Fault Detection in Dynamic Rule Bases Using Spanning Trees and Disjoint Sets

Nabil Arman
Palestine Polytechnic University, Palestine

Abstract: *Many fault detection techniques/algorithms for detecting faults in rule bases have appeared in the literature. These techniques assume that the rule base is static. This paper presents a new approach/algorithm for detecting faults in dynamic rule bases, where rules may be added/deleted in response to certain events happening in the system being controlled by the rule base. This is performed by maintaining a set of structures, where new rules can be added to the dynamic rule base without the need to rebuild the structures that represent the rule base. The approach makes use of spanning trees and disjoint sets to check a dynamic rule base for different kinds of faults. The algorithm devises a tree/forest of the underlying directed graph by treating the directed graph as an undirected graph, and then checks for various faults and properties. The algorithm devises a new rule base (which is a subset of the current rule base) that is equivalent, in terms of its reasoning capabilities, to the current rule base, with the properties that the new rule base is fault free. This is performed as rules are being added to the dynamic rule base one at a time.*

Keywords: *Dynamic rule bases, rule base faults, spanning trees.*

Received October 1, 2005; accepted March 15, 2006

1. Introduction

Developing 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 [1-11]. Verification is important to ensure the high quality of rule-based systems and to achieve an acceptable level of performance of these systems. 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.

Many approaches and algorithms for fault detection have been presented and proposed in the literature. The Expert System Validation Associate (EVA) program was developed at Lockheed [11]. EVA program was used to check for rule redundancy, inconsistency and contradiction. A decision-table-based processor for checking completeness and consistency in rule-based systems was presented in [3]. The COVER tool was presented in [9]. The tool was designed to build upon the best features of earlier systems. It is used to check rules based on a subset of first-order logic. A Petri-Net based approach for verifying rule bases was presented in [1]. A Transition Directed Graph (TDG), which represents rule sets, was presented in [5, 6, 11]. TDG was used in the development of a set of algorithms to detect inconsistency, contradiction, circularity, unreachability, and redundancy in chained inference rules. These programs employed different approaches for detecting some faults. Based on these approaches many automated tools have been developed and used to inspect a rule-based system for known potential faults.

The automated tools and approaches didn't consider the issue of dynamic rule bases, which are characterized by the capability of being updated during the operation of the system, i. e., some rules may be added at a certain point in time and other rules may be deleted at other points in time. Adding/deleting rules affect the rule chains in rule bases. Such rule bases are common in active database systems and information distribution systems, where rules are added as new events occur in the system. If a dynamic rule base is fault free at a certain time, then deleting rules may generate unreachability faults only, by making some output vertices unreachable. Other types of faults, namely, inconsistency, redundancy/subsumption, circularity, and redundancy, can occur by adding new rules to a dynamic rule base. Adding rules may affect reachability if the rule being added involves an input vertex. Generally, this doesn't happen since we always assume that the set of input and output vertices are always known beforehand. Therefore, the focus here is on adding new rules to the dynamic rule base.

2. Rule-Based Systems Faults

A set of well-known faults that may appear in a rule base are presented in [7, 8, 9]:

- *Redundancy/Subsumption*: Two rules conclude the same outcome from the same input data. A special case of redundancy is subsumption, where, two rules conclude the same outcome, but one has additional constraints, which may or may not be necessary.
- *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. Structures 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 is 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 [3]. In this approach, a decision table is created from the rules in the rule base. A directed-graph-based approach was presented in [6], 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 [6] is presented in [4, 5].

In this paper, we use the transformation technique where the dynamic rule base is modeled as a directed graph as new rules are being added to the dynamic rule base. In this directed graph, nodes correspond to propositions and rule identifiers (in case a rule antecedent is a conjunction of propositions) and edges correspond to the rules. Each rule has a rule identifier. A spanning tree/forest will be devised using Kruskal's like algorithm. During the operation of the algorithm, disjoint sets will be generated. These sets will be used for detecting various kinds of faults while the dynamic rule base is being updated.

4. Fault Detection Algorithm for Dynamic 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 dynamic rule base as a directed graph, 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 treat that as an undirected graph with some kind of interpretation of the edges that create cycles. A variation of Kruskal's algorithm is used, which is a greedy algorithm that builds a spanning tree by maintaining a forest (a collection of trees) as new rules are being added to the dynamic rule base. Initially, there are $|V|$ single-node trees. Adding an edge merges two trees into one. 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 *DRB_Fault_Detection* algorithm in Figure 1 is of great importance to devise an equivalent rule base RB', of m rules where $m \leq n$, to the current dynamic rule base RB, which has n rules, such that RB' has 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.

The pseudocode of the algorithms uses a set of conventions. Block structures are indicated using statement indentation. An "*end if*" matches every "*if*", an "*end while*" matches every "*while*", and an "*end for*" matches every "*for*". The looping and conditional constructs have the same interpretation as in C. The algorithms can be translated to working C or Java programs in a straightforward manner.

DRB_Fault_Detection algorithm checks the current rule base when a new rule is added to the dynamic rule base as follows:

1. It calls *Check_for_Redundancy_ and_Circularity (r, RB', C, R, S)* procedure to check if it causes a redundancy or circularity fault pattern. In this call, r is the new rule, RB' is the current fault-free dynamic rule base, C is the set of circularity fault patterns, R is the set of redundancy fault pattern, and S is the disjoint sets.
2. It checks if the new rule r contains exclusive vertices. If r contains exclusive vertices, it calls *Check_for_Inconsistency_and_Contradiction (r, S)* procedure to perform this check.

3. The algorithm calls *Check_for_Unreachability (r, S)* procedure to check for potential unreachability faults.

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 straightforward 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. To determine whether an edge <*x*, *y*> creates a cycle in the directed graph or undirected graph, 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*. 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.

```
DRB_Fault_Detection (r, RB′, C, R, S)
    Check_for_Redundancy_and_Circularity(r, RB′, C, R, S)
    If r contains exclusive vertices then
        Check_for_Inconsistency_and_Contradiction(r, S)
    End If
    Check_for_Unreachability(r, S)
End DRB  Fault  Detection
```

(a) DRB_Fault_Detection algorithm.

```
Check_for_Redundancy_and_Circularity (r, RB', C, R, S)
  For all edges comprising rule r do
    Choose the next edge <u, v>
    Delete <u, v> from r
    u_set = find (u, S)
    v_set = find (v, S)
    If <u,v> does not create a cycle in RB'  then
      /* (i. e., u_set <> v_set)*/
      Add <u, v> to RB'
      set_union(S, u, v)
    Else
      If find_path(u, v, S) == 'C' then
         /*<u, v> creates a cycle in the directed graph*/
         Add r to C
      Else /*<u, v> creates a cycle in the undirected graph*/
         Add r to R
      End If
    End If
  End For
End Check  for  Redundancy  and  Circularity
```

(b) Check_for_Redundancy_and_Circularity procedure.

```
Check_for_Inconsistency_and_Contradtion(r, S)
  /* RB' is not affected by this procedure. */
  For each vertex v in r do
    If v is an exclusive vertex then
      root_of_v = Find (v, S)
      /* vp is the exclusive vertex of v */
      root_of_vp = find (vp, S)
      If (root_of_v == root_of_vp) then
        While (S[root_of_v]!=0 && S[root_of_v]!=root_of_vp)
            root_of_v = S[root_of_v]
        End While
      End If
      If (S[root_of_v] == root_of_vp) then
          Display "r causes Inconsistency Fault Pattern"
      Else
          Display "r causes Contradiction Fault Pattern"
      End if
    End if
  End for
End Check_for_Inconsistency_and_Contradiction
```

(c) Check_for_Incosistency_and_Contradiction procedure.

```
Check_for_Unreachability(r, S)
  /* RB' is not affected by this procedure. */
  For each pair of vertices (x, y) in r do
    root_of_x = find (x, S)
    root_of_y = find (y, S)
    If (root_of_x == root_of_y) then
      While (S[root_of_x]!= 0 && S[root_of_x]!=root_of_y)
        root_of_x = S [root_of_x]
      End While
      If (S [root_of_x] == root_of_y) then
       Display "r causes Unreachability Fault Pattern"
      End If
    End If
  End For
End Check_for_Unreachability
```

(d) Check_for_Unreachability procedure.

Figure 1. Fault detection algorithm for dynamic rule bases.

```
find (x, S)
  If (S[x] <= 0) then
    Return x
  Else
    Return (find(S[x], S))
  End If
End find
```

Figure 2. Find procedure.

```
find_path(x,y,S)
  While (S[x] != 0 & S[x] !=y)
    x = S[x]
  End While
  If (S[x] == y) then
          Return 'R'  /* A redundancy fault pattern */
  Else
          Return 'C'  /* A circularity fault pattern */
  End If
End find_path
```

Figure 3. Find_path Procedure.

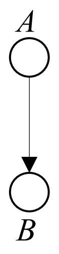
The process of detecting various kinds of faults by formulating faults as reachability problems in the graph-based representation should be augmented/followed by a check of the in-degree of the rule identifier vertices that comprise a certain path in the fault patterns. Although the formulation gives 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. In generating the new rule base, the assumption is that linear-edge-rule-relationship holds for the graph representing the rule base. This property says that if the graph has m edges and the rule base has n rules, then $O(m) = O(n)$.

When a change happens to the dynamic rule base, the new approach, as presented, checks for potential redundancy faults. It also checks for potential circularity faults in the current dynamic rule base. Once these sets of faults have been considered, it would be relatively simple to check for the rest of the well-known faults in a straightforward manner. An inconsistency fault 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 a vertex (e. g., A), we can reach to its exclusive vertex $\neg A$. 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*). A contradiction/conflict fault pattern 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 $\neg C$). To check for this kind of fault, 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. Unreachability faults occur 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 they are in the same set, we 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 that there is an unreachability anomaly. The benefit of our approach is its ability to detect faults as the dynamic rule base is being updated. If a rule r is added to the dynamic rule base, then the new dynamic rule base can be verified against various faults without having to rebuild any structures from scratch.

Example: Assume we started with an empty dynamic rule base. The following actions happen during the operation of a system controlled by a dynamic rule base. $A, B, C, \ldots$etc. are propositions:

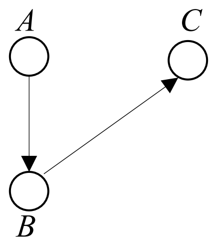
(*a*) *Rule $A \rightarrow B$* is added

A digraph representing the rule base after accepting $A \rightarrow B$.



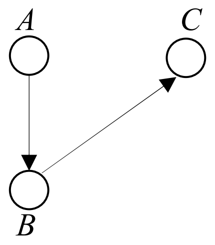
(b) Rule $B \rightarrow C$ is added

A digraph representing the rule base after accepting *$A \rightarrow B$ and $B \rightarrow C$.*



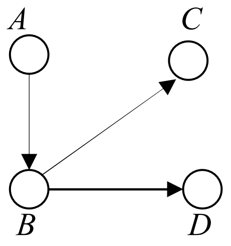
(c) *Rule $C \rightarrow \neg A$* is added

A digraph representing the rule base after accepting $A \rightarrow B$ and $B \rightarrow C$. $C \rightarrow \neg A$ is rejected since it creates an inconsistency fault pattern, i. e., a path from A to $\neg A$.



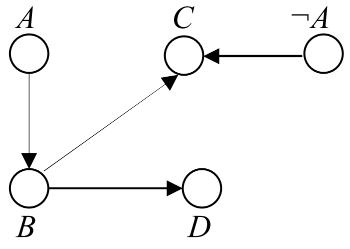
(d) *Rule $B \rightarrow D$* is added

A digraph representing the rule base after accepting $A \rightarrow B$, $B \rightarrow C$, and $B \rightarrow D$.



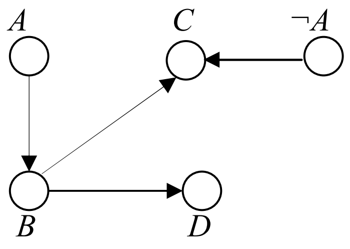
(e) *Rule $\neg A \rightarrow C$* is added

A digraph representing the rule base after accepting $A \rightarrow B$, $B \rightarrow C$, $B \rightarrow D$, and $\neg A \rightarrow C$.



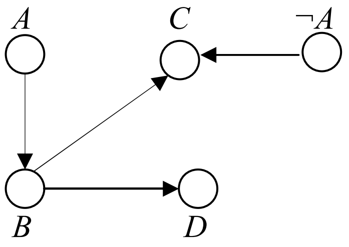
(f) *Rule $A \rightarrow C$* is added

A digraph representing the rule base after accepting $A \rightarrow B$, $B \rightarrow C$, $B \rightarrow D$, and $\neg A \rightarrow C$. $A \rightarrow C$ is rejected since it creates a redundancy fault, i. e., there exists a path from A to C through B.



(g) *Rule C→A* is added

A digraph representing the rule base after accepting $A\rightarrow B$, $B\rightarrow C$, $B\rightarrow D$, and $\neg A\rightarrow C$. $C\rightarrow A$ is rejected since it creates a circularity fault pattern, i. e., a cycle $A\rightarrow B\rightarrow C\rightarrow A$ *would occur.*



5. DRB_Fault_Detection Algorithm Computational Complexity

DRB_Fault_Detection algorithm is a variation of Kruskal's spanning tree algorithm, with no need to sort preprocessing step. Therefore, it has a worst-case complexity of *O(nlogn),* where *n* is the number of rules being considered for addition to the dynamic rule base. *DRB_Fault_Detection* algorithm calls *Check_for_Inconsistency_and_Contradtion* procedure *n* times (once for each rule added to the dynamic rule base). The *for* loop for the edge components of each rule is assumed to be constant with 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). DRB_Fault_Detection* algorithm calls *Check_for_Inconsistency_and_Contradtion* procedure to check for inconsistency and contradiction fault patterns. Once the spanning tree and the data structures are obtained, 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. Our approach represents a major improvement over Petri-Nets approach, which has a complexity of $O(n^2)$ for detecting inconsistency and redundancy [1].

6. Experimental Results of the Fault Detection Algorithms

Generally, an empirical study is an integral part of the analysis of algorithms. To study the experimental complexity of our algorithms, the fault detection algorithms were implemented in C and executed on different kinds of dynamic rule sets represented by the graph representation. A number of added rules generate a set of faults, and the algorithms detected all these faults. A performance profile, which represents the amount of time the algorithms consume, was also generated. This has been compared with the Petri Nets algorithm profile. The performance measurements have shown that our approach outperforms the Petri Nets approach. A set of four test cases, consisting of 100, 200, 300, and 400 rules were considered. Each test case uses a randomly-generated set of rules with a number of faults resulting from the random generation of the rule sets. The result of each case is plotted for our approach and the Petri Nets approach as shown in Figure 4. The performance measurement confirms the earlier theoretical analysis of the various algorithms. Using the timing data, the shapes of the curves are determined.

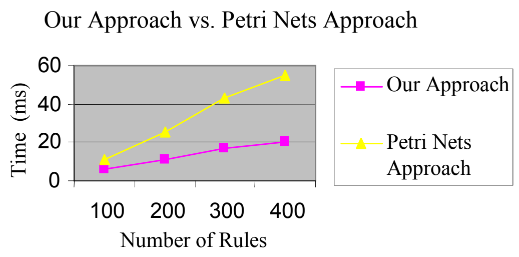


Figure 4. Experimental results of fault detection algorithms.

7. Conclusion

A new approach, based on spanning trees and disjoint sets, for verifying dynamic rule bases is presented. The approach uses an algorithm that checks for various fault patterns in a dynamic rule base and generates a new rule base, from the rules considered so far, that is fault free and has the same reasoning capabilities as the original rule base. Once the spanning tree(s) and the associated disjoint sets are built, checking for different faults as new rules are being added to the dynamic rule base can be performed in a straightforward manner. In addition, an empirical study, which confirms the theoretical analysis, is also presented.

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] Arman N, Richards D., and Rine D., "Structural and Syntactic Fault Correction Algorithms in Rule-Based Systems," *International Journal of Computing and Information Sciences (IJCIS)*, vol. 2, no. 1, pp. 1-12, 2004.

[3] Hwang Y., "Detecting Faults In Chained-Inference Rules in Information Distribution Systems," *PhD Dissertation*, School of Information Technology and Engineering, George Mason University, Virginia, USA, 1997.

[4] Hwang Y. and Rine D., "Algorithms to Detect Chained-Inference Faults in Information

Distribution Systems," *in Proceedings of the 2001 ACM Symposium on Applied Computing*, Las Vegas, Nevada, United States, pp. 679-685, 2001.

[5] 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.

[6] Nazareth D. and Kennedy M., "Verification of Rule-Based Knowledge Using Directed Graphs," *Knowledge Acquisition*, vol. 3, pp. 339-360, 1991.

[7] Preece A. and Shinghal R., "Practical Approach to Knowledge Base Verification," *SPIE*, vol. 1468, Applications of Artificial Intelligence IX, 1991.

[8] Preece A., Shinghal R., and Batarekh A., "Principles and Practice in Verifying Rule-Based Systems," *The Knowledge Engineering Review*, vol. 7, no. 2, pp. 115-141, 1992.

[9] 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.

[10] Preece D., Talbot S., and Vignollet L., "Evaluation of Verification Tools for Knowledge-Based Systems," *International Journal of Human-Computer Studies*, vol. 47, pp. 629-658, 1997.

[11] Stachowitz R., Combs J. and Chang C., "Validation of Knowledge-Based Systems," *in Proceedings of the 2nd AIAA/NASA/USAF Symposium on Automation, Robotics, and Advanced Computing for the National Space Program*, Arlington, VA, 1987.

Nabil Arman received his BS in computer science with high honours from Yarmouk University, Jordan in 1990, his MS in computer science from The American University of Washington DC, USA in 1997, and his PhD from the School of Information Technology and Engineering, George Mason University, Virginia, USA in 2000. Currently, he is an associate professor at Palestine Polytechnic University, Hebron, Palestine. His research interests include database and knowledge-base systems, and algorithms.