A Spanning-Tree Based Approach for Generating Fault-Free Rule Bases

Nabil Arman
Palestine Polytechnic University, Palestine

Abstract: This article presents a new approach for generating fault-free rule bases, which have no redundancy, circularity, inconsistency, contradiction/conflict and unreachability. The approach makes use of spanning trees. A new algorithm, based on this approach, is presented which checks a rule base for different kinds of faults. The rule base is represented using a directed graph. The algorithm devises a spanning tree/forest of the underlying directed graph by treating the directed graph as an undirected graph, and checks for various faults and properties. The algorithm devises a new rule base (which is a subset of the original rule base) that is equivalent, in terms of its reasoning capabilities, to the original rule base, with the properties that the new rule base is free from redundancy and circularity. It also determines the set of rules that cause redundancy and circularity faults. Once the new rule base is determined, checking for the remaining faults, namely inconsistency, contradiction, and unreachability, can be performed easily using the generated structures.

الملخص: يعرض هذا البحث طريقة جديدة للحصول على مجموعات خالية من الأخطاء مثل التكرار، الدائرية، عدم التوافق، التناقض، وعدم الوصول وذلك باستخدام (Spanning Trees) لمجموعة القواعد بمخطط متجه، وتقوم الخوارزمية بايجاد (Spanning Tree/Forest) لمخطط وذلك بالتعامل مع المخطط المتجه وكأنه غير متجه ومن ثم اكتشاف الأخطاء والخصائص المختلفة. وتقوم الخوارزمية بايجاد مجموعة جزئية من القواعد المكافئة للمجموعة الأصلية من حيث القدرة الاستنتاجية الا أنها خالية من الأخطاء.

1. Introduction

The development of efficient algorithms to verify rule-based systems against different kinds of faults within the context of large rule–based systems have attracted a large amount of research efforts due to the important role of rule-based systems in various application domains, including Expert Systems (ESs), Active Database Systems, and Information Distribution Systems (IDSs) to name a few [1,2,3,4]. Verification is important to ensure the high quality of rule-based systems and to achieve an acceptable level of performance of these systems.

It is agreed that the effects of faults may appear in the performance of rule-based systems. Such faults may cause incorrect or undesired actions. Sometimes, these effects may be harmless, such as redundancy that may cause the system's performance to be inefficient. On the other hand, contradiction faults may lead to incorrect conclusions. It is worth mentioning that some redundancy faults may be included intentionally to gain some performance, instead of going into a long chain process to reach some conclusion/goal. However, in such cases the designer must be knowledgeable of the presence of such faults and their consequences from the practical point of view.

In this article we present the principles behind the new approach. Section 2 discusses rule-based systems faults. In section 3, the structure to be used in the algorithm

is presented. Section 4 presents the new algorithm in detail and provides a complete example of how the algorithm builds the spanning tree/forest. In section 5, detecting rule base faults is explained. Section 6 contains the computational complexity analysis of the algorithm. Finally, section 7 contains the conclusion.

2. Rule-Based Systems Faults

A taxonomy of faults that may appear in a rule base, and which can be detected by using verification algorithms, was presented in [5,6,7,8,9]. The taxonomy includes:

Redundancy: Two rules conclude the same outcome from the same input data.
Subsumption: Two rules conclude the same outcome, but one has additional constraints, which may or may not be necessary. It is a specific kind of redundancy.
Contradiction/Conflict: Two rules conclude different outcomes from the same input data.
Inconsistency: An antecedent of one rule is mutually exclusive to the consequent of such rule (or a chain of rules).
Circularity: The rule base contains a cycle inference chain, which may cause a backward-chaining inference engine to enter an endless loop.
Unreachability: Unreachability occurs if there is no path between any two given vertices.

3. The Structure Used in the Algorithm

Many transformation techniques for rule bases have been suggested in the literature. Petri Nets were described in [1]. In this approach, a rule base will be modeled as a Petri Net where parameter-value pairs corresponding to places and rules are analogous to transitions. Then the transition/place relationship modeled in a Petri-Net can be summarized in the form of an incident matrix. Decision-table-based processors were presented in [2]. In this approach, a decision table is created from the rules in the rule base. A directed-graph based approach was presented in [3], where the rule base is modeled as a directed graph and the process of anomaly detection is reduced to reachability among nodes. A transition-directed-graph-based approach, which is similar to [3] is presented in [4]. In this article, we use the transformation technique where the rule base is modeled as a directed graph. In this directed graph, nodes correspond to propositions and rule identifiers (in case a rule antecedent is a conjunction of propositions) and edges corresponding to the rules. Each rule will have a rule identifier.

4. The Spanning Tree Based Algorithm for Verifying Rule Bases

A spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G. Formally, let G=(V,E) be an undirected connected graph. A subgraph T=(V,E´) of G is a spanning tree of G iff T is a tree. An interesting property of a spanning tree is that it represents the minimal subgraph G´ of G such that V(G´)=V(G). By minimal, we mean the one with the fewest number of edges. Representing a rule base as a digraph as in [3,4], a spanning tree/forest of such a graph will be devised. Although spanning trees are generally obtained for undirected graphs, they still make sense for directed graphs. In our case, despite the fact that the underlying graph is directed, we will

be treating that as an undirected graph with some kind of interpretation of edges that create cycles. A variation of Kruskal's algorithm will be used, which is a greedy algorithm that builds a spanning tree by maintaining a forest (a collection of trees). Initially, there are |V| single-node trees. Adding an edge merges two trees into one. The algorithm terminates when enough edges are accepted. It turns out to be simple to decide whether edge (u,v) should be accepted or rejected. The appropriate data structure or approach is the union/find algorithm. This approach, as presented in ST_Fault_Detection algorithm in Figure 1, is of great importance to devise an equivalent rule base RB´, of m rules where m≤n, to the given rule base RB, which has n rules, such that RB´ will have the same reasoning capabilities as RB. Due to the fact that spanning trees are not unique, such a devised rule base may not be unique. One may order the rules based on a certain priority such as the cost/time of applying such a rule, and consider the rules in that order.

```
ST_Fault_Detection(
  Input: A digraph representation G of a rule base RB of n rules.
         An undirected representation G´ of a rule base RB´ of m rules
  Output: A spanning tree/forest T of G
          A set of rules creating the circularity C
          A set of rules creating the redundancy patterns R)
  initialize (S) /*S: Disjoint Set */
  T={}
  for all rules r do
        for all edges comprising rule r do
              choose the next edge (rule) <u,v>
              delete <u,v> from E
              u_set=find(u,S)
              v_set=find(v,S)
              if <u,v> does not create a cycle in T (i.e., u_set <> v_set) then
                    add <u,v> to T
                    edges_accepted= edges_accepted+1
                    set_union(S,u,v)
              else
                    if  find_path(u,v,S)=='C' then
                        /*<u,v> creates a cycle in the directed graph (G) */
                          add r to C
                    else /*<u,v> creates a cycle in the undirected graph (G´)*/
                          add r to R
                    endif
              endif
        end for
  end for
end ST_Fault_Detection
```

Figure 1. *ST_Fault_Detection* Algorithm

The *set_union*(S,r1,r2) procedure implemented by (S[r2]=r1) maintains the direction of the edges in the original directed graph, by using the straight forward implementation of the algorithm. The *find* procedure, as presented in Figure 2, determines the root of the set to which a vertex (e.g., x) belongs by using the implementation:

```
find(x,S)
  if(S[x] <= 0) then
    return x
  else
    return(find(S[x],S))
  end if
end find
```

Figure 2. *find* Procedure

To determine whether an edge <x,y> creates a cycle in the directed graph (G) or undirected graph (G´), the procedure *find_path*, as presented in Figure 3, can be used to check if two nodes x and y are on the same path in a certain disjoint set S.

```
find_path(x,y,S)
  while (S[x] != 0 & S[x] !=y)
    x=S[x]
  if (S[x] == y) then
        return 'R'  /* A redundancy pattern */
  else
        return 'C'  /* A circularity pattern */
  end if
end find_path
```

Figure 3. *find_path* Procedure

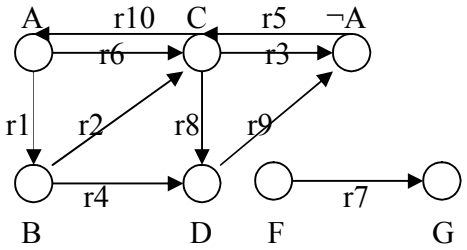
If x is reachable from y, then x and y are on the same path and adding an edge <x,y> does not create a cycle. However, it indicates that there is another path that connects x to y. Thus there is a redundancy fault pattern. On the other hand, if x is not reachable from y, then x and y are not on the same path and adding an edge <x,y> creates a real cycle. Thus, this is a circularity fault pattern.

The process of detecting various kinds of faults by formulating faults as reachability problems in the graph-based representation should be augmented/followed by a check of the in-degree of the rule identifier vertices that comprise a certain path in the fault patterns. Although the formulation gives a necessary condition for the existence of various kinds of faults in a rule base, the condition is not sufficient as long as rules with multiple antecedents are considered. To deal with this additional issue, we can compute the in-degree of the rule identifier vertices in the path(s) of the fault pattern to determine whether a certain fault satisfies the necessary or the sufficient conditions of representing a real fault. This information can be collected during the generation of the

spanning tree/forest and thus does not represent an expensive computational step. In addition, in real-world rule bases, the number of redundant and circularity fault patterns is relatively small and is assumed to be a constant number.

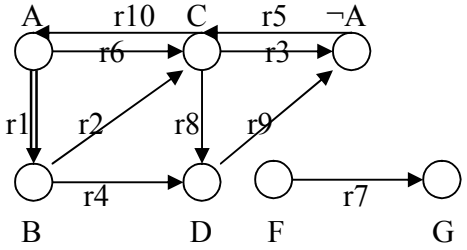
Example: Consider the following rule base (RB), where A,B,C, …etc. are propositions:
r1:A → B r2:B → C r3: C→ ¬A r4: B → D r5: ¬A → C
r6: A → C r7: F → G r8: C → D r9: D → ¬A r10: C → A

(a)



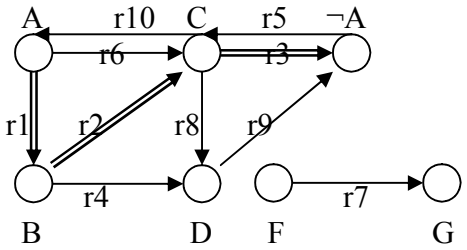
A digraph representing the rule base

(b)



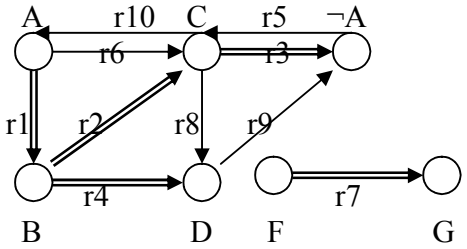
A digraph representing the rule base after accepting r1

(c)



A digraph representing the rule base after accepting r1, r2, and r3

(d)



A digraph representing the rule base after accepting r1, r2, r3, r4, and r7 and rejecting r5, r6, r8, r9, and r10

Figure 4. The execution of the ***ST_Fault_Detection*** on the rule base (RB), where double-lined edges/rules belong to RB′. Rules are considered based on their indices.

The rule base is represented as a directed graph as shown in Figure 4 (a). The algorithm accepts r1 since it doesn't create a cycle as in Figure 4 (b). It also accepts r2 and r3 for the same reason as shown in Figure 4 (c). The algorithms accepts r4 and rejects r5=<¬A,C>, since it creates a cycle, determined by computing u_set=find(¬A,S)=A and v_set=find(C,S)=A. Since u_set=v_set, then there is a cycle in the undirected graph. Starting from ¬A we can reach C, thus **C**={r5}. The algorithm also rejects r6=<A,C>, since it creates a cycle, determined by computing u_set=find(A,S)=A and v_set=find(C,S)=A. Since u_set=v_set, then there is a cycle in the undirected graph. Starting from A we can not reach C, thus **R**={r6}. It also accepts r7 since it doesn't create a cycle. The algorithm ejects r8=<C,D> since it creates a cycle, determined by computing u_set=find(C,S)=A and v_set=find(D,S)=A. Since u_set=v_set, then there is a cycle in the undirected graph. Starting from C we can not reach D, thus **R**={r6,r8}. The algorithm rejects r9=<D, ¬A> since it creates a cycle, determined by computing u_set=find(D,S)=A and v_set=find(¬A,S)=A. Since u_set=v_set, then there is a cycle in the undirected graph. Starting from C we can not reach D, thus **R**={r6,r8,r9}. Finally, the algorithm rejects r10=<C,A> since it creates a cycle, determined by computing u_set=find(C,S)=A and v_set=find(A,S)=A. Since u_set=v_set, then there is a cycle in the undirected graph. Starting from C we can reach A, thus **C**={r5,r10}. The final result of all these steps is shown in Figure 4 (d).

5. Detecting Rule Base Faults

The algorithm as presented in section 4 determines the set of redundant rules. It also determines the set of rules that cause cycles to be present in the rule base. Once these set of faults have been determined, it would be relatively simple to determine the rest of the well-known faults in a straightforward manner as follows:

(i) Inconsistency faults: Inconsistency occurs when an antecedent of one rule is mutually exclusive to the consequent of such a rule (or a chain of rules). This means that starting from one vertex (e.g., A) we can reach to its exclusive vertex ¬A. To check for this kind of anomaly, we first determine the set of exclusive vertices, and then we need only to check whether the exclusive vertices are in the same disjoint set and there is a path between them (using the procedure *find_path*).

Example: In the given example, A and ¬A are exclusive vertices. A and ¬A are in the same disjoint set. Since there is a path from ¬A to A, then there is an inconsistency anomaly.

(ii) Contradiction faults: A Contradiction/Conflict occurs when two rules conclude different outcomes from the same input data. This means that starting from one vertex/proposition (e.g. A) we can reach to two exclusive vertices (e.g., C and ¬C). To check for this kind of anomaly, we first determine the set of exclusive vertices, and then we only need to check whether the exclusive vertices are in the same disjoint set and none

of them is the root of the set. If they are in the same set and none of them is a root, then there is a contradiction anomaly, otherwise there is no contradiction anomaly.

Example: In the given example, A and ¬A are exclusive vertices. A and ¬A are in the same disjoint set. Since A is the root of the disjoint set, then this does not result in a contradiction pattern.

(iii) Unreachability faults: Unreachability occurs if there is no path between any two given vertices. To check for that, we first determine whether the two vertices are in the same disjoint set or not. If there are in the same set, we use the procedure *find_path* to determine whether there is a path between them, and in this case there is no unreachability anomaly, otherwise there is an unreachability anomaly. On the other hand, if two vertices are not in the same disjoint set, then we conclude (without using the *find_path* procedure) that there is an unreachability anomaly.

Example: In the given example, B and F are two vertices. They are in different disjoint sets, thus there is an unrachability anomaly in this case.

6. Computational Complexity

The spanning tree verification algorithm has a worst-case complexity of *O(nlogn),* where *n* is the number of rules in the rule base. Basically, this is a variation of Kruskal's spanning tree algorithm. The first *for* loop is iterated for *n* times. The edge components of each rule is assumed to be constant, and the second *for* loop has a complexity of *O(1)*. The complexity of *find* is *O(logn),* using smart union algorithms (union-by-size approach). Thus, the worst-case complexity of checking for all redundancy and circularity faults is *O(nlogn).*

Once the spanning tree and the data structures are obtained (which can be considered as a preprocessing step), the worst-case complexity of checking for inconsistency faults is *O(logn)*, since this can be determined by using the *find_path* procedure, which has a complexity of *O(logn)* using smart union algorithms. The worst-case complexity of checking for contradiction faults is *O(1)*, since a path compression technique can be used to obtain the disjoint sets. Finally, the worst-case complexity of checking for unreachability faults is *O(n),* which is dominated by the *find_path* procedure. The spanning-tree based approach represents a major improvement over Petri-nets approach, which has a complexity of $O(n^2)$ for detecting inconsistency and redundancy. Another interesting property is that new rules added to the rule base can be tested in *log(n)* time for creating circularity, redundancy, and inconsistency patterns, and in *O(1)* for detecting contradiction fault patterns.

7. Conclusion

We presented a spanning tree based approach for verifying rule-based system. The approach uses an algorithm that checks for redundancy and circularity in a rule base and generates a new rule base free of cycles and redundancy faults. Once the spanning tree and the associated data structures are built, checking for the remaining set of faults, such as contradiction/conflict, inconsistency, and unreachability can be performed easily.

References

[1] Agarwal, R., "A Petri-Net based approach for verifying the integrity of production systems," International Journal of Man-Machine Studies, Vol. 36, pp. 447-468, 1992.

[2] Nazareth, D., and Kennedy, M., "Verification of rule-based knowledge using directed graphs," Knowledge Acquisition, Vol. 3, pp. 339-360, 1991.

[3] Gragun, B., and Steudel, H., "A decision-table-based processor for checking completeness and consistency in rule-based expert systems," International Journal of Man-Machine Studies, Vol. 26, pp. 633-648, 1987.

[4] Hwang, Y., "Detecting Faults In Chained-Inference Rules in Information Distribution Systems," Ph.D. Dissertation, School of Information Technology and Engineering, George Mason University, 1997.

[5] Preece, A, and Shinghal, R., "Practical approach to knowledge base verification," SPIE, Vol. 1468 Applications of Artificial Intelligence IX, 1991.

[6] Preece, A., Shinghal, R., and Batarekh, A., "Principles and practice in verifying rule-based systems," The Knowledge Engineering Review, Vol. 7:2, 1992, 115-141.

[7] Preece, A., Shinghal, R., and Batarekh, A., "Verifying Expert Systems: A Logical Framework and a Practical Tool," Expert Systems With Applications, Vol. 5, pp. 421-436, 1992.

[8] Preece, A., "Towards a Methodology for evaluating expert systems," Expert Systems, November 1990, Vol. 7, No. 4.

[9] Preece, D., Talbot, S., and Vignollet, L., "Evaluation of verification tools for knowledge-based systems," International Journal of Human-Computer Studies, 47, 629-658, 1997.

Author's Biography

Nabil Arman: Nabil Arman received his BS in Computer Science with high honors from Yarmouk University in 1990 and an MS in Computer Engineering from Middle East Technical University in 1995. He also received an MS in Computer Science from The American University of Washington, DC in 1997. In May 2000, he received his Ph.D. in Information Technology/Computer Science from the School of Information Technology and Engineering, George Mason University, Virginia, USA. He is an Assistant Professor at the College of Engineering and Technology, Palestine Polytechnic University and was on a one-year leave teaching at the Department of Computer Science, Alisra University while part of this work was done. He is interested in Database and Knowledge-Base Systems, and Algorithms.