after the sorting.

4.1 Inference status in LBA

LBA is a linear backward chaining algorithm based on the Factor++ language designed in Section 3.2. Unlike LFA [Wu 93], LBA does not sort factors or rules into a partial order. With a given set of terminal factors, LBA can determine whether a factor is one of the premise factors of a goal factor or not when the goal factor is being visited. If a goal factor is unknown, LBA infers it immediately.

At each inference stage, a factor is in one of the following four states in LBA:

1. unknown
2. instantiated
3. failed
4. chaining

The unknown status indicates that the factor has not been processed. The instantiated status means that the factor has a value in the working memory. A factor being failed means that the factor cannot be evaluated, i.e., there is not enough evidence in the working memory. The last status, chaining, indicates that the factor is being processed. Before LBA processes a factor, the inference status of the factor is always unknown. After being processed, the factor is either instantiated or failed.

4.2 The chaining procedure

When LBA matches the factors on a premiselink of a goal factor, it checks the inference status of each of the factors on the premiselink. If all these factors are instantiated, then LBA calls the corresponding procedure of the premiselink in the goal factor to evaluate the goal factor. If there is a factor on the premiselink that is failed, the corresponding procedure is dropped immediately. If there is a factor on the Premiselink that is unknown, LBA moves control to the unknown factor, tries to evaluate it, and returns after this factor is either instantiated or failed.

The chaining process of LBA starts with a goal factor.

Step 1 Call the first Premiselink of the goal factor. Set the inference status of the goal factor as 'chaining'.

Step 2 Call the first factor of the Premiselink.

Step 3 Check the inference status of this factor:

- 3.1 If the inference status is unknown and the factor is a terminal factor, then ask the user to provide possible information about the terminal factor.
- 3.2 If the inference status is chaining and the factor is a goal factor, which indicates that a cycle has formed in the chaining path, then goto Step 4.
- 3.3 If the inference status is unknown and the factor is a goal factor, then move the control to this goal factor and try to evaluate it.
- 3.4 If the factor is instantiated, and if it is the last factor in the premiselink, then goto Step 5 else goto Step 3 to call the next factor on the premiselink. If the inference status of this factor is failed, then goto Step 4.

Step 4 Drop the current premiselink. If there is another untested premiselink, then call it and goto Step 2, else goto Step 6.

Step 5 Call the procedure corresponding to the current premiselink in the goal factor. If the conditions in the left hand side of a rule in the procedure are satisfied, then apply the

conclusion part of the rule, and set the inference status of this goal factor as instantiated, **else** set the inference status of this goal factor as failed. Return OK.

Step 6 **If** there is a cycle, **then** set the inference status of the current goal factor as unknown, and return ERROR, **else** set the inference status of the goal factor as failed and return OK.

An error in Step 6 indicates a dead cycle [Wu 93] in which no factors involved can be finally evaluated. For example, if A is a necessary condition for B, B is also a necessary condition for A, and there is no other way to determine A or B independently, we say A and B have formed a dead cycle. According to [Wu 93], a dead cycle is an error in a knowledge base. However, a cycle detected in Step 3.2 can be a live cycle, which can be resolved by other premiselinks in Step 4.

4.3 Time complexity and advantages

The time complexity of LBA is linear to the number of factors and the number of rules, if there is no dead cycle in a knowledge base. Each factor is processed only once, and LBA always chains a factor at the first time when it is met. After a goal factor is evaluated, it is treated as a terminal factor whether it is instantiated or failed, because its inference information is kept in the working memory. A rule within a factor is matched at most once in LBA. If a premiselink in a factor fails to match the working memory, the corresponding procedure will not be called, and therefore the rules within the procedure will not be matched and executed.

In addition to the linear time complexity, other advantages of the LBA algorithm include:

- It is a complete backward chaining algorithm, in which no explicit conflict resolution is necessary.
- It successfully integrates rules into objects.
- The partial order in LFA is not needed in LBA.

LBA and Factor++ have been implemented in C++. A detailed account of their implementation with example runs can be found in [Lin 96].

5 Conclusions

The efficiency of rule based systems and the integration of object-oriented and rule based programming paradigms are two major research theme in AI. There have been some achievements along these lines. CLIPS [Donnell 94] and Prolog++ [Moss 94] are two good examples for rule-object integration. CLIPS is a forward chaining system based on the rete algorithm [Forgy 82]. Prolog++ has its own notation, but compiles directly into Prolog. The rete algorithm is a method for comparing a set of patterns to a set of objects in order to determine all the possible matches. It does not iterate over the working memory and the rule base. LFA [Wu 93] is a linear forward chaining algorithm based on the rule schema + rule body representation language. It sorts the rules in a rule base into a partial order according to the hierarchy among factors, and then matchs rules one by one based on the order of rules. Its time complexity is $O(n)$ where n is the number of rules in a rule based system.

LBA designed in Section 4 is a linear backward chaining algorithm. Its knowledge representation is the Factor++ language designed in Section 3 which models rules into object-oriented programming. The most significant achievements of LBA are its linear chaining process for rule based systems and its integration of rules and objects. A factor in Factor++ and LBA is an object that holds its data items, inference information, and all the rules in which this factor is computed.

When a knowledge base gets larger and larger, inheritance and encapsulation become more and more important in the design and maintenance of the knowledge base. For future research, it is important to maintain LBA's linear performance, but inheritance and encapsulation will be explored on the Factor++ representation.

References

[Booch 94] G. Booch, *Object Oriented Analysis and Design with Applications*, Addison-Wesley, 1994.

[Bratko 90] Ivan Bratko, *PROLOG — Programming for Artificial Intelligence*, Second Edi-

tion, Addison-Wesley Publishing Company, 1990.

[Donnell 94] Brian L. Donnell, Object Rule Integration in CLIPS, *Expert Systems*, 11(1), 1994.

[Forgy 82] C. L Forgy, A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence*, 1982, 17–27.

[Frost 86] R.A Frost, *Introduction to Knowledge Base Systems*, Collins, 1986.

[Graham 93]
I. Graham, Migration Using SOMA: A Semantically Rich Method of Object-Oriented Analysis, *Journal of Object-Oriented Programming*, 5(1993), 9: 31–42.

[Kwok & Norrie 94] A.D. Kwok and D.H. Norrie, Integrating a Rule-Based Object System with the Smalltalk Environment, *Journal of Object-Oriented Programming*, 6(1994), 9: 48–55.

[Lee & Marshall 92] Ho Soo Lee and I.S. Marshall, Match Algorithms for Generalised Rete Networks, *Artificial Intelligence*, 54(1992), 249–274.

[Lin 96] Xiaoya Lin, Linear Backward Chaining with Knowledge Objects, *Master of Computing by Research Thesis*, Dept. of Software Development, Monash University, 1996.

[McCabe 92] F.G. McCabe, *Logic and Objects*, Prentice-Hall, 1992.

[Meyer 88] B. Meyer, *Object-Oriented Software Construction*, Prentice-hall, 1988.

[Miranker 87] D.P. Miranker, TREAT: A New and Efficient Match Algorithm for AI Production Systems, *PhD Thesis*, Columbia University, 1987.

[Moss 94] Chris Moss, *Prolog++: The Power of Object-Oriented and Logic Programming*, Addison-Wesley Publishing Company, 1994.

[Odell 93] J. Odell, Specifying Requirements Using Rules. *Journal of Object-Oriented Programming*, 6(1993), 2: 20–24.

[Snyder 86] A. Snyder Encapsulation and Inheritance in Object-Oriented Programming Languages, *Proc. of OOPSLA '86*, ACM, 1986.

[Wong 90] Limsoon Wong, Inference Rules in Object Oriented Programming Systems, *Deductive and Object-Oriented Databases*, W. Kim, J.-M. Nicolas, and S. Nishio (Eds.), Elsevier Science Publishers B. V., North-Holland, 1990.

[Wu 96] Xindong Wu, A Comparison of Objects with Frames and OODBs, *Object Currents*, Volume 1, Issue 1, January 1996.

[Wu 94] Xindong Wu, Rule Schema + Rule Body: A 2-level Representation Language, *International Journal of Computers and Their Applications*, 1(1994): 49–59.

[Wu 93] Xindong Wu, LFA: A Linear Forward-Chaining Algorithm for AI Production Systems, *Expert systems: The Int. J. of Knowledge Engineering*, 10(1993), 4: 237–242.

[Wu et al. 95] Xindong Wu, Sita Ramakrishnan and Heinz Schmidt, Knowledge Objects, *Infomatica*, 19(1995), 4: 557–571.

[Zhao 94] Liping Zhao, ROO: Rules and Object-Orientation, *TOOLs Pacific '94 Technology of Object-Oriented Languages and Systems*, 1994, 31–44.