# Structural and Syntactic Fault Correction Algorithms in Rule-Based Systems

*Nabil Arman, **Dana Richards and **David Rine
*Department of Electrical and Computer Engineering
Palestine Polytechnic University, Hebron, Palestine
** Department of Computer Science
George Mason University, Fairfax, VA 22030-4444, USA

**Abstract:** *The development of efficient algorithms to correct faults in rule-based systems is very crucial in extending the verification and validation of rule sets and in the development of rule-based systems. While it is important to detect various kinds of faults in rule sets, it is also equally important to provide a user/expert with a set of heuristics that can aid in correcting these faults. In this paper, a set of correction algorithms/heuristics for inconsistency, contradiction, circularity, redundancy, and unreachability faults are presented.*

***Keywords:*** *Rule-Based Systems, Rule Bases Fault Correction*

**Received**: *September 06, 2004* | **Revised**: *January 31, 2005* | **Accepted**: *February 25, 2005*

## 1. Introduction

There is a need to extend the verification and validation of rule sets to include correction and refinement phases. It is common for the rule sets to contain faults; e.g., inconsistency, contradiction/conflict, redundancy/subsumption, circularity, and unreachability. While it is important to detect various kinds of faults in rule sets, it is also equally important to provide a user/expert with a set of heuristics that can aid in correcting these faults.

Traditionally, refinement has been identified as a sub field of Knowledge Acquisition [2]. Early in rule-based system (or KBS) development, knowledge acquisition consists of new knowledge/rules being assembled and integrated into the rule set (KB). However, as the rule set evolves, it is often more appropriate to change existing rules rather than always acquiring new rules. This process of altering rules, and possibly incorporating new rules, is known as rule set refinement.

Rule set refinement and correction respond to the existence of evidence suggesting the need for change to remove the faults from a rule set. In this paper, it is suggested that the faults identified by the fault detection algorithms are suitable triggers for refinement and correction of the rule set. The evidence consists of the effects of faults in a certain rule set. A major objective or goal within refinement and correction of rule sets is precisely: "identifying exactly what should be changed and how". This correction and refinement process will be interacting with the expert/user by providing a set of heuristics of how to correct the rule set to eliminate the detected faults. Thus, automated refinement and correction of rule sets are not considered in this study. Zlatareva indicated that refinement during rule base (KB) development phase is concerned with two tasks [6]:

(a) Rule base revision: The purpose of this task is to correct a functional or structural fault, by making suitable generalization or specialization in the rule base to correct the set of conclusions of the rule base.

(b) Rule base restructuring: The purpose of this task is to eliminate sources of potential performance inefficiencies such as redundant, circular or subsumed rules. This does not affect the conclusions of the rule base, but it may affect the way in which some conclusions are derived.

In this paper, we focus on rule base restructuring rather than rule base revision. In doing so, we devise a set of heuristics that can be used to aid a user/expert in correcting or removing performance inefficiencies. Refinement and correction may include heuristics for changes such as deleting a rule, inserting a rule, deleting a condition or inserting a condition, deleting a conclusion or inserting a conclusion. However, in this research rule deactivation/deletion and addition are used to correct different kinds of faults in rule-based systems.

## 2. Rule Sets Correction and Refinement

The development of efficient algorithms to verify rule-based systems against different kinds of faults within the context of large rule–based systems has attracted

much research due to the important role of rule-based systems in various application domains, including Expert Systems (ESs), Active Database Systems, Information Distribution Systems (IDSs) to name a few. Verification is important to ensure the high quality of rule-based systems and to achieve acceptable levels 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.

Using a graph representation of the rule set, the faults are defined as follows [1, 3, 5]:

1) *Inconsistency Faults:* Inconsistency means that $X$ and $\neg X$ both occur in the same context, i.e., there exists a path from vertex $X$ to its exclusive vertex $\neg X$ or vice versa. In Boolean logic, the law of excluded middle holds, i.e., $X \wedge \neg X$.

2) *Contradiction Fault:* Contradiction happens when two rules conclude different outcomes from the same input, i.e., two paths from vertex $A$ to two exclusive vertices like $V$ and $\neg V$.

3) *Circularity Fault:* Circularity fault occurs if a rule chain that starts with a certain vertex reaches the same vertex.

4) *Redundancy Faults:* Redundancy means that an input reaches the same output along two or more paths in the TDG.

5) *Unreachability Faults:* Vertices which are not reachable from input vertices or vertices that do not reach any output vertices.

Therefore, the issues of fault detection and correction become graph reachability problems where general graph algorithms are used. In section 2.1, refinement and correction criteria are explained. Section 2.2 emphasizes some important issues that should be considered in performing faults corrections. In section 2.3, a metric is defined to guide the correction process and section 2.4 presents the algorithms/heuristics used to perform faults correction.

## 2.1 Refinement and Correction Criteria

Using a graph representation of the rule set [1, 3, 5], we develop a set of heuristics for correcting faults and refining rule sets. These heuristics are domain independent heuristics that suggest plausible refinements to help a domain expert or user to correct the rule faults. Since we are developing heuristics, we do not generally expect the evidence to be so complete to determine whether a heuristic actually corrects the faults. The heuristics are organized by fault categories as follows:

(1) Inconsistency: To correct an inconsistency fault, the path between the two exclusive vertices (e.g., $P$ and $\neg P$) must be broken. This path may have the form:

$$P \rightarrow N_1 \rightarrow N_2 \rightarrow \ldots N_m \rightarrow \neg P \text{ or,}$$
$$\neg P \rightarrow N_1 \rightarrow N_2 \rightarrow \ldots N_m \rightarrow P,$$

where $N_1, \ldots, N_m$ are predicate vertices along the path. At least one of the rules must be deactivated/deleted to correct the fault. However, a new set of rules should be added to preserve the reasoning capabilities of the rule-based system.

(2) Contradiction: To correct a contradiction fault, two paths should be considered. These paths start from one vertex and proceed to two exclusive vertices. These paths may have the form:

$$Q \rightarrow N_{11} \rightarrow N_{12} \rightarrow \ldots N_{1m_1} \rightarrow \neg P$$
$$Q \rightarrow N_{21} \rightarrow N_{22} \rightarrow \ldots N_{2m_2} \rightarrow P$$

In this case, one rule must be deactivated/deleted to correct the fault. However, a new set of rules should be added to preserve the reasoning capabilities of the rule-based system.

(1) Circularity: Circular rule sets may result in a problem in backward chaining systems (non-terminating loop can occur when the rules are fired). Cycles in a rule set may have a path of the form:

$$P \rightarrow N_1 \rightarrow N_2 \rightarrow \ldots N_m \rightarrow P$$

Cycles should be broken to correct the circularity fault pattern. This means that a rule should be deactivated/removed. This should take into account the output vertices (conclusions) and vertices that are used in other rules. In this case, a new set of rules should be added to preserve the reasoning capabilities of the rule-based system.

(2) Redundancy/Subsumption: To correct a redundancy fault, two paths should be considered. These paths have the same starting and ending vertices. These paths may have the form:

$$P_1 : P \rightarrow N_{11} \rightarrow N_{12} \rightarrow \ldots N_{1m_1} \rightarrow Q$$
$$P_2 : P \rightarrow N_{21} \rightarrow N_{22} \rightarrow \ldots N_{2m_2} \rightarrow Q$$

In this case, one rule must be deactivated/deleted to correct the fault. However, a new set of rules should

be added to preserve the reasoning capabilities of the rule-based system.

(3) Unreachability: Unreachability means that there is an internal vertex that is not reachable from any one of the input vertices or there is an internal vertex that does not reach any one of the output vertices in the underlying graph representation. To correct an unreachability fault pattern, a procedure should loop through all unreachable vertices form the input vertices and delete/deactivate all rules having the unreachable vertex in their premises. Next, the procedure loops through all unreachable vertices with respect to the output vertices and deletes/deactivates all rules having the unreachable vertex as a conclusion of those rules.

## 2.2 Issues to be considered

In correcting various kinds of faults, one should maintain the reasoning capabilities of the rule set under consideration. Notice that the correction uses one rule as the smallest entity in the correction process. For unreachability fault correction, one may choose to suggest deleting some predicate vertices if they do not appear in any active rule, but it is believed that rules should be the concern when interacting with the user. Another point to notice is that the correction is based entirely on the structure and syntax of the rules (structural and syntactic correction) and without considering the semantics of the underlying rules.

Rule base fault correction should be guided by error importance to the rule base function. This means that if different kinds of faults have been detected, a knowledge engineer/user specify the order of correction to these faults. In addition to that, the number of generated corrections must be controlled to prevent the combinatorial explosion, since the set of modifications that can potentially correct a fault is very large. If every potential correction was tried, the process would be computationally intractable [4]. Therefore, the correction of faults is implemented in terms of heuristics that are hoped to correct the faults in most cases, but that is not guaranteed in all situations. Similar to the process of detecting faults, faults correction heuristics are formulated as reachability maintaining properties in the graph-based representation of the rule set. Using these criteria, we would like to keep the same reasoning capabilities of the rule set after correcting various kinds of faults. This means that starting from the same input vertices, the rule set should be able to reach the same output vertices (conclusions). Another issue to consider is that fault correction heuristics algorithms should be invoked to correct a real fault (and not a potential fault). Therefore, issues related to rules with multiple antecedents (rule identifiers vertices in the graph-based representation whose in-degree is greater than one) are considered at the fault detection phase and not at the fault correction phase.

## 2.3 Rule Faults Correction Heuristics Metrics

In considering the correction heuristics, the main goal to be accomplished is to get rid of the different kinds of faults (if possible). This is achieved by suggesting the addition of some rules and deleting/deactivating some other rules. A metric can be formulated in terms of the number of rules to be added and the number of rules to be deleted from the rule set. The number of rules to be added is dependent on the out-degree and in-degree of all vertices of the paths between fault pattern vertices. These in/out-degrees can be pre-computed and stored in the direct accessible structure (as extra fields) using Standard Depth First Search Algorithm. Based on this discussion, we can first determine which vertex along a certain path to be point of interest in the correction. This means that we are calculating a metric $M(v)$ for some vertices $v$ on the fault pattern paths. In developing the correction metric and algorithms/heuristics, four functions are assumed to be defined for fault paths. For a path $P: v_1, v_2, v_3, \ldots, v_n$, the functions are:

(1) $Successor_1(v)$, which finds the first successor of vertex $v$ in a specific path. For example, $Successor_1(v_1) = v_2$.

(2) $Successor_2(v)$, which finds the second successor of vertex $v$ in a specific path. For example, $Successor_2(v_1) = v_3$.

(3) $Predecessor_1(v)$, which finds the first predecessor of vertex $v$ in a specific path. For example, $Predecessor_1(v_3) = v_2$.

(4) $Predecessor_2(v)$, which finds the second predecessor of vertex $v$ in a specific path. For example, $Predecessor_2(v_3) = v_1$.

For a non-fault path $P$, where a rule identifier may have an in-degree/out-degree of more than *1*, we define the functions (where $\wedge$ denotes conjunction of propositions):

(1) $P.Predecessor_1(v)$= *The set of all immediate predecessor vertices of v.*

(2) $P.Predecessor_2(v) = \wedge P.Predecessor_1(P.Predecessor_1(v))$

(3) $P.Successor_1(v) = Successor_1(v)$ (*The immediate successor vertex of v in P*)

(4) $P.\ Successor_2(v) = Successor_2(v)$
(5) $P.Conjunct(v) = \wedge(P.Predecessor_1(P.\ Successor_1(v)) - \{v\})$

Let $u$ be the vertex under consideration (predicate vertex and not a rule identifier), and A($u$) be number of rules to be added at vertex $u$. Let $D(u)$ be number of rules to be deleted at vertex $u$, then $M(u)=A(u) + D(u)$, where $M(u)$ is the metric that determines which vertex to be the focus of the correction. Notice that different types of patterns require the metrics to be computed differently, as explained below:

(1) *Inconsistency*

An inconsistency fault is represented by a path between two exclusive vertices $e_1$ and $e_2$ (e.g., $e_1 = u \rightarrow \ldots \rightarrow e_2 = \neg u$). There is a set of different cases:
a) $e_1$ is an input vertex and $e_2$ is an output vertex. The knowledge engineer/user should review input and output vertices.
b) $u = e_1$, where $e_1$ is the first vertex in the inconsistency fault path and is not an input vertex).
$A(u)=in\text{-}degree(u)$
$D(u)=1$,
Thus for the first vertex $u = e_1$,
$M(u)=in\text{-}degree(u) + 1$
c) $u = e_2$,where $e_2$ is the last vertex in the inconsistency fault path and is not an output vertex.
$A(u)=out\text{-}degree(u)$
$D(u)=1$,
Thus for the last vertex $u = e_2$,
$M(u)=out\text{-}degree(u) + 1$
d) $u = e_1$, where $e_1$ is an input vertex. $e_1$ should not be considered in the correction.
e) $u = e_2$, where $e_2$ is an output vertex. $e_2$ should not be considered in the correction.
f) $u = w$, where $w$ is any vertex on the path, except the first and last vertex in the inconsistency fault pattern.
$A(u)=out\text{-}degree(e_2)+\sum out\text{-}degree(v)-1$,*for all v in* $Successor_2(u) \rightarrow Predecessor_2(e_2)$
$D(u)=1$
$M(u)=out\text{-}degree(e_2)+\sum out\text{-}degree(v)-1$,*for all v in* $Successor_2(u) \rightarrow Predecessor_2(e_2)+1$ or,
$A(u)=in\text{-}degree(e_1)+\sum in\text{-}degree(v)-1$,*for all v in* $Successor_2(e_1) \rightarrow Predecessor_2(u)$
$D(u)=1$
$M(u)=in\text{-}degree(e_2)+\sum in\text{-}degree(v)-1$,*for all v in* $Successor_2(e) \rightarrow Predecessor_2(u)+1$
Thus, for all vertices other than $e_1$ and $e_2$,
$M(u)=out\text{-}degree(e_2)+\sum out\text{-}degree(v)-1$,*for all v in* $Successor_2(u) \rightarrow Predecessor_2(e_2)+1$ or,
$M(u)=in\text{-}degree(e_2)+\sum in\text{-}degree(v)-1$,*for all v in* $Successor_2(e) \rightarrow Predecessor_2(u)+1$
and if $e_1$ is not an input vertex
$M(u)=in\text{-}degree(u) + 1$
and if $e_2$ is not an output vertex
$M(u)=out\text{-}degree(u) + 1$

Thus, the vertex to consider is vertex $u$ that belongs to the inconsistency fault pattern path, which has the minimum $M(u)$. It is only sufficient to consider the first and last vertices (exclusive vertices) in the correction, since $M(u)$ for the general case is bounded by $M(u)$ for $e_1$ or $M(u)$ for $e_2$.

*(2) Contradiction*

A contradiction fault is represented by paths between a common ancestor $w$ to two exclusive vertices $e_1$ and $e_2$ (e.g., $w \rightarrow \ldots \rightarrow e_1 = v$; $w \rightarrow \ldots \rightarrow e_2 = \neg v$). There is a set of different cases:
a) $e_1$ and $e_2$ are output vertices. The knowledge engineer/user should review output vertices.
b) $e_1$ or $e_2$ is not an output vertex; assume that it is $u$.
$A(u)= out\text{-}degree(u)$, where $u$ can be either $e_1$ or $e_2$.
$D(u)=1$, therefore,
$M(u)=out\text{-}degree(u) + 1$, where $u$ can be $e_1$ or $e_2$.
For all vertices $u$ on the path $w \rightarrow \ldots \rightarrow e_1$, other than $w$ and $e_1$
$M(u)=out\text{-}degree(u)+\sum out\text{-}degree(v)-1$, *for all v in* $Successor_2(u) \rightarrow Predecessor_2(e_1)+1$
For all vertices $u$ on the path $w \rightarrow \ldots \rightarrow e_2$ other than $w$ and $e_2$.
$M(u)=out\text{-}degree(u)+\sum out\text{-}degree(v)-1$, *for all v in* $Successor_2(u) \rightarrow Predecessor_2(e_2)+1$

Thus, the vertex to consider is vertex $u$ that belongs to the contradiction fault pattern paths, which has the minimum $M(u)$. Notice that it is sufficient to consider the exclusive vertices in the correction, since $M(u)$ for the general case is bounded by $M(u)$ for $e_1$ or $e_2$.

*(3) Circularity*

A circularity fault is represented by a path starting and ending at the same vertex. (e.g., $u \rightarrow \ldots \rightarrow u$). If the selected cycle vertex is $v$, then for all vertices $u$ in the circularity fault pattern path,
$A(u)=$ *Number of output vertices reachable from v - Number of output vertices reachable from v after breaking the cycle at u*
$D(u)=1$
$M(u)=$ *Number of output vertices reachable from v - Number of output vertices reachable from v after breaking the cycle at u* $+ 1$

Thus, the vertex to consider is vertex *u* that belongs to the circularity fault pattern path with minimum $M(u)$

*(4) Redundancy*

A redundancy fault is represented by two paths between two vertices. (e.g., $a \rightarrow \ldots P_1 \ldots \rightarrow v_1 \rightarrow b$; $a \rightarrow \ldots P_2 \ldots \rightarrow v_2 \rightarrow b$). We consider the two paths $P_1$ and $P_2$, and compute the metrics:

$M(P_1) = \sum(\textit{out-degree}(v)-1), \textit{for all } v \textit{ in } Successor_2(a) \ldots P_1 \rightarrow Predecessor_2(b) + 1$

$M(P_2) = \sum(\textit{out-degree}(v)-1), \textit{for all } v \textit{ in } Successor_2(a) \ldots P_2 \rightarrow Predecessor_2(b) + 1$

We choose the path $P_1$ or $P_2$, whichever has the minimum value of the metric.

*(5) Unreachablitiy*

For all unreachable vertices from input vertices, deactivate all rules starting from these vertices. This means deactivating all rules that have the unreachable vertices in their premises. For all unreachable vertices with respect to output vertices, deactivate all rules that have the unreachable vertices in their conclusion.

### 2.3.1 Rule Set Representation

In this paper, rule sets are transformed into a transition directed graph (TDG), where propositions and rule identifier are represented by vertices, and edges represent the transitions from propostions to rule identifiers or rule identifiers to propositions [1, 3]. For example: Rule set R is represented in TDG as in Figure 1.

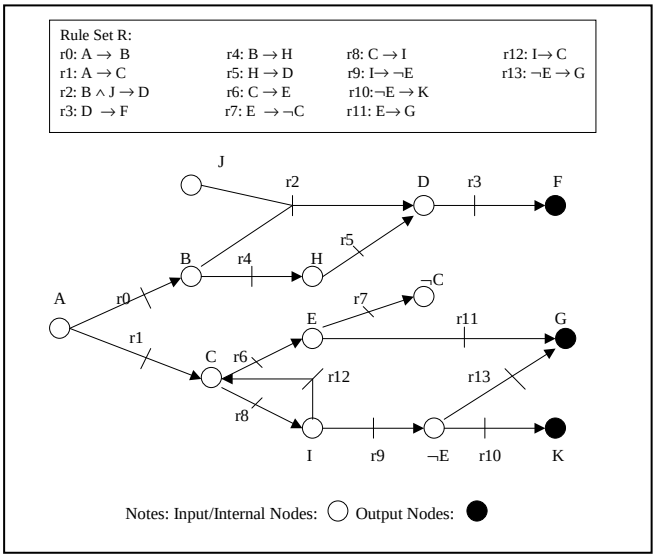


**Figure 1**. Transition Directed Graph (TDG)

## 2.4 Rule-Based Faults Correction Algorithms/Heuristics

*(1) Inconsistency Correction Heuristics*

To correct an inconsistency fault, the exclusive vertices of the pattern must be known (determined by an inconsistency fault detection algorithm or entered by the user). Once these exclusive vertices are known, the correction algorithm can be invoked. The heuristic is represented by Inconsistency_Fault_Correction Algorithm (Figure 2), which can be invoked with the exclusive vertices and the pattern identifier as the input. It starts by checking the first exclusive vertex $e_1$ for input. If the first exclusive vertex $e_1$ is not an input vertex, the algorithm computes the metric $M(e_1)$ ($M(e_1)=\textit{in-degree}(e_1) + 1$). If the second exclusive vertex $e_2$ is not an output vertex, the algorithm also computes the metric $M(e_2)$($M(e_2)= \textit{out-degree}(e_2) + 1$. The algorithm then determines the vertex $e_1$ or $e_2$ with the minimum value of the metric, and this vertex becomes the focus of correction. Once the vertex is determined, the algorithm suggests some modification to the rule set to correct the fault. If the first exclusive vertex is not an input vertex and it has the minimum value of the metric, then the procedure determines the first rule (a sub path of three vertices with one that is a rule identifier) that is part of the inconsistency fault pattern and finds all rule chains that include this rule (all paths that contain the sub path). The procedure then suggests adding a set of new rules to the rule set that make all chains pass the first exclusive vertex and reach to the first exclusive vertex successor vertex (not a rule identifier) in the fault pattern. Once this is done the first rule in the inconsistency fault pattern is deleted/deactivated. Notice when adding a rule to maintain reachability, all conjuncts of the unreachable rule are included in the new rule. If the second exclusive vertex is not an output vertex and it has the minimum value of the metric, then the procedure determines the last rule (a sub path of three vertices with one that is a rule identifier) that is part of the inconsistency fault pattern and finds all rule chains that start with this rule conclusion (all paths that start with the conclusion of the rule). The procedure then suggests adding a set of new rules to the rule set that make all chains pass the second exclusive vertex and reach to the exclusive vertex successor vertices (not rule identifiers). Notice when adding a rule to maintain reachability, all conjuncts of the unreachable rule, except the exclusive vertex, are included in the new rule. Once this is done the last rule in the inconsistency fault pattern is deleted/deactivated. If the first exclusive vertex is an input vertex, then it is not considered and the

correction is performed at the second exclusive vertex. If the second exclusive vertex is an output vertex, then the correction is performed by considering the first exclusive vertex. If the first exclusive vertex is an input and the second exclusive vertex is an output vertex, then the set of input, output, and exclusive vertices should be reviewed by the user. Notice that considering any other vertex other than the first and second exclusive vertices results in a larger value of the metric and should not be considered. The main cases are shown in Figure 3.

```
Procedure Inconsistency_Fault_Correction_Heuristics
    (ex_vertex1,ex_vertex2,pattern_id)
{

 // Inconsistency Fault Detection Algorithm must be
      executed before this procedure!!!

 if(Input_Vertex(ex_vertex1)==1 &&
                    Output_Vertex(ex_vertex2)==1)
          {
          print Heuristic: Input and Output vertices
          can not be exclusive vertices;
          print Please, review Input and Output
          vertices to correct the fault;
 }
 else {
          M(ex_vertex1)=in-degree(ex_vertex1) + 1;
          M(ex_vertex2)=out-degree(ex_vertex2) + 1;
          min_m_vertex= vertex ex_vertex1 or ex_vertex2
           with the minimum metric value(M);
          if((Input_Vertex(ex_vertex1)!=1 &&
              min_m_vertex==ex_vertex1) ||
              Output_Vertex(ex_vertex2)==1) {
              //Determine last two predecessors of ex_vertex1
                of all paths containing ex_vertex1
             //Suggested Rules for Addition
              for all paths p ending with ex_vertex1  do
                    print p.Predecessor2(ex_vertex1) –
                    Rule  ID→Successor2(ex_vertex1);
             // Suggested Rule for Deletion
               print ex_vertex1—Successor1(ex_vertex1)
               →Successor2(ex_vertex1);
          }
          else {
               //Determine first two successors of
               ex_vertex2 of all paths containing ex_vertex2
               //Suggested Rules for Addition
               for all paths s starting with  ex_vertex2 do
                     print Predecessor2(ex_vertex2)
                     ^s.Conjunct(ex_vertex2) –Rule ID→
                     s.Successor2(ex_vertex2);

                    // Suggested Rule for Deletion

                    print Predecessor2(ex_vertex2)—
                     Predecessor1(ex_vertex2)→ ex_vertex2;

          }
 }

}
```

**Figure 2.** Inconsistency_Fault_Correction Algorithm

Example: Consider the TDG representation of the rule set given in Figure 1. In this rule set, there is an inconsistency fault pattern represented by the path $C \rightarrow r6 \rightarrow E \rightarrow r7 \rightarrow \neg C$. The correction algorithm determines that C is not an input vertex and compute the metric $M(C)= in\text{-}degree(C) + 1 = 2 + 1 = 3$. For the second exclusive vertex $\neg C$, the algorithm computer $M(\neg C)=out\text{-}degree(\neg C) + 1=1$. Thus, vertex $\neg C$ is the focus of the correction. The algorithm determines the last rule in the inconsistency fault pattern, which is $E \rightarrow r7 \rightarrow \neg C$. Next, the algorithm determines all chains that start with $\neg C$ by determining all succecessor rules that have $\neg C$ in their premises. and finds that there are no rules/paths. This means that the algorithm does not suggest adding any rules to the rule set.( i.e., no rules are added since there are no vertices reachable from $\neg C$). Once this is done, the algorithm suggests deleting/deactivating the last rule in the inconsistency fault pattern $E \rightarrow r7 \rightarrow \neg C$.

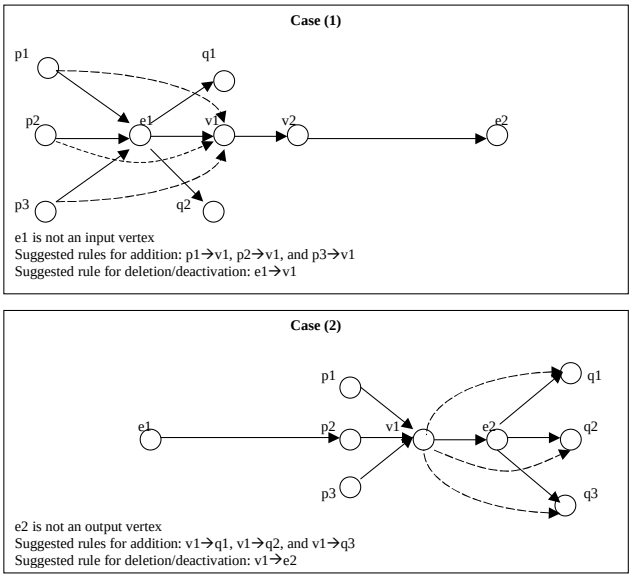


**Figure 3.** Inconsistency Fault Correction Main Cases

*(2) Contradiction Correction Heuristics*

To correct the contradiction fault, the exclusive vertices of the pattern, the common ancestor of the pattern, and the pattern identifier must be known (determined by a contradiction fault detection algorithm or entered by the user). Once these vertices are known, the correction algorithm can be invoked. The heuristic is represented by the procedure Contradiction_Fault_Correction Algorithm (Figure 4), which can be invoked with the exclusive vertices, the common ancestor of the pattern, and the pattern identifier as the input. It starts by checking the exclusive vertices for output vertices. If both exclusive vertices are output vertices, then the algorithm suggest deleting/deactivating either rule starting form the

common ancestor of the contradiction fault pattern, otherwise, the algorithm computes the metric for both exclusive vertices $e_1$ and $e_2$ using the formulas $M(e_1)$=*out-degree*$(e_1)$ *+1* and $M(e_2)$=*out-degree*$(e_2)$ *+ 1*. The algorithm then determines the vertex with the minimum value of the metric (*M*) to be the focus of the correction. Assume that *e* (either $e_1$ or $e_2$) is the vertex with the minimum value of the metric, then the algorithm determines the last rule (a sub path of three vertices with one that is a rule identifier) that is part of the contradiction fault pattern and finds all rule chains that start with this rule conclusion (all paths that start with the conclusion of the rule). The procedure then suggests adding a set of new rules to the rule set that make all chains pass the exclusive vertex *e* and reach to the exclusive vertex successor vertices (not rule identifiers). Notice when adding a rule to maintain reachability that all conjuncts of the unreachable rule, except the exclusive vertex, are included in the new rule. Once this is done the last rule in the contradiction fault pattern is deleted/deactivated. This case is shown in Figure 5.

Example: Consider the TDG representation of the rule set given in Figure 1. In this rule set, there is a contradiction fault pattern represented by the paths $C \rightarrow r6 \rightarrow E$ and $C \rightarrow r8 \rightarrow I \rightarrow r9 \rightarrow \neg E$. The correction algorithm computes the metric $M(E)$= *out-degree*$(E)$ *+ 1 = 2 +1= 3* and the the metric $M(\neg E)$ = *out-degree* $(\neg E)$ *+ 1= 2+ 1 =3*. Since both exclusive vertices have the same value of the metric, the algorithm can choose either one of them. Assume that the algorithm chooses $\neg E$, then the algorithm determines the last rule (a sub path of three vertices with one that is a rule identifier) that is part of the contradiction fault pattern, which is $I \rightarrow r9 \rightarrow \neg E$, and finds all rule chains that start with this rule conclusion $\neg E$ (all paths that start with the conclusion of the rule). The procedure then suggests adding a set of new rules to the rule set that make all chains pass the exclusive vertex $\neg E$ and reach to the exclusive vertex successor vertices (not rule identifiers), which means adding the rules $I \rightarrow$ *RULE ID* $\rightarrow G$ *and* $I \rightarrow$ *RULE ID* $\rightarrow K$. Once this is done the rule $I \rightarrow r9 \rightarrow \neg E$ in the contradiction fault pattern is deleted/deactivated, which breaks the contradiction fault pattern.

*(3) Circularity Correction Heuristics*

To correct a circularity fault, a vertex in the pattern and the pattern identifier must be known (determined by a circularity fault detection algorithm or entered by the user). Once this vertex and the specified pattern are known, the correction algorithm can be invoked. The heuristic is represented by the procedure Circularity_Fault_Correction Algorithm

```
Procedure Contradiction_Fault_Correction_Heuristics
    (ex_vertex1,ex_vertex2,common_ancestor,pattern_id)
{

 // Contradiction Fault Detection Algorithm must be
   executed before this procedure!!!

 if(Output_Vertex(ex_vertex1)==1 &&
         Output_Vertex(ex_vertex2)==1) {
          print  Heuristic: Both exclusive vertices are
                                              output vertices;
          print Please, delete/deactivate either
                 common_ancestor—
                 Successor1(common_ancestor) →
                 Successor2(common_ancestor) through
                 ex_vertex1 or common_ancestor—
                 Successor1(common_ancestor)→
                 Successor2(common_ancestor) through
                 ex_vertex2;
 }
 else {
          M(ex_vertex1) = out-degree(ex_vertex1) + 1;
          M(ex_vertex2) = out-degree(ex_vertex2) + 1;
          ex_vertex= vertex ex_vertex1 or ex_vertex2 with the
          minimum metric value (M);

          //Suggested Rules for Addition

          for all paths s starting with ex_vertex do {
                 // Determine first two successors
                 Successor1(ex_vertex) and
                 Successor2(ex_vertex);
              print Predecessor2(ex_vertex)
                    ^s.Conjunct(ex_vertex) –Rule ID→
                       s.Successor2(ex_vertex);
          }

          // Suggested Rule for Deletion

          print common_ancestor—
          Predecessor1(ex_vertex)→
          Predecessor2(ex_vertex);
 }
}
```

**Figure 4.** Contradiction_Fault_Correction Algorithm

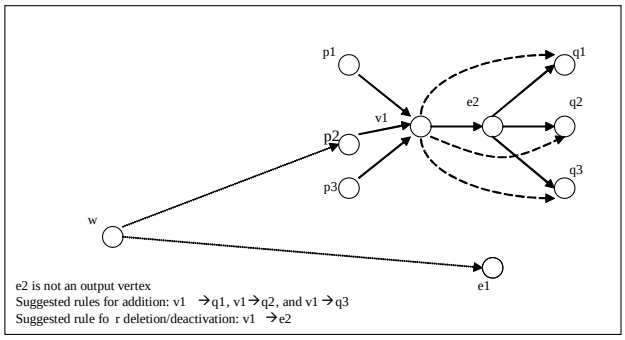


**Figure 5.** Contradiction Fault Correction Case

(Figure 6), which can be invoked with one of the vertices in the circularity fault pattern and the pattern identifier as the input. The algorithm starts by computing the metric for all vertices *u* in the circularity fault pattern path using the formula $M(u)$= *Number of output vertices reachable from u - Number of output vertices reachable from u after breaking the cycle + 1.* The algorithm then determines the rule that starts with *u* (a sub path of three vertices with one that is a rule identifier and the first vertex is *u*) that is part of the circularity fault pattern. The procedure then suggests adding a set of new rules to the rule set that make vertex *u* reach all output vertices originally reachable before the cycle was broken. Once this is done, the first rule in the circularity fault pattern starting with *u* is deleted/deactivated, and this breaks the cycle while maintaining the reasoning capabilities of the rule set. If the output vertex is reachable through a path whose rule identifiers have an in-degree greater than one, then all non-path conjuncts are added to the antecedent of the rule to be added. A simple case is shown in Figure 7.

Example: Consider the TDG representation of the rule set given in Figure 1. In this rule set, there is a circularity fault pattern represented by the path $C \rightarrow r8 \rightarrow I \rightarrow r12 \rightarrow C$. The correction algorithm prompts the user for one of the vertices (e.g., *C*) and the number of the circularity fault pattern, and uses this information to determine the vertex *u* with the minimum $M(u)$ and the rule in the circularity fault pattern which starts with that vertex. In this case, the algorithm computes $M(C)$*=2 - Number of output vertices reachable from C if the cycle is broken at C +1 =2*. It also computes $M(I)$*= 2- Number of output vertices reachable from C if the cycle is broken at I +1 =1,* and determines that I is the focus of the corrrection. Therefore, the first rule in the pattern becomes $I \rightarrow r12 \rightarrow C$. Next, the algorithm determines if there are output vertices that are not reachable after breaking the cycle. Since, in this case, there are no vertices, the algorithm does not suggest adding any new rules. After doing that, the algorithm suggests deleting/deactivating the first rule in the circularity fault pattern (i.e., $I \rightarrow r12 \rightarrow C$), which breaks the circularity fault pattren.

### *(4) Redundancy Correction Heuristics*

To correct a redundancy fault, the two vertices of the pattern that represent the starting and ending of the fault pattern paths and the pattern identifier must be known (determined by the redundancy fault detection algorithm or entered by the user). Once these vertices and the pattern identifier are known, the correction algorithm can be invoked. The heuristic is represented by the procedure Redundancy_Fault_Correction Algorithm (Figure 8), which can be invoked with the two vertices representing the starting and ending of the redundancy fault pattern and the pattern identifier as input. The algorithm starts by computing the metric of the two paths ($P_1$ and $P_2$) representing the redundancy fault pattern using the formula: $M(P_1) = \sum(\textit{out-degree}(v)-1), \textit{for all } v \textit{ in } \textit{Successor}_2(a) \ldots P_1 \rightarrow \textit{Predecessor}_2(b) + 1$

```
Procedure Circularity_Fault_Correction_Heuristics
      (cycle_vertex,pattern_id)
{

  // Circularity Fault Detection Algorithm must be
     executed before this procedure!!!

 for all vertices u in the circularity fault pattern path  do {
            Compute  M(u)= Number of output
            vertices reachable  from cycle_vertex –
            Number of output vertices reachable
            from cycle_vertex after breaking the
            cycle at u + 1;
            Determine the vertex u with minimum M(u);
            min_m_vertex=u;
 }

 //Suggested Rules for Addition

 for all output vertices v reachable from min_m_vertex do {
    if (there is no path from min_m_vertex  to vertex v) then
               print min_m_vertex ^  Conjunct(
               (P.Predecessro1(u) for all rule identifiers u in the
               path Successor2(min_m_vertex to v ) →
               All predicate vertices in the path from
               Successor2(min_m_vertex) to Predecessor2(v))
 }

 // Suggested Rule for Deletion

 // Successor1 and Successor2 are part of the rule in the cycle
containing min_m_vertex

 print min_m_vertexSuccessor1(min_m_vertex) →
                                  Successor2 (min_m_vertex);

}
```

**Figure 6.** Circularity_Fault_Correction Algorithm

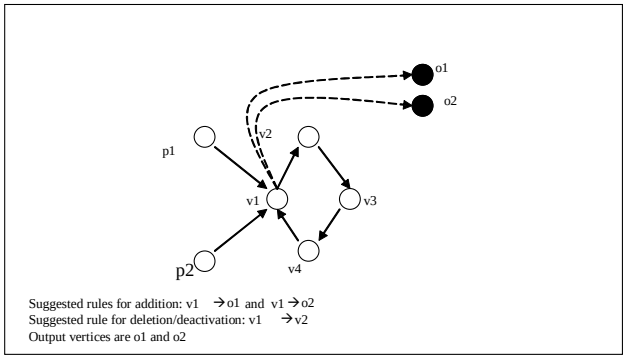


**Figure 7.** Circularity Fault Correction Case

for the first path, and the formula:
$M(P_2) = \sum(out\text{-}degree(v)-1), for\ all\ v\ in\ Successor_2(a)\ldots\ P_2 \rightarrow Predecessor_2(b) + 1$
for the second path. The algorithm chooses the path with the minimum value of the metric. The algorithm then determines the first rule (a sub path of three vertices with one that is a rule identifier and the first vertex is the starting vertex of the redundancy fault pattern) that is part of the redundancy fault pattern path with the minimum value of the metric. Next, the procedure then adds new rules to link the redundancy fault pattern starting vertex to the set of all vertices reachable form the conclusion of the selected rule except those that are on the redundancy fault pattern path or those that are reachable from the ending vertex of the redundancy fault pattern. Notice when adding a rule to maintain reachability that all conjuncts of the unreachable rule, except the redundancy pattern vertex, are included in the new rule. Once this is done the first rule in the redundancy fault pattern path with minimum value of the metric is deleted/deactivated to get rid of the redundancy fault pattern. This case is shown in Figure 9.

Example: Consider the TDG representation of the rule set given in Figure 1. In this rule set, there is a redundancy fault pattern represented by the paths $P_1$: $B \rightarrow r15 \rightarrow D$ and $P_2$: $B \rightarrow r4 \rightarrow H \rightarrow r5 \rightarrow D$. The correction algorithm computes $M(P_1) = 0 + 1 = 1$ and $M(P_2) = 0 + 1 = 1$. Since both paths have the same value of the metric, the algorithm can choose either one. In this case the algorithm chooses the first path $P_1$. It then determines the first rule in the first path of the redundancy fault pattern, which is $B \rightarrow r2 \rightarrow D$. The algorithm then determines all vertices that are reachable from $D$ (before $D$ on the path) and finds that there are no vertices and thus there is no need to add any rule to the rule set. After that, the algorithm suggest deleting/deactivating this first rule, which eleminates the redundancy fault pattern by breaking the first path of the pattern $B \rightarrow r2 \rightarrow D$. There is another redundancy fault pattern represented by the paths $P_1$: $C \rightarrow r6 \rightarrow E \rightarrow r11 \rightarrow G$ and $P_2$: $C \rightarrow r8 \rightarrow I \rightarrow r9 \rightarrow \neg E \rightarrow r10 \rightarrow G$. Again, the algorithm computes $M(P_1) = 1 + 1 = 2$ and $M(P_2) = 2 + 1 = 3$. Since the first path $P_1$ has minimum value of the metric, the algorithm chooses $P_1$ for the corrction. It then determines the first rule in the first path $P_1$ of the pattern, which is $C \rightarrow r6 \rightarrow E$. The algorithm then determines all vertices reachable from $E$ (before $G$ on the path) and finds that there are is only vertex $\neg C$ on the path that is not on the path $P_1$. After that, the algorithm suggest adding the rule $C \rightarrow RULE\ ID \rightarrow \neg C$ to the rule set. Once this is done, the algorithm suggests deleting/deactivating this first rule, which eleminates the redundancy fault pattern by breaking the first path of the redundancy fault pattern $C \rightarrow r6 \rightarrow E$.

```
Procedure Redundancy_Fault_Correction_Heuristics
    (red_vertex1,red_vertex2,pattern_id)
{
  // Redundancy Fault Detection Algorithm must be
   executed before this procedure!!!
 metric1=1;  metric2=1;
 for all vertices v on the first redundancy path P1 do
          metric1 = metric1 + (out-degree(v) –1);
 for all vertices v on the second redundancy path P2 do
          metric2 = metric2 + (out-degree(v) –1);
 if(metric1 <= metric2) {
   //Suggested Rules for Addition
   for all vertices v on the path P1 do
     for all paths s starting with v do {
      Determine first two successors Successor1(v)
       and Successor2(v);
       if(there is no path from s.Successor2(v) to
       red_vertex2 && there is no path from
       red_vertex2 to s.Successor2(v)) then
         print red_vertex1^s.Conjunct(v) –Rule
                                        ID→s.Successor2( v);
         //Suggested Rules for Deletion
         print  red_vertex1—P1.Successor1(red_vertex1)→
                                    P1.Successor2(red_vertex1;
 }
 else {
          // Suggested Rules for Addition

          for all vertices v on the path P2 do
                for all paths s starting with v do {
                     Determine first two successors
                     Successor1(v) and Successor2(v);
                     if(there is no path from s.Successor2(v) to
                     red_vertex2 &&  there is no path from
                     red_vertex2 to s.Successor2(v)) then
                       print red_vertex1^s.Conjunct(v) –Rule ID
                                        →s.Successor2( v);

          }

          // Suggested Rules for Deletion

          print  red_vertex1—P2.Successor1(red_vertex1)→
                                    P2.Successor2.(red_vertex1);
 }
}
```

**Figure 8.** Redundancy_Fault_Correction Algorithm

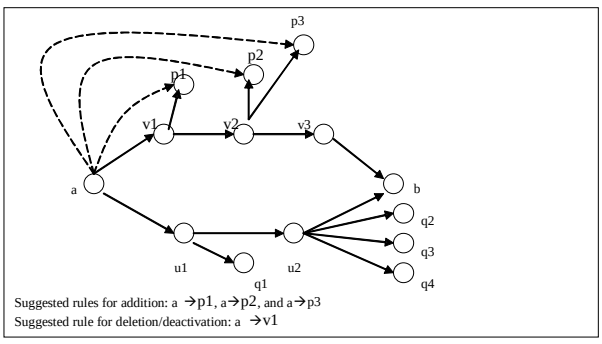


**Figure 9.** Redundancy Fault Correction Case

*(5) Unreachability Correction Heuristics*

To correct unreachability fault patterns, the unreachable vertices of the patterns must be known (determined by the unreachability fault detection algorithm or entered by the user). Once these vertices are known, the correction algorithm can be invoked. The heuristic is represented by the procedure Unreachability_Fault_Correction Algorithm (Figure 10), which can be invoked with the two vertices sets representing the unreachable vertices form input and output vertices. It loops through all unreachable vertices form the input vertices and deletes/deactivates all rules having the unreachable vertex in their premises/conditions. Next, it loops through all unreachable vertices with respect to the output vertices and deletes/deactivates all rules having the unreachable vertex as a conclusion of those rules. Of course, vertices representing rule identifiers are not considered as real vertices for correction purposes.

**Example:** Consider the TDG representation of the rule set given in Figure 1. In this rule set, there is a set of unreachability fault patterns, classified into two categories: vertices that are unreachable from input vertices and vertices that are unreachable with respect to the output vertices. There is one vertex that is unreachable from the input vertices, namely *J*. Since *J* is not a rule identifier, the algorithm suggests deleting/deactivating all rules that include this vertex in their premises/conditions, which means that deleting/deactivating the rule $J \rightarrow r2 \rightarrow D$. There are two unreachable vertices with respect to the output vertices, namely r7 and $\neg C$. There is no need to consider the rule identifies like r7. For the vertex $\neg C$, the algorithm suggests deleting/deactivating all rules that include the predicate vertices in their conclusions, which means that deleting/deactivating the rule $E \rightarrow r7 \rightarrow \neg C$.

## 2.5 Fault Correction Heuristics Algorithms Analysis

Using the handshaking theorem (Given an undirected graph $G=(V,E)$, where $|E|$ is the number of edges and $|V|$ is the number of vertices, then $2.|E| = \sum_{v \in V} \text{in} - \text{degree}(v)$), the complexity of the heuristics can be studied. The heuristics mainly use a procedure *Determine_Predecessor_Paths* to find all predecessor vertices and a procedure *Determine_Successor_Paths* to find all successor vertices of a certain vertex. The predecessor vertices number of a vertex is equal to the in-degree of a vertex and the successor vertices number of a vertex is equal to the out-degree of a vertex.

```
Procedure Unreachability_Fault_Correction_Heuristics()
{
// Unreachability Fault Detection Algorithm must be executed
before this procedure!!!
// Suggested Rules for Deletion
for all unreachable vertices u from input vertices do {
        // Based on Input Vertices
        if( u is not a rule identifier) then {
        // Determine first two successors of u of all paths
        containing u
        for all paths s stating with u do {
          Determine first two successors
                        Successor1(u) and Successor2(u);
          print u –s.Successor1(u)→s.Successor2(u);
         }
        }
}
for all unreachable vertices u with respect to output vertices do {
        // Based on Output Vertices
        if(u is not a rule identifier) then {
         // Determine last two predecessors of u of all paths
         containing u
         for all paths p ending with u do {
                Determine last two predecessors
                Predecessor1(u) and Predecessor2(u);
                print p.Predeccessor1(u)—.Predeccessor2(u)→u;
        }
 }
}
```

**Figure 10.** Unreachability_Fault_Correction Algorithm

The underlying graph of the rule set is a directed graph. Using a variation of the handshaking algorithm, the formula to be used is $2.|E| = \sum_{v \in V} \text{in} - \text{degree}(v) + \sum_{v \in V} \text{out} - \text{degree}(v)$, where each edge is counted twice in the undirected graph and counted once in the in-degree sum and once in the out-degree sum in the directed graph.

Since $\sum_{v \in V} \text{in} - \text{degree}(v) = \sum_{v \in V} \text{out} - \text{degree}(v)$ (an edge has two directions), then

$2. \quad |E| = 2. \sum_{v \in V} \text{in} - \text{degree}(v)$, and thus $|E| = \sum_{v \in V} \text{in} - \text{degree}(v)$, and

$2. \quad |E| = 2. \sum_{v \in V} \text{out} - \text{degree}(v)$, and thus $|E| = \sum_{v \in V} \text{out} - \text{degree}(v)$.

Let $AVG(\textit{in-degree}(v))$ be the average in-degree for the vertices in the graph, then

$|E| = |V| \cdot AVG(\textit{in-degree}(v))$, and thus $AVG(\textit{in-degree}(v)) = |E|/|V|$

Let *AVG*(*out-degree*(*v*)) be the average out-degree for the vertices in the graph, then
$|E| = |V| \cdot AVG(out\text{-}degree(v))$, and thus $AVG(out\text{-}degree(v)) = |E|/|V|$

Since we assume that the underlying graph of the rule set is sparse, $O(|E|)=O(|V|)$, then
$AVG(in\text{-}degree(v))=|E|/|V|= constant$, and $AVG(out\text{-}degree(v))= |E|/|V|= constant$

Based on this analysis, then the average computational complexity of the procedure *Determine_Pred_Paths* is $O(1)$, since $AVG(in\text{-}degree(v))$ is a constant and the average computational complexity of the procedure *Determine_Succ_Paths* is $O(1)$, since $AVG(out\text{-}degree(v))$ is a constant.

(*1*) *Inconsistency_Fault_Correction Procedure*
The most dominant operation in this procedure is determining all successors/predecessors of a certain vertex, which is constant on the average. Thus, the average-case computational complexity is $O(1)$.

(*2*) *Contradiction_Fault_Correction Procedure*
The most dominant operation in this procedure is determining all successors/predecessors of a certain vertex, which is constant on the average. Thus, the average-case computational complexity is $O(1)$.

(*3*) *Circularity_Fault_Correction Procedure*
The most dominant operation in this procedure is the Check_Directed_Path_Existence, which has a worst-case computational complexity of $O(m)$. Thus, the procedure has an average-case complexity of $O(m)$, assuming that the number of vertices in the circularity path is a constant.

(*4*) *Redundancy_Fault_Correction Procedure*
The most dominant operation in this procedure is the Check_Directed_Path_Existence, which has a worst-case computational complexity of $O(m)$. Thus, the procedure has an average-case complexity of $O(m)$, assuming that the number of vertices in a redundancy path is a constant.

(*5*) *Unreachability_Fault_Correction Procedure*
The procedure will determine all predecessors and successors of a certain a vertex. This has a complexity of $O(1)+O(1)$. The loop that adds rules to the rule set is iterated for the number of predecessors, and thus has a complexity of $O(1)$. The whole procedure section is iterated for the number of unreachable vertices that are reachable from input vertices, which is assumed to be constant, and thus it still has a complexity of $O(1)$. The same procedure section is iterated for the number of unreachable vertices with respect to the output vertices, which is assumed to be constant, and thus has a complexity of $O(1)$. Therefore, the procedure has a complexity of $O(1)$.

## 3. Conclusion

A set of algorithms that can correct faults in rule-based systems against inconsistency, contradiction, circularity, redundancy/subsumption, and unreachability has been presented. These algorithms are very crucial in extending the verification and validation of rule sets and in the development of rule-based systems.

## 4. References

[1] Arman, N., *Detecting and Correcting Faults in Chained-Inference Constrained Rules in Information Distribution Systems*, Ph.D. Dissertation, School of Information Technology and Engineering, George Mason University, (2000).

[2] Craw, S., Refinement complements verification and validation, *International Journal of Human-Computer Studies*, 44, 245-256 (1996).

[3] Hwaing, Y., *Detecting Faults in Chained-Inference Rules in Information Distribution Systems*, Ph.D. Dissertation, School of Information Technology and Engineering, George Mason University, (1997).

[4] Meseguer, P. and Verdaguer, A., Verification of Multi-Level Rule-Based Expert Systems: Theory and Practice, *International Journal of Expert Systems*, Vol. 6, No. 2, 163-192 (1993).

[5] Nazareth, D. and Kennedy, M., Verification of rule-based knowledge using directed graphs, Knowledge Acquisition, 339-360 (1991).

[6] Zlatareva, N., A refinement framework to support validation and maintenance of knowledge-based systems, *Expert Systems with Applications*, Vol. 5, 245-252 (1998).

**Nabil Arman** received his BS in Computer Science with high honors from Yarmouk University in 1990 and 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 of Computer Science at Palestine Polytechnic University. Dr. Arman is interested in Database and Knowledge-Base Systems, and Algorithms.

**Dana Richards** received the Certificate of Intermediate Honors and the M.S. from the *University of Virginia* in 1976 and the Ph.D. from the *University of Illinois* in 1984, all in computer science. He has been a programmer for Comress, Inc., and Assistant Professor of Computer Science at the University of Virginia, and most recently a Program Director of Theory of Computing at the National Science Foundation. He has written or edited three books and numerous journal articles. In addition, he is a reviewer for many journals, and he has received numerous research awards. His research interest include *comparisons of protein sequences, steimer tree algorithms, information dissemination in networks, parallel heuristics and methodology for computationally intractable problems, and parallel algorithms for median filters* . Dr. Richards joined the faculty of George Mason University in the Fall of 1994 as an Associate Professor of Computer Science.

**David Rine** has been practicing software development, computational sciences and software systems engineering for thirty-three years. He joined George Mason University in June 1985 and was the founding chair of The Department of Computer Science, as well as co-developer of the School of Information Technology and Engineering. He is presently Professor of Computer Science, Professor of Information and Software Engineering, and Professor on the Faculty of the Institute for Computational Sciences and Informatics. He has been researching, teaching, consulting, working with the software industry and directing research projects in the areas of software systems engineering, computational science, information systems, computer science and science and engineering education. Within the span of his career in computing he has published over one hundred - seventy papers in the general areas of computer science, engineering, information systems, computer applications, computational science, science and engineering education, systems engineering and software engineering. Dr. Rine is internationally known for his work in science and engineering education, having accumulated many years of experience in directing curriculum, large scale software, computational science and systems projects.