Foundation and Application of Knowledge Base Verification

Alun D. Preece*

Laboratoire d'Intelligence Artificielle (LIA)

Université de Savoie—ESIGEC

2, route de Chambery, F-73376 Le Bourget du Lac cedex, France

Email: preece@univ-savoie.fr

Rajjan Shinghal

Department of Computer Science

Concordia University

1455 de Maisonneuve Boulevard West, Montréal, Québec, Canada H3G 1M8

Email: shinghal@cs.concordia.ca

*Please address all correspondence to Alun Preece at the address given.

Abstract

Anomalies such as redundant, contradictory and deficient knowledge in a knowledge base are symptoms of probable errors. Detecting anomalies is a well-established method for verifying knowledge-based systems. Although many tools have been developed to perform anomaly detection, several important issues have been neglected, especially the theoretical foundations and computational limitations of anomaly detection methods, and analyses of the utility of such tools in practical use. This article addresses these issues by presenting a theoretical foundation of anomaly detection methods, and by presenting empirical results obtained in applying one anomaly detection tool to perform verification on five real-world knowledge based systems. The techniques presented apply specifically to verifying rule-based knowledge bases without numerical certainty measures.

Acknowledgments

The work described in this article was supported by funding from Bell Canada. This article is an extended version of papers presented at (1) the Workshop on Verification and Validation of Knowledge-based Systems held during the *Tenth National Conference on Artificial Intelligence (AAAI-92)*, San Jose, California, July 1992, and (2) the *Tenth European Conference on Artificial Intelligence (ECAI-92)*, Vienna, Austria, August 1992 (proceedings published by John Wiley and Sons Ltd., Chichester, U.K.).

Knowledge Base Verification

The importance of assuring the reliability of knowledge-based systems (KBS) is now widely recognized. It is also widely recognized that the process of *verification* is an important part of reliability assurance for these systems. Although definitions of KBS verification vary in the literature, one common theme is that verification of a knowledge base (KB) includes checking the knowledge for logical *anomalies* such as redundant knowledge, contradictory knowledge, and missing knowledge. Such verification is called *anomaly detection.*

In the decade following the appearance of the first known anomaly-detection program for KBS (Suwa et al., 1982), the field has developed rapidly. Two broad phases of research and development can be identified:

1982–1986 Systems developed during this period detected simple manifestations of redundant knowledge (typically duplicate and subsuming pairs of rules), contradictory knowledge (pairs of rules with equivalent conditions but contradictory conclusions), and missing rules (logical combinations of permissible input not matching the conditions of an existing rule in the KB). Pioneering systems included the RCP (Suwa et al., 1982) and CHECK (Nguyen et al., 1985); systems addressing efficiency issues included ESC (Cragun and Steudel, 1987) and Puuronen's algorithm (Puuronen, 1987).

1987–1991 Work in this period increased the power of anomaly detection systems by extending the definitions of anomalies beyond simple pairs of rules: redundancies and contradictions arising over chains of rules could be detected, as could more subtle cases of missing knowledge. Significant systems included COVADIS (Rousset, 1988), EVA (Stachowitz et al., 1987), and KB-Reducer (Ginsberg, 1988), with efficiency issues being addressed by tools such as COVER (Preece, 1989) and SACCO (Laurent and Ayel, 1989).

The field of verification has until now focussed upon building novel anomaly detection systems and improving the efficiency of existing systems. While these are important pursuits, some other critical issues have remained largely unaddressed:

- What are the theoretical foundations of KBS verification by anomaly detection, and how can we be sure that such methods are reliable?

- What are the theoretical limitations of anomaly detection methods, and how do they impact upon the use of the methods in practice?

- How useful are these methods in practice? More specifically, is the cost of detecting the anomalies repaid by the possible improvement in the KB?

This article is an "experience report" on anomaly detection in two ways: first, we draw upon the past ten years of experience to establish a theoretical framework in first order logic by which anomaly detection systems can be analyzed; then, we study the application of a sample anomaly detection tool called COVER to a number of real-world KBS, highlighting the actual performance and utility experienced from using the tool. To give the reader a flavour of the performance of COVER, the appendix to this article lists examples of many of the anomaly types, taken from our sample of real-world KBS (listing examples of *every* anomaly would have made this article overly long). The techniques presented in this article apply specifically to the verification of rule-based knowledge bases which do not use numerical certainty measures.

Verification Principles

The following commonly-accepted assumptions about KBS verification are made in this article:

1. Both the syntax (form) and semantics (meaning) of anomalies must be defined in terms of the syntax and semantics of the knowledge representation language used to express the KB.

2. Anomalies are defined in terms of the declarative meaning of the KB, rather than any procedural meaning.

3. Anomalies are detected by examining the *syntax* of a KB, although they may be understood semantically.

4. Anomaly detection methods apply only to the *knowledge base* of a KBS—certain properties of the inference engine are assumed but not verified.

5. Anomalies are not *errors*: they are symptoms of probable errors in a KB.

The first principle states that syntactic and semantic definitions of anomalies are representation language-dependent; they are defined usually in first-order logic or, when no variables or functions are needed, in propositional logic.

The second principle states that anomaly detection is concerned with the static meaning of the KB, rather than any dynamic meaning conferred by the use of an inference engine. This implies that, in some cases, it will be necessary to separate procedural (task) and declarative (domain) knowledge in order to perform anomaly detection. In such cases, the best approach might be to perform anomaly detection at the level of a *conceptual model* of the KB, rather than at the level of an implemented KB. The reason for this is chiefly that the pragmatics of an implementation may force the implementor to mix procedural and declarative knowledge that are conceptually separate (Preece, 1990), thereby complicating anomaly detection.

The third principle means that, for example, where there are two identical rules in a KB, one of them is clearly redundant semantically, but this can be detected only by finding two rules that are syntactically identical.

The fourth principle is generally considered to be a strength of the approach, because it allows for independent verification of inference engine and knowledge base. However, it is necessary to specify those properties of the inference engine upon which the correctness of the results of anomaly detection relies (examples include definitions of the logical inference rules employed by the engine, methods for handling uncertainty, and conflict resolution); moreover, the inference engine should be verified with respect to these properties; an example of this approach was presented by Krause et al. (1990).

The fifth principle makes the distinction between *anomaly* and *error*, which has often been blurred in the literature. For example, in the case of duplicate rules, the anomaly is *redundancy*, and this is a symptom of any of a number of possible errors, including a simple KB editing error (the same rule was entered twice) or a case in which one (or both) of the two rules is incorrect or has some parts missing. Some anomalies may not represent an error at all; for example, a circular chain of inference in a KB to be used with an inference engine that eliminates endless loops. One issue, then, is the utility of detecting certain anomalies, in terms of the likelihood with which they indicate actual errors in a KB.

Anomalies in the Knowledge Base

In this section, the types and subtypes of KB anomalies are specified. In order to do this, it is necessary to specify first the form of the KB in which the anomalies may be present.

Knowledge Base Definitions

We assume the KB to be expressed at the level of a *conceptual model* of the system (Preece, 1990). The conceptual model describes the knowledge base of the expert system in an implementation-independent manner, in terms of real-world entities and relations (Wielinga and Schreiber, 1989). To remove any ambiguity in verification, it is advantageous for the conceptual model to have an explicit semantic foundation. Therefore, for our purposes, we assume the conceptual model to be expressed in a subset of first-order logic, as defined below. A familiarity with the basic concepts and notations of first-order logic is assumed (Shinghal, 1992).

Knowledge base For the purposes of this article, a knowledge base $\mathcal{K}$ is defined to be a set of expressions, $\mathcal{K} = \mathcal{R} \cup \mathcal{D}$, where $\mathcal{R}$ is a *rule base* and $\mathcal{D}$ is a *declaration set*, defined below.

Rule base A rule base $\mathcal{R}$ is a set of expressions $\{R_1, \ldots, R_n\}$ called rules, where $R_i = L_{i_1} \wedge \ldots \wedge L_{i_m} \rightarrow M_i$, in which $L_{i_1}, \ldots, L_{i_m}, M_i$ are literals from first-order logic. Within the scope of this section, the rules are restricted to Horn clause form. (Note that the COVER verification tool described later permits a restricted treatment of negated literals in rule antecedents—see the appendix example on circular dependency.) Two functions are defined on a rule $R = L_1 \wedge \ldots \wedge L_m \rightarrow M$:

$$\begin{aligned} antec(R) &= \{L_1, \ldots, L_m\} \\ conseq(R) &= M \end{aligned}$$

Informally, $antec(R)$ supplies the literals from the antecedent of R, and $conseq(R)$ supplies the literal from the consequent of R.

Declaration set A declaration set $\mathcal{D}$ is a set of expressions, $\mathcal{D} = \mathcal{G} \cup \mathcal{L} \cup \mathcal{C}$, where: $\mathcal{G}$ is a set of *goal literals*, $\mathcal{L}$ is a set of *input literals*, and $\mathcal{C}$ is a set of *semantic constraint* expressions.

The goal literals declaration defines the set of all literals that could possibly be output from the knowledge base (depending on the domain, this may be interpreted as a set of "conclusions" or "actions"). The input literals declaration defines the set of literals that constitute all possible input to the knowledge base (depending on the domain, this may be interpreted as a set of "symptoms" or "findings"). Goal literals and input literals are assumed to be ground literals; that is, no variables occur in them. Typically, the set of goal literals and the set of input literals are disjoint.

The semantic constraints declaration is a set of literals $\{L_1, \ldots, L_n\}$, which means that $L_1 \wedge \ldots \wedge L_n$ is an inconsistency (this is equivalent to saying that the indefinite Horn clause $\neg L_1 \vee \ldots \vee \neg L_n$ is a tautology). For example, the semantic constraint $\{\texttt{male}(x), \texttt{pregnant}(x)\}$ says that, for all x, x cannot be both a male and pregnant. The set of such semantic constraints constitutes $\mathcal{C}$, defined above.

Environment An environment is a subset of input literals that does not imply a semantic constraint. More formally, for environment E, $\neg(E \rightarrow C\sigma)$, for all $C \in \mathcal{C}$ and for any substitution σ (where $C\sigma$ denotes the instance of C obtained by carrying out substitution σ in C).

Inferrable hypothesis The set $\mathcal{H}$ of hypotheses in the rule base is defined to be the set of literals in the consequents and their instances: $H \in \mathcal{H}$ iff $(\exists R \in \mathcal{R})(conseq(R)\sigma = H)$.

Informally, a literal H is inferrable from a rule base $\mathcal{R}$ if there is some environment E such that H is a logical consequence of supplying E as input to $\mathcal{R}$. We say that H is an *inferrable hypothesis*. Formally, the predicate `infer` is defined as follows:

$\texttt{infer}(H, \mathcal{R}, E)$ iff $(\mathcal{R} \cup E) \vdash H$

This implicitly assumes that $R \cup E$ is satisfiable.

$\mathcal{I}$ is the set of all inferrable hypotheses, $\mathcal{I} = \{H \mid \exists E(\texttt{infer}(H, \mathcal{R}, E))$. Note that $\mathcal{I} \supseteq \mathcal{G}$. That is, for any rule base, a subset of the inferrable hypotheses will be declared to be the goal literals for the rule-based system; any remaining inferrable hypotheses are merely intermediate conclusions or "sub-goals".

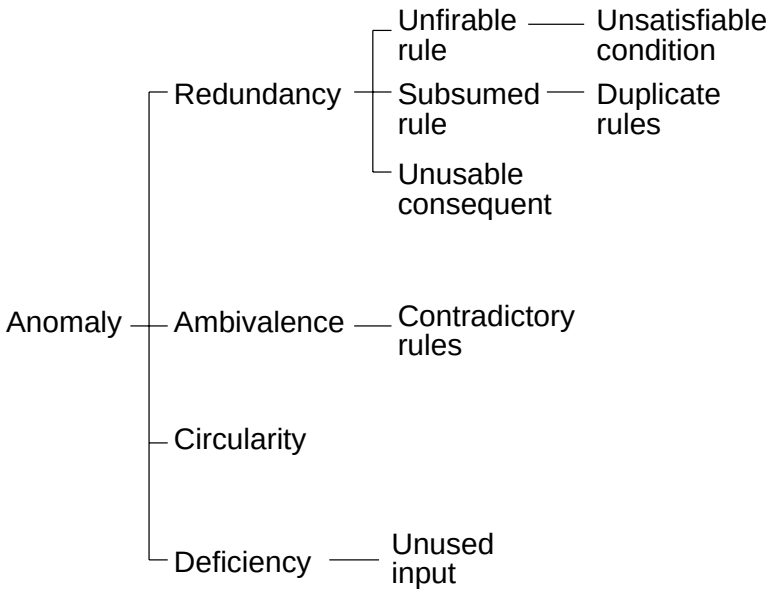

Figure 1: The four types of anomaly and their special cases.

Firable rule Informally, a rule $R \in \mathcal{R}$ is firable if there is some environment E such that the antecedent of R is a logical consequence of supplying E as input to $\mathcal{R}$. Formally:

$\texttt{firable}(R, \mathcal{R}, E)$ iff $(\exists\sigma)(\mathcal{R} \cup E) \vdash antec(R)\sigma$

Anomaly Definitions

We can identify four types of knowledge base anomaly: redundancy, ambivalence, circularity, and deficiency. Each of these types covers a number of special cases. The relationships between the general and special cases of anomalies are shown in the tree of Figure 1. The general types of anomaly are near the root of the tree; anomalies shown deeper in the tree are special cases of the anomalies directly above them. For example, an unused input is a special case of missing rule (deficiency). We discuss each type of anomaly in detail below. Examples of most of these types of anomaly, taken from real expert system applications, appear in the appendix.

Redundancy: Unsatisfiable condition A rule R in a rule base $\mathcal{R}$ contains an unsatisfiable condition if a literal in its antecedent unifies with neither an input literal nor the consequent of another rule in $\mathcal{R}$. Formally, R is redundant if:

$$(\exists L \in antec(R))\neg((\exists I \in \mathcal{L})(L\sigma = I) \vee (\exists R' \in (\mathcal{R} - \{R\}))(L\sigma \in conseq(R')))$$

Redundancy: Unusable consequent A rule R in a rule base $\mathcal{R}$ contains an unusable consequent if its consequent unifies neither with a goal literal nor a literal in the antecedent of another rule in $\mathcal{R}$. Formally, R is redundant if:

$$(\forall G \in \mathcal{G})\neg((conseq(R)\sigma = G) \vee (\exists R' \in (\mathcal{R} - \{R\}))(conseq(R)\sigma \in antec(R')))$$

Redundancy: Subsumed rule A rule R is redundant if another rule R' *subsumes* it; that is, for some substitution σ, we have $R \rightarrow R'\sigma$. Rules R and R' are *duplicates* iff $(R \rightarrow R'\sigma) \wedge (R' \rightarrow R\sigma)$.

Redundancy: Redundant rule This is the most general case of redundancy for a rule: rule R is redundant in rule base $\mathcal{R}$ iff, for every environment, the hypotheses inferred by $\mathcal{R}$ are the same, regardless of the presence or absence of R. Formally, R is redundant if:

$$(\forall E \in \mathcal{E})(\{H \mid \texttt{infer}(H, \mathcal{R}, E)\} = \{H \mid \texttt{infer}(H, \mathcal{R} - \{R\}, E)\})$$

An *unfirable rule* R is a special case of the above:

$\neg(\exists E \in \mathcal{E})$ $\texttt{firable}(R, \mathcal{R}, E)$

The case of an unsatisfiable antecedent condition in R, defined above, is a special case of an unfirable rule. In addition, an unusable consequent in R, and duplication or subsumption of R are special cases of the general case of redundancy.

Ambivalence: Ambivalent rule pair A pair of rules R and R' are ambivalent if the antecedent of R' subsumes the antecedent of R, and their consequents infer a semantic constraint C. Let σ be a substitution. Then formally, R and R' are ambivalent if:

$$(\exists\sigma((antec(R) \rightarrow antec(R')\sigma) \wedge (\{conseq(R), conseq(R')\sigma\} \in \mathcal{C}))$$

Ambivalence: Ambivalent rules A rule base $\mathcal{R}$ contains ambivalent rules if, for some environment E and some substitution σ, all the literals in some semantic constraint are inferrable from the rule base. Formally, $\mathcal{R}$ contains ambivalent rules if:

$$(\exists C \in \mathcal{C}, E \in \mathcal{E})(\forall L \in C\sigma)\ \texttt{infer}(L, \mathcal{R}, E)$$

The case of an ambivalent rule pair, defined above, is a special case of ambivalent rules.

Circularity: Circular dependency A knowledge base contains circular dependency if there is a hypothesis H that unifies with the consequent of a rule R in rule base $\mathcal{R}$, where R is firable only when H is supplied as an input to $\mathcal{R}$. Formally, the knowledge base contains circularity if:
$(\exists R \in \mathcal{R}, \exists E \in \mathcal{E}, \exists H \in \mathcal{H})$

$(H = conseq(R)\sigma \wedge \neg \texttt{firable}(R, \mathcal{R}, E) \wedge \texttt{firable}(R, \mathcal{R}, E \cup \{H\})$

That is, H depends on itself. If H cannot be input since it is not an element of $\mathcal{L}$, then there is no way that H can be inferred.

Deficiency: Unused input A knowledge base has deficiency if it contains a literal I that is declared as an input, but which is unused; that is, it is neither declared as a goal nor unifies with a literal in the antecedent of some rule R in $\mathcal{R}$. Formally, I is unused if:

$$(\exists I \in \mathcal{L}) \neg ((I \in \mathcal{G}) \vee (\exists R \in \mathcal{R})(I \in antec(R)\sigma))$$

Deficiency: Missing rule The knowledge base has deficiency when, for an environment E, the rule base $\mathcal{R}$ will not produce any output. Formally, $\mathcal{R}$ is deficient if, for environment $E \in \mathcal{E}$:

$$(\{G \mid \texttt{infer}(G, \mathcal{R}, E)\} = \emptyset)$$

where $\emptyset$ denotes the empty set. A missing rule is the most general case of deficiency; the case of an unused input is a special case of deficiency.

Anomaly Detection Procedures

This section analyzes the types of KB anomaly along a different dimension to that of the previous section. Whereas the previous section presented a taxonomy of anomaly types under four basic classes (redundancy, ambivalence, circularity, deficiency), this section groups anomalies according to the theoretical computational complexity of the algorithms required to detect them.

Integrity Checks

The following anomalies, as defined above, are detected by an integrity check: unsatisfiable condition, unusable consequent, and unused input. All of these anomalies basically involve simple

"connectivity" of the KB, viewed as a directed graph; detecting them requires an algorithm that checks each antecedent and consequent literal of each rule in the KB against a set of cache tables (for the input set, goal set, and hypothesis set). The theoretical complexity of this algorithm is $O(n)$, for a KB with n rule expressions. One such algorithm is presented by Preece et al. (1992b).

Rule Checks

The following anomalies, as defined above, are detected by a rule check: redundant rule pair, and ambivalent rule pair. Detecting each of these anomalies requires an algorithm that compares each rule in a KB against all other rules; the theoretical complexity of this is $O(n^2)$ for n rules. Various rule-check algorithms are detailed by Cragun and Steudel (1987), Nguyen et al. (1985), Preece et al. (1992b), and Puuronen (1987).

Rule Extension Checks

The following anomalies, as defined above, are detected by a rule extension check: redundant rule, ambivalent rules, circular dependency, and missing rule. These are the most difficult anomalies to detect; detecting them essentially requires that every possible inference chain through the KB be traced and analyzed; in other words, it is necessary first to compute the *extension* of the KB, and then to check the extension for anomalies. By extension, we mean the result of firing every possible chain of rules in the knowledge base. For example, to detect the general case of ambivalence, it is necessary to find a set of rule chains that leads to the inferring of all the literals of a semantic constraint, such that all the chains would arise from some environment. The statement of this anomaly would require reporting to the verification tool user: the inference chains, the environment, and the semantic constraint inferred (the user must then decide whether there is an error in the rules, the environment, or the semantic constraint declaration).

This procedure is very expensive computationally, owing to the need for computing the extension of the knowledge base. Essentially, the method must compute every possible path through the search space defined by the rule base; this will be b^d paths, where b is the *breadth* of

the rule base, defined as the average number of literals in rule antecedents, and d is the *depth* of the rule base, defined as the average number of rules in an inference chain (Preece et al., 1992b). Nevertheless, experiments using implementations of this method show that it is feasible, even for large real-world knowledge bases (Ginsberg, 1988; Preece et al., 1992b). Sample algorithms for the procedure are described by Ginsberg (1988), Preece et al. (1992b), Rousset (1988), and Stachowitz et al. (1987). In the worst case, this method is intractable; for this reason, recent research has proposed methods which make use of heuristics to focus the search for anomalies (Laurent and Ayel, 1989; Preece, 1989). The main idea behind this work is to generate only a partial extension for parts of the knowledge base where anomalies are likely.

Empirical Study

In this section, the results of applying an anomaly detection tool to a sample of real-world KBS are presented and analyzed. The tool used in this study is the COVER verification program, described in detail by Preece et al. (1992b). Two questions are addressed in this study:

- How well does the actual performance obtained using COVER conform to the theoretical complexities of the algorithms employed, and what additional, practical factors affect performance?
- What types of errors are revealed from the anomalies reported by COVER, and, therefore, how useful are the checks performed by the tool?

COVER was used to verify the knowledge bases of five KBS, described in Table 1. Three of the systems are currently in operational use, while the other two (MMU-FDIR and NEURON) were built specifically as demonstration systems (and were verified for that purpose). The five systems exhibit a representative set of applications, from diagnosis to planning, in a variety of domains, and vary in complexity from small (105 rules) to large (550 rules). The table shows also the number of declarations present in each system (that is, the size of $\mathcal{D}$ for each). Note that these were *implementations*, rather than conceptual models. While we would have preferred to verify conceptual models, as discussed earlier, none were available for these systems. Although the five systems cannot be considered a statistically significant or even truly representative

Name	Rules	Declarations	Purpose	Used?	Developed for
MMU-FDIR	105	65	Fault diagnosis	No	NASA
TAPES	150	80	Product selection	Yes	3M Corp
NEURON	190	155	Neurological diagnosis	No	Concordia University
DISPLAN	350	55	Health care planning	Yes	U.K. Health Service
DMS1	550	510	Fault diagnosis	Yes	Bell Canada

Table 1: Description of KBS verified by the COVER anomaly detection tool.

System name	Integrity check	Rule check	Extension check	Breadth/ depth	Total time
MMU-FDIR	$<$ 1 sec.	34 sec.	21 min.	6/1	22 min.
TAPES	$<$ 1 sec.	22 sec.	44 min.	3/1.5	44 min.
NEURON	9 sec.	14 sec.	72 min.	4/2	72 min.
DISPLAN	17 sec.	50 sec.	328 sec.	2/2	395 sec.
DMS1	35 sec.	111 sec.	3.5 hrs.	5/2	3.5 hrs.

Table 2: Approximate real time elapsed using COVER to check five KBS.

sample of KBS, they do provide a useful testbed for a preliminary analysis of anomaly detection by COVER.

Performance Considerations

Timing data obtained from running COVER on the five sample systems appear in Table 2; these data were obtained running on a Sun 4/300 Workstation (times are either to the nearest second or minute, whichever is most appropriate).

The time for the integrity check is approximately linear, in keeping with the theoretical complexity. Interestingly, the COVER program performs a check for circular inference chains in this time, as well as the integrity checks; the circularity check has little effect on performance, because the major computational effort involved is in checking each rule when the inference chains are short (see "depth" in Table 2). The additional effort, which is roughly similar for

each rule, is in building cache tables.

The time for the rule check shows performance of COVER to be better than the theoretical complexity of $O(n^2)$. This is because the main cost of the check lies in comparing the antecedent parts of pairs of rules which have equivalent or contradictory consequents. Determining whether the consequents are equivalent or contradictory is a minor operation, and so the actual cost of the check depends upon the number of rules that have equivalent or contradictory consequents. In the TAPES system, for example, this cost was high, while in the NEURON system this cost was low. As can be seen from the data, the TAPES system, despite its fewer rules, required more computational effort than NEURON for the rule check.

The time for the extension check reflects both the complexity and the variability of this check. There is no correlation between the number of rules or declarations in the system and the cost of this check. Instead, as revealed by the theoretical analysis, it is the breadth and depth of the search space defined by the KB that determine the complexity of the extension check; for this reason, mean values for these ratios for each system are given in Table 2. It should be noted that in no instance did the complexity of this check approach the worst case indicated by the theoretical analysis. Chiefly, this is because the inference chains in the sample knowledge bases are short, and it is clear that, when this is so, the relatively expensive (but exhaustive) extension check is feasible.

Experience and anecdotal evidence suggests that short inference chains may be far more common in practice than long inference chains. If this is true, then intensive research into methods for dramatic improvement of the efficiency of extension checks will be of limited utility, since such improved methods will be of value in verifying only a small number of KBS.

Utility Considerations

Table 3 shows, for each of the five sample systems, the number of each type of anomaly detected by COVER, and the number of each that directly revealed one or more errors in the knowledge base. In all these cases, "completed" knowledge bases were checked by COVER—that is, the systems were considered to have been tested satisfactorily by other means (typically running a large number of actual and contrived test cases on the systems, and evaluating the output produced) and had all been delivered to their users. (Of course, all errors revealed using COVER

Anomaly type	**MMU-FDIR**	**TAPES**	**NEURON**	**DISPLAN**	**DMS1**
Redundancy					
unsatisfiable condition	0	0	15/15	14/14	2/2
unusable consequent	5/5	0	0	4/4	0
redundant rule pair	0	5/5	0	9/4	59/5
redundant rule	10/10	5/5	21/21	45/40	61/7
Ambivalence					
ambivalent rule pair	0	4/4	0	1/1	0
ambivalent rules	0	4/4	0	4/4	10/10
Circularity					
circular dependency	0	0	0	24/20	0
Deficiency					
unused input	0	0	0	28/8	0
missing rule	0	16/16	0	59/17	17/0

Table 3: Numbers of anomalies/errors detected by COVER in five KBS.

were subsequently corrected.)

The most striking result here is that, despite the best efforts of their developers, real-world KBs often contain anomalies, sometimes in surprisingly large numbers. Furthermore, where anomalies are present, they frequently correspond to errors in the KB. The notable exceptions in Table 3 are the following:

- Cases of subsumed (redundant) rule pairs, where a more general rule subsumes a specific rule, and the conflict resolution strategy to be employed by the inference engine would ensure that the more specific rule is fired in preference—therefore, the subsumption is not considered to be an error.

- Cases of circularity which are intended, by the KB designers, to explicitly model looped reasoning;

- Cases of deficiency where the "missing" cases represent "do nothing" situations, and were intentionally not included in the KB.

Therefore, from this study, these three types of anomaly would seem to have the least utility, in terms of their indication of actual errors in the knowledge base. We might say that the reporting of these three types of has the lowest "signal-to-noise" ratio, which is an important factor in considering the utility of detecting the various anomalies.

A second point to note is that the totals for the more general anomalies listed in the table include the totals for the anomalies that they subsume. For example, the redundant rule totals include the unsatisfiable condition, unused consequent, and redundant rule pair totals; similarly, the more general cases of ambivalence and deficiency include the totals for their special cases (that is, the total for ambivalent rules includes that for ambivalent pair, and the total for missing rules includes that for unused inputs).

In view of the above observations, there is evidence to suggest that the most useful types of anomaly are those detected by the integrity check; this assertion is based on the relative number of these anomalies, the high likelihood with which they indicate actual errors, and the low cost of detecting them.

A final point is that the types of anomaly present in a KB seem to be influenced by certain characteristics of the KB. One observation supporting this statement is that those systems with short inference chains appear unlikely to contain many extension-check anomalies (other than those already revealed by integrity and rule checks), because there is less potential for the designer to make subtle errors in formulating chains of rules. A second supporting observation is that KBs written in first-order logic tend to harbour more anomalies than those written in propositional logic, because there is more potential for the designer to make errors when writing rules with variables; in the sample, only DISPLAN made extensive use of variables in first-order logic.

Examples of the actual anomalies and errors uncovered by COVER in the five sample systems appear in the appendix.

Conclusions

Some conclusions arising from this work are stated below.

We do not agree with the claim made by Rushby (1988) that the theoretical foundation of

anomaly detection methods is poorly understood. Although this foundation was unclear in the early literature, it is straightforward to recast the definitions of the early tools in terms of a firmly-specified set of definitions of anomalies and algorithms (Preece et al., 1992a).

Integrity checks inexpensively reveal anomalies in KBs, which have a high likelihood of indicating errors. Many popular commercial KBS tools do not provide built-in integrity checkers (although some do offer such tools as utilities, similar in spirit to the *lint* tool for the C programming language).

Although KB extension checks have been criticized in the past because of their theoretical complexity limitations, there is evidence to suggest that these techniques will often be practical in real-world systems development. In fact, before more research is conducted into methods for improving the efficiency of these techniques, perhaps more experience should be obtained regarding whether such work is really necessary, although it may be desirable.

References

Cragun, B. J. and Steudel, H. J., 1987, A Decision-Table-Based Processor for Checking Completeness and Consistency in Rule-Based Expert Systems, *International Journal of Man-Machine Studies*, 26(5), 633–648.

Ginsberg, A., 1988, Knowledge-Base Reduction: A New Approach to Checking Knowledge Bases for Inconsistency and Redundancy, in *Proc. 7th National Conference on Artificial Intelligence (AAAI 88)*, St Paul, MN, volume 2, pages 585–589.

Krause, P. J., Byers, P., Hajnal, S., and Fox, J., 1990, The Use of Object-Oriented Process Specification for the Verification and Validation of Decision Support Systems, in *ECAI '90 Workshop on Validation, Verification and Test of KBS*, Stockholm, Sweden.

Laurent, J. P. and Ayel, M., 1989, Off-Line Coherence Checking for Knowledge Based Systems, in *IJCAI-89 Workshop on Verification, Validation and Testing of Knowledge-Based Systems*, Detroit.

Nguyen, T. A., Perkins, W. A., Laffey, T. J., and Pecora, D., 1985, Checking an Expert Systems Knowledge Base for Consistency and Completeness, in *Proc. 9th International*

Joint Conference on Artificial Intelligence (IJCAI 85), Los Angeles, volume 1, pages 375–378.

Preece, A. D., 1989, Verification of Rule-Based Expert Systems in Wide Domains, in Shadbolt, N., editor, *Research and Development in Expert Systems VI (Proc. Expert Systems 89)*, pages 66–77. Cambridge University Press.

Preece, A. D., 1990, Verifying Expert Systems Using Conceptual Models, in *Proceedings of 5th Annual Knowledge-Based Software Assistant (KBSA) Conference*, Syracuse, pages 280–289.

Preece, A. D., Shinghal, R., and Batarekh, A., 1992a, Principles and Practice in Verifying Rule-Based Systems, *Knowledge Engineering Review*, 7(2), 115–141.

Preece, A. D., Shinghal, R., and Batarekh, A., 1992b, Verifying Expert Systems: A Logical Framework and a Practical Tool, *Expert Systems with Applications*, 5(3/4), 421–436.

Puuronen, S., 1987, A Tabular Rule-Checking Method, in *Proc. 7th International Workshop on Expert Systems and their Applications*, pages 257–268, Paris-La Défense. Agence Inf.

Rousset, M.-C., 1988, On the Consistency of Knowledge Bases: the COVADIS System, *Computational Intelligence*, 4(2), 166–170.

Rushby, J., 1988, Quality Measures and Assurance for AI Software, NASA Contractor Report CR-4187, SRI International, Menlo Park CA.

Shinghal, R., 1992, *Formal Concepts in Artificial Intelligence: Fundamentals*. Van Nostrand, New York.

Stachowitz, R. A., Combs, J. B., and Chang, C. L., 1987, Validation of Knowledge-Based Systems, in *Proc. 2nd AIAA/NASA/USAF Symposium on Automation, Robotics and Advanced Computing for the National Space Program*, Arlington, VA, pages 1–10. (Report No. AIAA-87-1685.)

Suwa, M., Scott, A. C., and Shortliffe, E. H., 1982, An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System, *AI Magazine*, 3(4), 16–21.

Wielinga, B. and Schreiber, G., 1989, Future Directions in Knowledge Acquisition, in Shadbolt, N., editor, *Research and Development in Expert Systems VI (Proc. Expert Systems 89)*, pages 288–301. Cambridge University Press.

Appendix: Example Anomalies

In this appendix, we examine some illustrative examples of anomalies detected in the real-world KBS listed in the body of this article. For each example, we highlight the anomaly, consider whether it indicates an error and, if so, explain how the error was corrected.

Redundancy Example: Unfirable Rule

This example is taken from the NASA MMU-FDIR system, written using the CLIPS rule language. (CLIPS is described fully in J. Giarratano and G. Riley, *Expert Systems: Principles and Programming*, PWS-Kent, New York, 1989. We are grateful to Chris Culbert of the NASA Lyndon B. Johnson Space Center for making the MMU-FDIR system available to us.) The MMU-FDIR system contained the following rule:

```
(defrule cea-a-gyro-input-roll-pos-6
    (aah off) (gyro on)
    (gyro movement roll pos)
    (side a on)
    (side b on)
    (rhc roll none pitch none yaw none)
    (thc x none y none z none)
    (vda a ?m on)
=>
    (assert (failure cea))
    (assert (suspect a))
    (printout t crlf "aah failed to correct pos roll")
)
```

The problem here lies in the first line of the antecedent: the combination of the conditions `(aah off)` and `(gyro on)` is inconsistent (this knowledge was made available to COVER as a semantic constraint declaration) and, therefore, the rule is unfirable. The error here is that the first condition should read `(aah on)`.

Redundancy Example: Subsumed Rule

This example is taken from the Bell Canada DMS1 system, which was originally written using the Level 5 expert system shell product (Level 5 is a trademark of Information Builders Inc.). The DMS1 system contained the following rules:

```
RULE  321  to check for 1.DGP FAIL A,2,3,4
IF MA_LAMPS IS 1.DGP FAIL A
AND MA_LAMPS IS 2.DGP FAIL B
AND MA_LAMPS IS 3.LINE FAIL A
AND MA_LAMPS IS 4.LINE FAIL B
THEN DONEIT
AND POINTA := "Replace the Digroup A & B
     QPP419,QPP417 Cards at the RCT"
AND POINTB := "Replace the Digroup A & B
     QPP419,QPP418 Cards at the CCT"

RULE  322  to check for 1.DGP FAIL A,2,3
IF MA_LAMPS IS 1.DGP FAIL A
AND MA_LAMPS IS 2.DGP FAIL B
AND MA_LAMPS IS 3.LINE FAIL A
THEN DONEIT
AND POINTA := "Replace the Digroup A & B
     QPP419,QPP417 Cards at the RCT"
AND POINTB := "Replace the Digroup A
     QPP419,QPP418 Cards at the CCT"
```

The intention of the designer in writing these rules was to assign values (via the `:=` operator) to string variables (`POINTA` and `POINTB`) as side-effects of establishing the fact `DONEIT`. This is a common method of forcing procedural operations in backward-chaining systems. This works in Level 5 because the rules are entered into the knowledge base in order of antecedent specificity; more specific rules (such as rule 321) appear before less specific ones (such as rule 322). The conflict resolution strategy employed by Level 5 of selecting rules in order of appearance means that the most specific rule applying in a given case will always be the one that fires. COVER,

however, had no access to this assumption (it was not stated anywhere in the rule base), and so COVER reported the two rules as a subsumed pair.

Although no error results from these subsumed rules, it can be argued that such implied dependencies between rules make understanding and maintaining the system unnecessarily difficult. The problem is that knowledge which could have been explicitly stated in the rule antecedents has instead been 'hidden' in the ordering of the rules. An alternative to the approach used would be to remove the implicit dependency between these rules by making the later rules more specific. For example, after consulting the domain expert, rule 322 above could be modified as follows (note the addition of the fourth condition in the antecedent), which would allow the system to perform correctly regardless of rule ordering.

```
IF MA_LAMPS IS 1.DGP FAIL A
AND MA_LAMPS IS 2.DGP FAIL B
AND MA_LAMPS IS 3.LINE FAIL A
AND NOT MA_LAMPS IS 4.LINE FAIL B
THEN ...
```

The disadvantage of this approach is that it results in more complex rules. For this reason, the system was not modified to remove the subsumption.

Redundancy Example: Redundant Rule

This example is taken from the DISPLAN system, which is written in a rule language based on Prolog. DISPLAN contained the following three rules:

```
if      mentalState hasValue lucid
then    noProblem(mentalState).

if      mentalState hasValue lucid
then    assessmentIrrelevant(mentalState).

if      assessmentIrrelevant(Category)
then    noProblem(Category).
```

The COVER environment checker detected that, in the above three rules, the first rule is redundant, since it is subsumed by a combination of the second and third rules, but the third rule is more general. The first rule states that, if patients are classified as lucid, then they have no problem with their mental state. The second rule states that the assessment of mental state is not relevant to DISPLAN if it is classified as lucid. The third rule states that, for any assessment category, if it is irrelevant to DISPLAN then there can be no problem with the category.

This error occurred because the concept of irrelevant assessments was introduced into the knowledge base via the predicate assessmentIrrelevant, but the knowledge base was not fully modified to incorporate it properly. This example highlights the value of a verification tool during maintenance and expansion of a knowledge base.

Ambivalence Example: Ambivalent Rules

This example is taken from the DISPLAN system, which has the following rules, in a Prolog-based rule language:

```
if      patientHomeless
then    goingHome hasValue no.

if      homeAssessmentCategories includes Category
and     patientHomeless
then    Category hasValue irrelevant.
```

The COVER environment checker detected ambivalence in this case, where the first rule says that a patient will not be going home if he or she is homeless, and the second rule says that, for all home assessment categories, if the patient is homeless then the category is irrelevant.

The ambivalence arises because, by default, the set homeAssessmentCategories includes the element goingHome (that is, goingHome is a home assessment category). This leads to the following instance of the second rule, which is clearly ambivalent with the first rule above (remembering that goingHome is a single-valued parameter):

```
if      homeAssessmentCategories includes goingHome and
        patientHomeless
then    goingHome hasValue irrelevant.
```

Circularity Example: Circular Dependency

This example is taken from the DISPLAN system. The COVER integrity checker detected circularity in the following rules:

```
if    additionalServices includes support(Service, Problem)
then  servicesSupport includes support(Service, Problem).

if    extraSupportNeeded(mobility)
then  additionalServices includes support(occupationalTherapist, mobility).

if    carerCannotSupport(Problem)
and   not servicesSupport includes support(Service, Problem)
then  extraSupportNeeded(Problem).
```

The first rule says that a pair support(Service, Problem), representing the notion that a patient's problem is to be supported by a particular service, is to be added to the set of supporting services servicesSupport if it is a member of the set of additional services additionalServices. The second rule says that an occupational therapist will provide support for mobility if extra support is needed for mobility. The third rule says that extra support is needed for some problem if the patient's "carer" (usually his or her family) cannot support the problem, and the problem is not already supported by some service in the set of supporting services.

To see the circularity more clearly, consider what happens when DISPLAN needs to determine whether the pair support(occupationalTherapist, mobility) is a member of servicesSupport. This leads to the following rule instances:

```
if    additionalServices includes support(occupationalTherapist, mobility)
then  servicesSupport includes support(occupationalTherapist, mobility).

if    extraSupportNeeded(mobility)
then  additionalServices includes support(occupationalTherapist, mobility).

if    carerCannotSupport(mobility) and
      not servicesSupport includes support(Service, mobility)
```

then extraSupportNeeded(mobility).

In trying to prove the condition

not servicesSupport includes support(Service, mobility)

in the third rule as negation by failure, one of the cases DISPLAN will need to test is:

servicesSupport includes support(occupationalTherapist, mobility)

This will lead it to the first rule, and we have a circular inference chain, which was not obvious from the original rules.

Deficiency Example: Missing Rule

This example is taken from the DMS1 system. COVER reports pseudo-conditions for rules which it determines to be logically necessary for the completeness of the knowledge base, but which are absent. One such set of pseudo-conditions appears below.

```
IF FLOW8B_COM_PWR_FUSE IS NO AND
FLOW8B_REPL_FUSE IS YES AND
FLOW8B_VOLTAGES IS NO
THEN ???
```

The notation ??? indicates that COVER believes that a consequent should be supplied for this set of conditions. In actuality, this case does not require a rule to be added to the knowledge base because of the procedural semantics of the rule: establishing the first condition to be true requires that the user investigates whether a fuse has blown; if this is not so (the query is answered NO), then there is no need to investigate the second condition, which involves replacing the fuse! Although COVER has no way of knowing this, it is obvious to the designer that this is not really a missing case.

Missing rules are detected by COVER using a generate-and-test method: the program generates combinations of data items and values and tests to see if the situations indicated by these combinations are covered by rules in the knowledge base. If not, COVER reports the combinations as indicating potentially missing rules. Since this checking procedure will be very lengthy for complex knowledge bases, owing to the number of combinations of data items and values,

the user of COVER is able to specify parameters to control the extent of the search for missing rules.

COVER also makes use of the existing domain knowledge to control the search, enabling it to look for likely missing rules. For example, by using information about which data items appear together in rules or inference chains, COVER can decide if certain data items are likely to appear together in missing rules; these are called *relevant* data items. Then, instead of generating combinations of all data items and values, it can restrict the generation to combinations of the relevant data items and their values. The data items `F6B_FUSE_BLOWN`, `F6B_RESET_FAIL` and `F6B_CT_FUSE_AGAIN` in the above rule were determined to be relevant to one another using this rule-of-thumb.