N96- 12938

525-61

p. 21

CLIPS Enhanced with Objects, Backward Chaining, and Explanation Facilities.

M. ALDROBI, S. ANASTASIADIS, B. KHALIFE, K. KONTOGIANNIS, R. De MORI

McGill University, School of Computer Science, 3480 University St.
Montreal, Canada, H3A 2A7
demori@cs.mcgill.ca

Abstract

In this project we extend CLIPS, an existing Expert System shell, by creating three new options. Specifically, first we create a compatible with CLIPS environment that allows for defining **objects and object hierarchies**, second we provide means to implement **backward chaining** in a pure forward chaining environment, and finally we give some simple **explanation facilities** for the derivations the system has made. Objects and object hierarchies are extended so that facts can be automatically inferred, and placed in the fact base. Backward chaining is implemented by creating run time data structures which hold the derivation process allowing for a depth first search. The backward chaining mechanism works not only with ground facts, but also creates bindings for every query that involves variables, and returns the truth value of such a query as well as the relevent variable bindings. Finally, the **WHY** and **HOW** explanation facilities allow for a complete examination of the derivation process, the rules triggered, and the bindings created. The entire system is integrated with the original CLIPS code, and all of its routines can be invoked as normal CLIPS commands.

1. INTRODUCTION.

The C Language Production System (CLIPS) is an expert system tool written in and fully integrated with the C language. It provides high portability, and easy integration with external systems, making embedded applications easy. The primary representation methodology is forward chaining based on the Rete algorithm. A.I. methodologies not provided in CLIPS are the organization of seperate data into hierarchies which exhibit inheritance, the backward chaining inference strategy, and facilities to justify the reasoning process and the conclusions derived.

In [1] object oriented systems are discussed as one of the most promissing paradigms for the design, construction, and maintenance of large scale systems. This general model for computing has major applications in A.I. (e.g. [2, 3, 4]). Moreover, in [1] techniques such as deligation [5, 6], genericity [7, 8], conformance [7], enhancement [7], and inheritance [5, 6, 7, 8] are thought to be the basis of "object-related systems". The object-oriented system embedded onto CLIPS gives the capability to the user for defining objects using a frame-like structure, and allows the flow of information between objects by invoking methods. The above object-configuration was adopted to facilitate encapsulation, inheritance, and set-based abstraction, which are main characteristics of these systems [1, 9].

Furtheremore, production rules in CLIPS are not trigerred using a backward chaining inference mechanism. In [10] backward chaining is exhibited as an inference strategy that verifies or denies one particular conclusion or hypothesis. In [11] this mechanism is initiated by establishing a goal and then is matched with a conclusion of a production rule. This subgoal is substituted by a sequence of subgoals which are the premises of the relevent rule. The entire process terminates when all subgoals are proven to be true. Backward chaining is used in many applications such as diagnosis, decision making, and trouble shooting, and simplifies the explanation facilities [12, 13]. In our extension of CLIPS we use forward chaining to implement bakward chaining by creating data structures and traversing the structures in order to obtain a simulation of the mechanism.

Finally, the development of a "complete" shell requires to enhance the environment with informative explanations [14]. These facilities have recently been approached and many solutions have been proposed. In [15] clarity is the focus for the structure of an explanation, in [16] a proof-tree is created, while in [17] one creates a model suited for specific users. On the other hand, [18, 19] stress that the content of the explanations is of more importance than the form in terms of providing meta-rules that descibe the expert strategic knowledge. But, one of the most difficult problems in the explanation domain is to answer negative questions concerning facts that were not inferred by the shell [20, 21]. Our approach is to create a well-defined semantic structure containing the knowledge derived and the derivation process followed. This approach guarantees that the same explanation will be given for the same question [14].

In this paper we present a compatible with CLIPS environment allowing for defining objects as well as establishing hierarchies between objects, a backward chaining inference mechanism capable of performing bindings in queries involving variables, and finally two explanation facilities , WHY and HOW. The paper is organized in six sections. The first section deals with objects and object hierarchies where we present the object definition, the hierarchy schema adopted, and how attributes along with their values are inherited. In the second section the query language used to interrogate the objects is presented. In the third section we describe the backward chaining mechanism embedded in the expert system shell, as well as the data structures, and routines that implement it. In the fourth section the explanation facilities WHY and HOW are presented. Specifically, we investigate the data structures created during run time, and the mechanism involved for traversing the list in order to provide answers to the WHY and HOW facilities. In the fifth section we present a list of the new commands implemented, and what actions are taken accordingly for each command. Finally, we conclude by reviewing our work, identifying extensions we are working on, and exploring potential applications in the field of diagnosis, and troubleshooting.

It should be noted that in the first three sections some implementation details concerning algorithms used on the data structures are mentioned.

2. OBJECTS - OBJECT HIERARCHIES.

In an expert system shell such as CLIPS a necessity arises to construct a well defined hierarchical network of entities that will support user-system interaction resulting in a structured KB. These entities are objects that can be inherited via a network to other objects that reside lower in the hierarchy. Moreover we maintain a common syntax for facts and we use the hierarchy and the inheritance in order to create new facts and update the knowledge base in an efficient, well structured, and meaningful way.

2.1 OBJECTS.

The implementation represents an object as a record with the following fields :

- object name.
- object parent.
- object children.
- inheritance type[0] ... inheritance type[MaxAttributes].
- object type.
- attribute name[0] ... attribute name[MaxAttributes].
- attribute value[0] ... attribute value[MaxAttributes].
- comment.

The **object name** defines the name the user gives for the object which is unique in the entire hierarchy. The **object parent** is the parent of the object in the hierarchy, the **object children** points to a linked list containing all the children of the object, the **inheritance type** is either own or member (which will be explained shortly), and the **object type** is one of class, subclass, and instance. Finally, one has for each object a list of **attribute**

name value pairs which identify the characteristics of each object (they are limited to MaxAttributes), and a simple field for a **comment** is allocated for any special note about the object that must be known.

The **hierarchical network** is a set of objects distributed among three layers according to the semantic meaning of each object. The first layer contains objects of type class (the most general type of object), the second layer contains sublayers of objects of type subclass (the next least genaral type of object), and finally one has a layer of objects of type instance (the least general among all types of objects). See Fig. 1.

Inheritance is built in the network as a flow of information from objects with abstract semantic context to objects with specific semantic context. In this hierarchical network attributes, and their corresponding values are inherited from classes to subclasses, from subclasses to other subclasses, and from classes and subclasses down to instances. If the inheritance type is **member**, the flow of inheritance is not interrupted, while if the inheritance type is **own**, the values are not inherated and overwrite all other inherited values. It should be noted that each attribute name value pair for each object has a different inheritance type. The default type is member. In such a way our hierarchical network can be thought of as a set of oriented trees, where the roots are the corresponding classes.

In this schema the ideal implementation is a forest of trees, where the roots are classes, internal nodes are subclasses, and leaf nodes are instances. Also it is possible for nodes from one tree to have a parent or children in an other tree, interleaving the trees resulting in a complex forest structure. The data structure used in order to preserve all the properties, and the inheritance among the objects is to maintain n-ary tree structures for every class definition created, such that for every class maintain pointers that will allow traversals to move only down, for each subclass maintain pointers that will allow the traversal of a tree to move up or down, and for each instance maintain pointers that will allow traversals to move only up.

Inheritance alters the contents of the Knowledge base and the patterns we use to accomplish such a goal. The major observation here is that CLIPS handles and manipulates facts as strings and matching is done using string manipulation functions. With this observation in mind we restricted our facts to have a particular pattern for describing an attribute and its corresponding value as follows :

The [*attribute*] **of** [*object*] **is** [*value*].

which can be asserted directly as a CLIPS fact.

Moreover we use another pattern for all children of a class or a subclass. These patterns are :

All [*subclass*] **are** [*class*].

All [*instance*] **are** [*class*].

All [*subclass*] are [*subclass*].

All [*instance*] are [*subclass*].

All the above patterns create a complete set of facts, since the patterns encapsulate the information described by the attributes and the connections between the objects.

In such a way traversing a hierarchical network we can create facts that do not originate from the user, but can be inferred by the hierarchy. This has two advantages.

- First , the user specifies only the attributes absolutely necessary for an object assuming that all other attributes higher in the hierarchy are available.
- Second , we minimize the information stored in every object without losing any information.

Hence,the user specifies the world, and the system creates the relevant facts.

The final use of the pre-determined patterns is that knowing their syntax we can reserve positions for (single or multiple) bindings in rules or facts in forward or backward chaining. For example we know that a question :

The ?x of car is red

is a meaningful query and that the query

The color of car ?x red

is not a meaningful one.

It should be mentioned that the inheritance type controls the assertion of facts since, own attribute values participate in the generation of new facts, and overwrite all other inherited values for the same attribute. All inserted facts become immediately available to the rules, and participate equally in the derivation process.

2.2 AN EXAMPLE OF OBJECT HIERARCHIES.

Define the objects to be : Car (class), PrivateCars (subclass), Porsche (instance), BMW (instance).

Assign inheritance type own to : Porsche, and PrivateCars, for attribute name Color.

Assign inheritance type member to : Car, and BMW, for attribute name Color.

Define the connections to be : Porsche is an instance of PrivateCars, BMW is an instance of PrivateCars, and PrivateCars is a subclass of class Cars.

Assing the Color Red for Porsche, the Color Blue to PrivateCars, and the Color white to Cars.

The following facts are inserted in the KB of CLIPS :

- All Porsche are PrivateCars.
- All Porsche are Cars.
- All BMW are PrivateCars.
- All BMW are Cars.
- All PrivateCars are Cars.
- The Color of Porsche is Red.
- The Color of BMW is White.
- The Color of PrivateCars is Blue.
- The Color of Cars is White.

See Fig. 2 for details.

3. QUERY LANGUAGE.

Here we give a description of the query language applicable to the hierarchy network. This query language provides the means for obtaining information regarding the entries found in the network. Specifically we have the following possible queries :

a) (**Display?** [*object type*])

returns the description of all objects of the specified object-type

b) (**IdType?** [*object type*])

returns the object names given the type

c) (**GenType?** [*object name*])

returns the parent of the specified object

d) (**SpecType?** [*object name*])

returns all children of a specified object

e) (**GetAttribList?** [*object name*])

returns all attributes and their values an object may have.
This query takes care of own values and discrards member values for same attributes.

f) (**GetAttribValue?** [*object name*] [*attribute name*]) returns the corresponding value else returns false.

g) (**IsAttribValue?** [*object name*] [*attribute name*] [*attribute value*])

returns true or false

h) (**JustObj?** [*object name*])

returns the comments added for this object

Operations e,f,g,h take into account the inheritance that exists in the network.

The overall network constructed operates under CLIPS control, updating the KB, supplying the user with mechanisms for viewing the status of the system, and hence controlling the derivations that CLIPS produces as a result of applying ground facts to rules using forward or backward chaining.

4. BACKWARD CHAINING MECHANISM.

In this section we present a mechanism to implement **backward chaining** within the framework of the CLIPS shell. The aim is to provide means for analyzing an initial goal (query) into a set of subgoals each of which has to be solved using this method, up to the time the set of subgoals will contain only ground facts known to be true. The way we treat the set of subgoals implies that all subgoals in the set must be recursively satisfied in order for the initial goal to be true. For rules that have premise in disjunctive normal form we create a "set of subgoals" for each disjunction and every subgoal from each "set" must be proven true in order to have a successful derivation.

The user supplies a query and the system tries to match this query with an existing known true fact. If no such fact can be found then the rule(s) which has as its RHS this fact is considered and its premises are recursively considered as the new goals. The whole derivation process ends when all relevant rules have been examined and tested. Because we are interested not only in ground queries, but also in queries with variables, we use CLIPS's binding mechanism so that the appropriate bindings can be made.

In order to simulate the backward chaining mechanism we create a **Backward Chaining Network (BCN)** consisting of instantiated conclusions and facts interconnected as shown in Fig. 3.

This approach involves **four major steps.**

- In the first step we insert into the BCN all ground facts.
- The second step is to invoke Forward Chaining and add the derivations into the BCN as well.

- The third step provides a method of traversing this data structure of linked lists so that it implements the depth first search strategy. The way the linked lists are structured and traversed simulates the desired backward chaining.
- The final step is to create a dynamic data structure so that we can keep the derivation steps meaningfuly grouped in order to be used for the explanation facilities later on.

4.1 CREATING THE NETWORK (BCN).

The primary method of representing knowledge in CLIPS is rules of the form

IF [*PREMISE 1*] or

[*PREMISE 2*] or

....

[*PREMISE n*]

THEN

[*ACTION 1*] and
[*ACTION 2*] and

....

[*ACTION m*]

where each [*PREMISE i*] group could be a conjuctive expression combining different fact patterns, not necessarily ground facts, of the form

[*Fact 1*] and [*Fact 2*] .... and [*Fact k*]

and each [*ACTION i*] be of the form [*Asserted Fact i*]

Specifically, when a rule is fired we get a set of ground facts related in the following format :

[*Rulename*] - [*Asserted Fact i*] - [*Fact 1*] and ..... and [*Fact k*]

Moreover the logical combinations between entries in LHS of a rule as well as in the RHS of the rule are treated as follows :

1. If there are more than one OR related [*PREMISE*] in the LHS of a rule then we create a format as indicated above for each [*PREMISE*] expression.

2. If there are more than one AND related [*ACTION*] in the RHS of a rule then we create a format as indicated above for each [*ACTION*] expression.

3. For all other logical combinations involving (1) and (2) we create as many formats as can be derived when (1) and (2) are simultaneously applied (e.g. one format for each

[*PREMISE*] and [*ACTION*] combination).

These operations are equivalent to splitting the rules involving complex disjunctions and conjunctions into an equivalent set of rules of the form :

IF [*Fact 1*] and [*Fact 2*] and ... [*Fact m*] **THEN** [*Asserted Fact*]

Backward chaining is implemented using Forward Chaining in **two major steps**. In the **first step** we create a network of data structures in order to capture the relation between ground facts, premises, and conclusions in the Knowledge base, with the substitutions computed during Forward Chaining .In the **second step** we traverse the network so that we can find all possible derivation paths and bindings for a particular query. The way we traverse the network simulates a depth first search strategy. During the Forward Chaining derivation process we are creating our data structures using the following strategy :

For each rule we keep track of the premises and the conclusions that participated in each derivation step along with the Rule Name as Forward Chaining proceeds. For each conclusion reached we create a node pointing:

a) to the *next* Conclusion derived from Forward Chaining, and

b) to a list of rules that support this conclusion.

Each such rule node points to:

i) the next rule used and

ii) to a linked list representing the premises in conjunctive form.

In the case of premise groups which are combined disjunctively we maintain a different rule node pointing to a group of Premises which contains Facts in a conjunctive form.

See Fig. 3 for details.

4.2 TRAVERSING THE NETWORK (BCN).

After creating the network, the way we traverse it is implemented using a depth first search strategy with recursion. Specifically, the user specifies a query,which may contain variables, and the system tries to find legal bindings for the variables in order to prove or disprove the query. This is a two step process:

a) The first step is to create , if possible, legal bindings scanning all conclusions in the rule network, and, if found, generate the first goal.

b) The second step is to invoke a function in order to implement the desired backtracking. This is done using the BACKCHAIN(goal) function which is a recursive function. Specifically, if the goal is immediately derivable as a ground fact the function returns the bindings and the query has succeeded, otherwise finds the first premise that supports the current goal, sets the premise as the current goal and is re-invoked recursively. The result of the recursive execution is the creation of a derivation path which will be later used by the explanation facilities HOW and WHY. This derivation path forms a branch in the search tree so it is represented as a linked list of facts. It is possible that more than one derivation path exists so we keep them in different branches in a data structure as in Fig. 4. The process ends when there are no more conclusions to be tested in the BCN for possible legal bindings. The bindings are computed using a word by word comparison between two strings representing the ground term and the query. The notation for a variable is *?variable-name* and all words are tested with the words of same position in the instantiated ground term.

This process will return the correct answer as well as the relevant bindings because one is working in a subset of the Knowledge base that has been created using Forward Chaining. This new space is simply integrated, organized, and traversed in a way that simulates Backward Chaining.

5. JUSTIFICATION AND EXPLANATION.

The ability of expert systems to give explanations of their results and of the reasoning leading to those results is considered as one of the main advantages of these systems, as compared to usual programs. In rule based expert systems, explanations are often confined to the trace of the program execution. A trace is a record of fired rules. It may also include the data which allows these firings, cast into some readable form, preferably in a natural language. In some approaches, a distinction is made between WHY and HOW explanations.

All these types of explanations rely on the notion of trace. It seems that explanations produced depend heavily on the way the expert knowledge was encoded into rules. Often, explanations are more reminiscent of the language provided by the expert system shell rather than of the language employed by the domain expert.

WHY queries provide explanations on a conclusion that has been derived. Specifically, they allow for a quick reference on both the rule that supports the particular conclusion, and on the premises in the rule that supports this conclusion.

HOW queries provide explanations for the whole derivation The difference between the HOW and WHY facilities is that, WHY lists and gives information on the last rule triggered and HOW lists all possible derivation paths, rules, and bindings that suport the conclusion .

5.1 WHY QUESTIONS.

WHY questions are implemented using the data structure illustrated in Fig. 3. In this structure, which is implemented using linked lists we have three node categories : "CONCLUSION" nodes, "RULE" nodes, and " PREMISE" nodes. A "CONCLUSION" node contains a particular ground derivation obtained, and points to the rules that support it. Each "RULE" node contains the rule name and points to a linked list of "PREMISE" nodes. This structure allows for storing all the groups of premises that triggered the rule. Note that when two or more groups of premises are combined with OR's it may be the case that both groups may have contributed in the derivation. In such a case we keep them in separate lists under different "RULE" nodes having the same rule name.

Consider the following rule (CLIPS syntax) :

(defrule Rule1 (or ((p1) (p2)) ((p3) (p4) (p5)))
$\Rightarrow$ ((assert (c1)) (assert (c2))))

According to this rule if all the premises p1,p2,p3,p4,p5 are satisfied we will have two "RULE" nodes and five "PREMISE" nodes linked as follows :

One "CONCLUSION" node for c1 pointing to "RULE" node (Rule1) and to the next "CONCLUSION" node c2. Rule1 node points to an identical node Rule1 since we have two groups of ORed premises, and to a linked list of premises, consisting of p1,p2. The other "RULE" node points to null and to a linked list of premises representing the ORed second group of premises, p3,p4,and p5 (See Fig. 5).

Finally the second conclusion (i.e. c2) points to an identical , as above , structure .

Actually we will have as many answers to such questions as the number of times the corresponding rules where fired. Referring to Fig. 3, each ANDed group of premises going vertically forms one answer and we have as many answers as the number of these vertical groups going horizontally. These different answers are grouped by an OR in Fig. 3.

The way we implement a WHY [*FACT*] query for a specific conclusion is as follows. First we search for a particular "CONCLUSION" node that matches the query, then we traverse the relevant linked lists for every "RULE" node and every group of premises, and print in a user friendly format all the premises encountered, as well as all relevant rules names. (refer to Fig. 3).

5.2 HOW QUERIES.

Every time a **HOW query** is asked we compute all the possible derivation paths through which this conclusion had been derived. Thus, a derivation path is equivlent to a branch in the search tree. The computation is carried out in a recursive way using both the structure which implements the WHY questions and a new data structure, "DERIVATION PATHS", which is illustrated in Fig. 4. This data structure consists of two node types. "BRANCH" nodes and "FACT" nodes. We create it every time a HOW question is asked and destroy it thereafter. The answer to a HOW question is computed as follows :

When a **HOW** [*CONCLUSION*] **query** is asked , our goal becomes the "CONCLUSION" we want to satisfy, so we refer to the BCN to find the corresponding conclusion node. If we could NOT find any, because this "CONCLUSION" is neither derived nor a ground fact, then we return false. If the corresponding node is found we consider it as the current goal and we pick up the first premise from the BCN. This premise becomes our new goal and we repeat the same operation until there are no more premises in every premise group considered. We create a data structure of linked lists as in Fig. 4. "FACT" nodes represent the backward chaining derivation process, and the "BRANCH" nodes represent derivations performed in different premise groups.

Consider the following example :

```
(defrule Rule2 (or (d) ((a) (b)))
        ⇒ (assert (y)))
```

where (d),(a),and (b) are assumed to be ground facts for simplicity.

According to this rule and these facts, if we ask the question:

(HOW ?y)

then the possible branches are :

branch 1 : (y) (d);
branch 2 : (y) (a) (b);

6. NEW COMMANDS.

1. **(OBJECT)** : Creates interactively a new object and places it in the hierarchy network. Also one has the ability to query the hierarchy.

2. **(QUERY)** : Starts interactively backward chaining for a user specified query. It creates bindings and returns the corresponding truth value of the query.

3. **(HOW)** : Returns all possible path derivations for a specific query and explains how the particular subgoals were established and proved.

4. **(WHY)** : Returns information on the rule that proves a particular query and explains the truth values of the corresponding premises.

7. CONCLUSION.

In this paper we presented an extension of the CLIPS Expert System shell. We have created an enhanced version by allowing Objects and Hierarchies to be defined, adding a Backward Chaining mechanism for triggering rules, and finally creating two basic explanation facilities WHY and HOW. The whole system is fully integrated in the original CLIPS environment. The new version is currently running on a SUN 4 machine. Future extensions will be available in a DOS environment as well, so that maximum flexibility and portability can be obtained. Special care is taken so that the interface for the new commands is user friendly and much attention was paid on error checking and reporting.

Currently, we are integrating methods for objects. Methods will be defined as an attribute of an object and will have the same inheritance properties as any other attribute of the hierarchy. The internal structure of a method will be identical to a normal C function, and accessing attribute values will be acomplished by implementing two functions available to all methods that will get an attribute value given an object name and attribute name, and put an attribute value given an object name and attribute name (see Fig. 6).

Furthermore, we are integrating explanation facilities to answer questions of the form "WHY a conclusion was not derived ?", and "WHY a rule was not fired ?". The basis for answering these questions is to incorporate the closed-world assumption for the current status of our knowledge base.

Finally, we are implementing a user friendly interface in the form of a natural language system in order for a user to input definitions of rules, facts, objects, methods, and a menu driven system in order for the user to access all the commands that the new version of CLIPS supports.

These extensions are currently tried under a SUN 4 and a NeXT machine environment.

References

[1] Blair G., Gallagher J., Malik J., " Genericity vs Inheritance vs Delegation vs Conformance vs .. ". *Journal of Object Oriented Programming*, Sept./Oct. 1989 Vol.2 No. 3.

[2] Stefik M., Brobrow D. G., " Object Oriented Programming : Themes and Variations". *The AI Magazine*, 1985, pp.40 - 62.

[3] Goldberg A. and Robson D. " Smalltalk 80 : The Language and its Implementation", Addison Wesley, 1983.

[4] Morris J. H., Meyer B. , Nerson J., Matsuo M., " Eiffel : Object Oriented Design for Software Engineering *In*, Proceedings of the First European Software Engineering Conference, Strasbourg, France, Sept. 1987, pp. 120 - 124.

[5] Lieberman H., "Delegation and Inheritance : Two Mechanisms for sharing Knowledge in Object Oriented Systems". *Journees Languages Orientes Objet*, 1985, pp. 79 - 89.

[6] Stein L. A., " Delegation is Inheritance ", *Special Issue of SIGPLAN Notices*, Orlando, FL. Oct. 4 - 8, 1987, 22 (12), pp. 138 - 146.

[7] Horn C. " Conformance, Genericity, Inheritance and Enhancement", *In*: Proc. ECOOP, Paris, June 1987.

[8] Meyer B. " Genericity versus Inheritance", *In* : Proceedings of OOPSLA 1986, Conference, pp. 391 - 405, Portland, OR, Sept. 1986.

[9] Taenzer D., Ganti M., Podar S. " Object Oriented Software Reuse : The Yoyo Problem", *Journal of Object Oriented Programming*, Sept./Oct. 1989 Vol.2 No. 3.

[10] Jackson P. "Introduction To Expert Systems", Addison Wesley, 1986.

[11] Winston P. "Artificial Intelligence", Addison Wesley, 1984.

[12] Waterman A. D. " A Guide to Expert Systems", Addison Wesley, 1984.

[13] Buchanan B., Shortliffe E. " Rule Based Expert Systems ", Mc Graw Hill.

[14] Millet C., Gilloux M. " A Study of the Knowledge Required for Explanation in Expert Systems ". *1989 IEEE Fifth Conference on Artificial Intelligence Applications.*

[15] Weiner J., "BLAH, a System which explains its reasoning", *Artificial Intelligence* 15 (1 - 2) pp. 19 - 48, 1980.

[16] Erickson A. "Neat Explanation of Proof Trees". *Proc. of the 9th IJCAI*, Los Angeles, Ca, 1985.

[17] Forsyth R. "Expert Systems Principles and Case Studies". Chapman and Mall Publ. Co.

[18] Hasling D. W. "Abstarct Exlanations of Strategy on a Diagnostic Consultation System". *Proc. of the National Conference on Artificial Intelligence* , Washington DC, pp. 157 - 161, 1983.

[19] Clancey W. J., "Transfer of Rule Based Expertise through a tutorial dialog". Stanford University, Dpt. of Computer Science, 1979.

[20] Krekels X., " Why-not Explanations in Expert Systems and their Use as a Debugging Tool", *Cognitiva 85*, 1985.

[21] Safar B., Rousset M - C., "Negative and Positive Explanations in Expert Systems". Tech. Rep. LRI, Univ. d'Orsay, 1985.

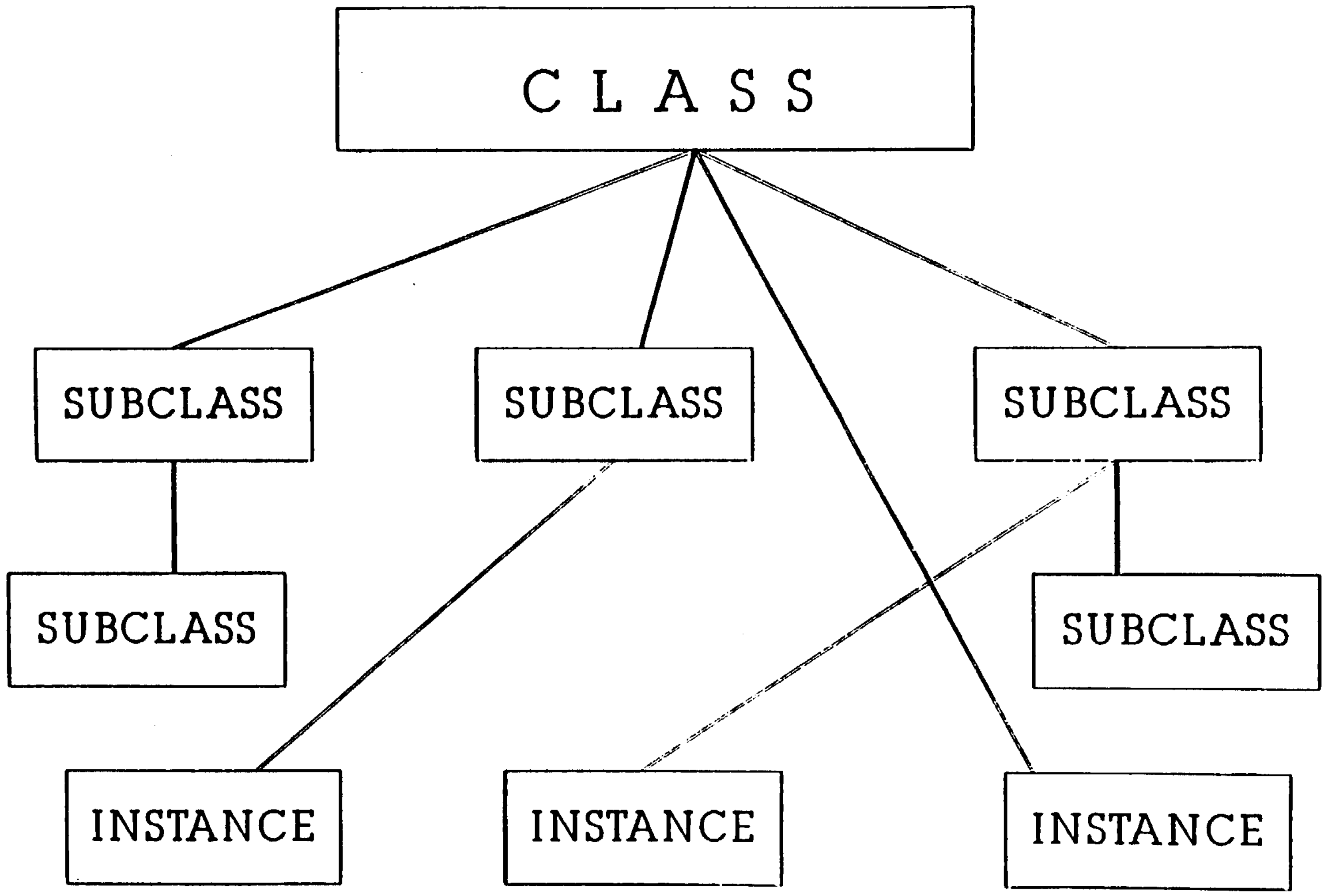


FIGURE #1 : An Object Hierarchy.

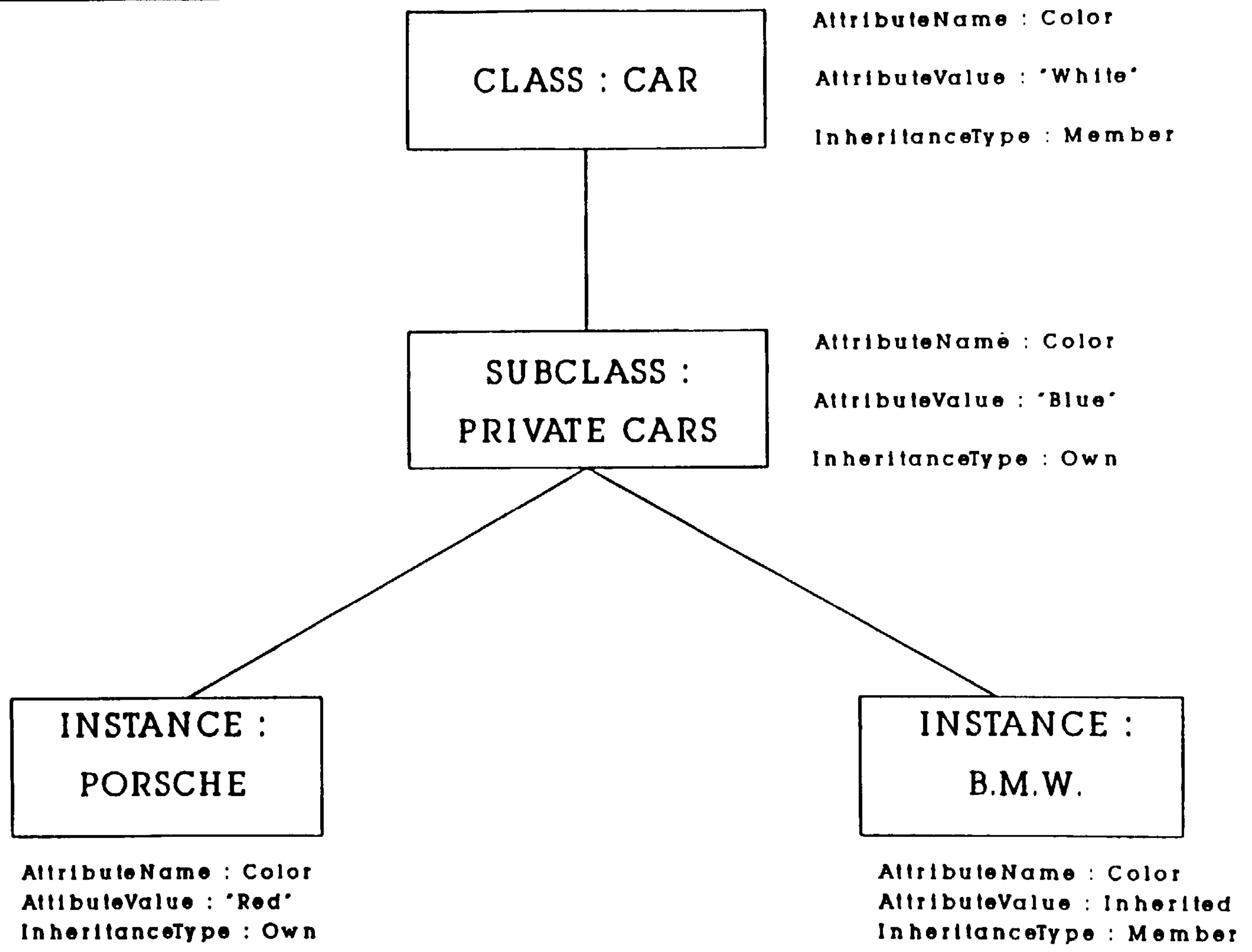


FIGURE #2 : An Example Of Object Hierarchies

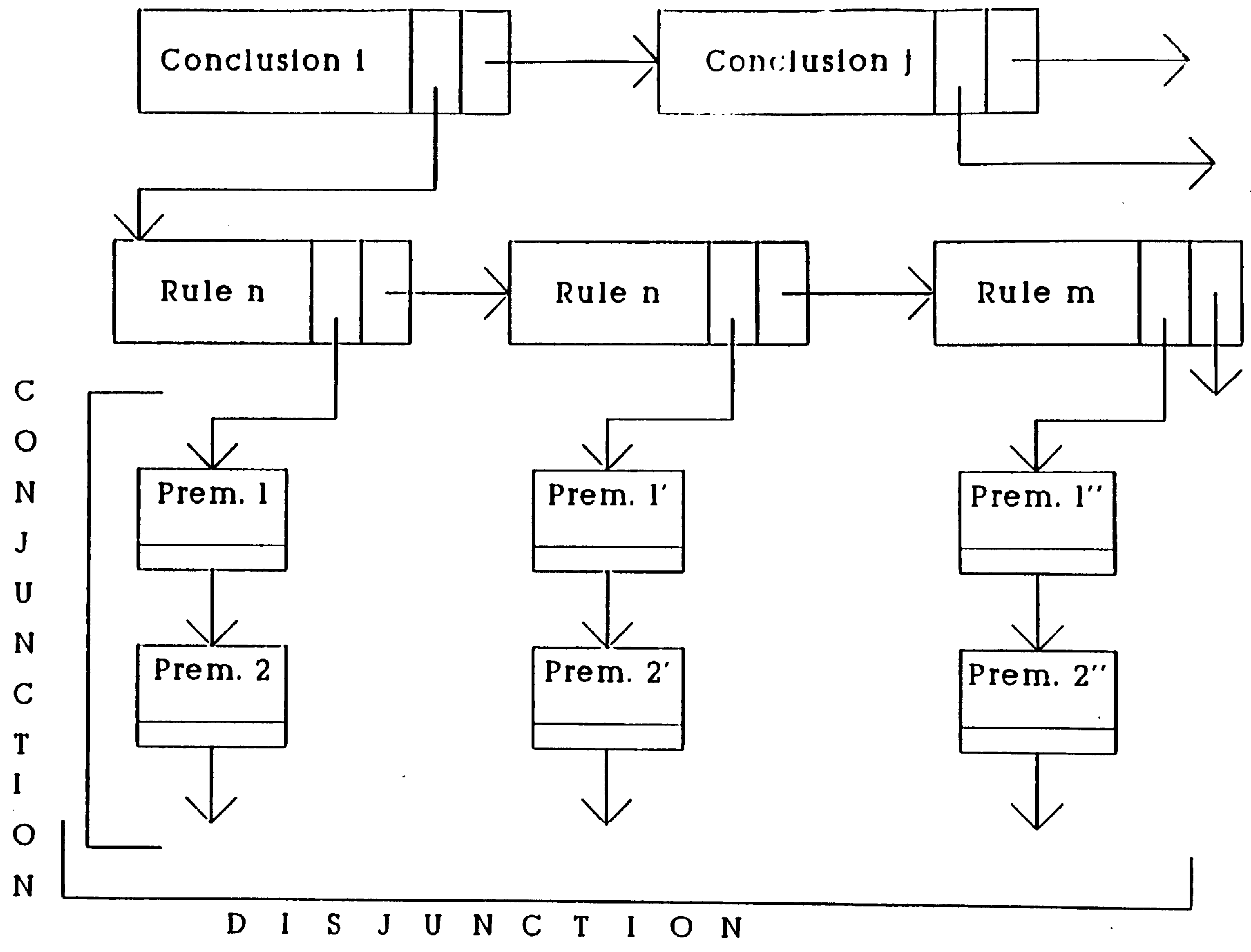


FIGURE # 3 : Backward Chaining Network.

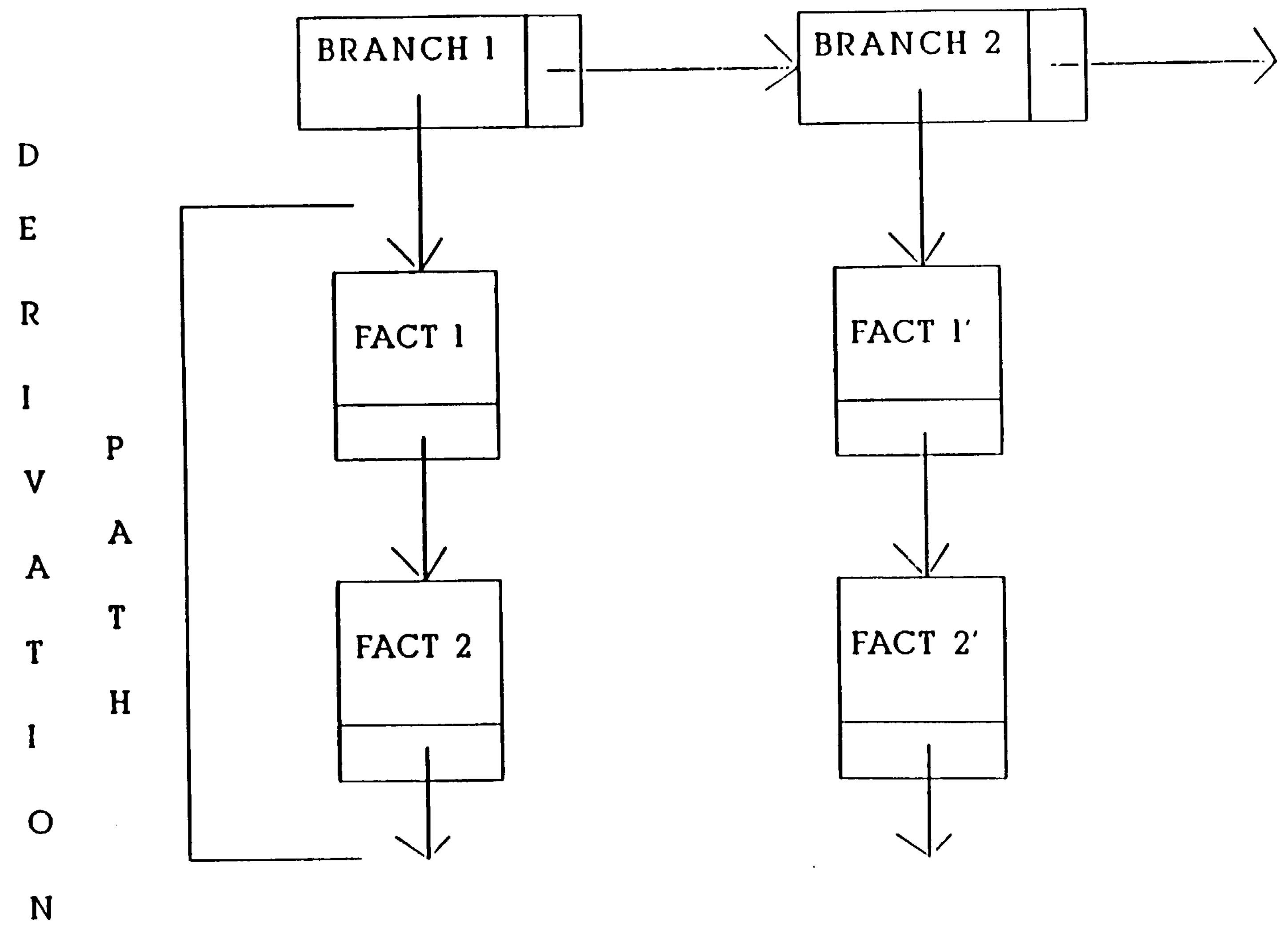


FIGURE # 4 : Derivation Paths.

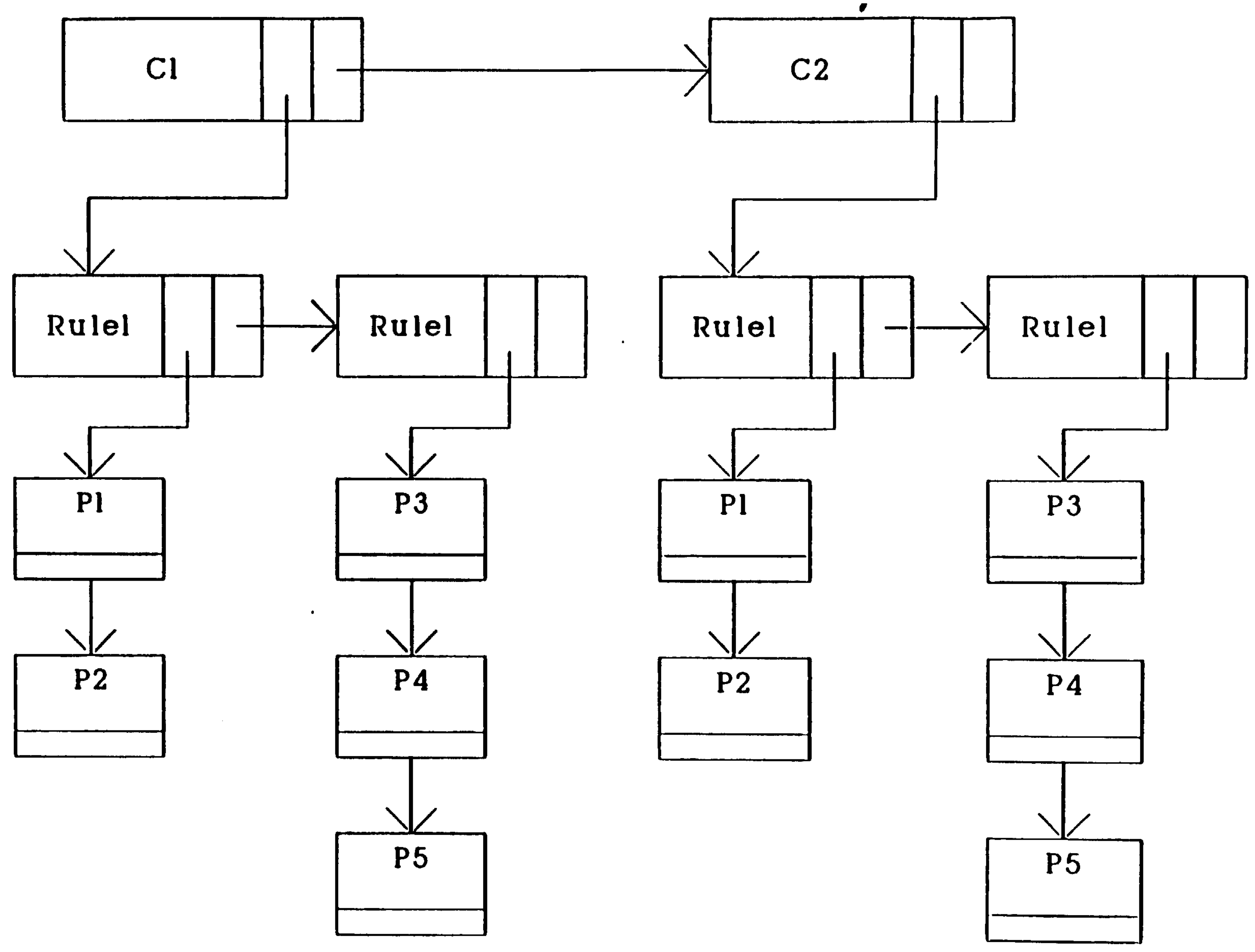
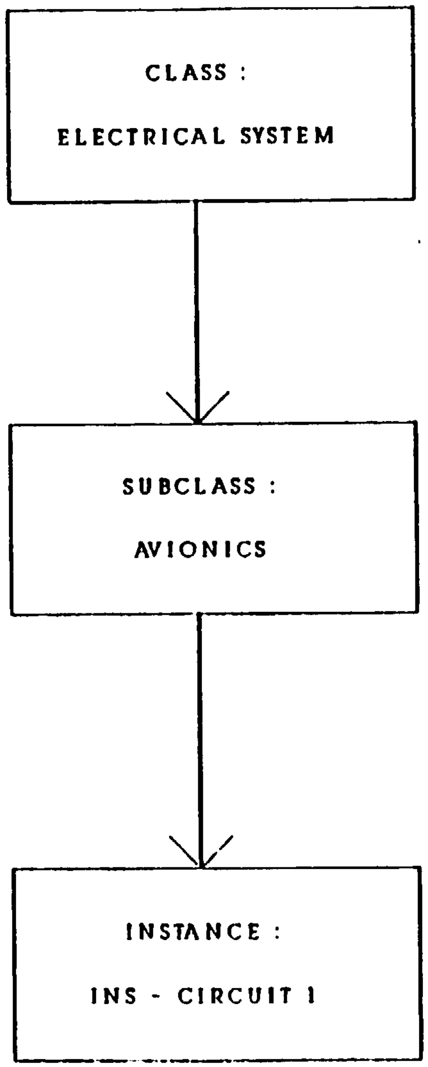


FIGURE # 5 : An Example BCN.



AttributeName : Current
AttributeType : Numerical
AttributeValue : Unknown
InheritanceType : Member

AttributeName : Resistance
AttributeType : Numerical
AttributeValue : Unknown
InheritanceType : Member

AttributeName : Power
AttributeType : Method

AttributeValue : POWER
InheritanceType : Member

(Inherited From Above)

(Inherited From Above)

```
(DefMethod POWER(ObjectName)

float Power(char ObjectName)

{

float I = getValue(ObjectName, Current)

float R = getValue(ObjectName, Resistance)

return I * I * R

}
```

FIGURE # 6 : An Example Method Implementation