The Knowledge Engineering Review, Vol 7:2, 1992, 115–141

Principles and practice in verifying rule-based systems*

ALUN D. PREECE, RAJJAN SHINGHAL and AÏDA BATAREKH

Centre for Pattern Recognition and Machine Intelligence, Department of Computer Science, Concordia University, 1455 de Maisonneuve Boulevard West, Montreal, Canada H3G 1M8

Abstract

This paper surveys the verification of expert system knowledge bases by detecting anomalies. Such anomalies are highly indicative of errors in the knowledge base. The paper is in two parts. The first part describes four types of anomaly: redundancy, ambivalence, circularity, and deficiency. We consider rule bases which are based on first-order logic, and explain the anomalies in terms of the syntax and semantics of logic. The second part presents a review of five programs which have been built to detect various subsets of the anomalies. The four anomalies provide a framework for comparing the capabilities of the five tools, and we highlight the strengths and weaknesses of each approach. This paper therefore provides not only a set of underlying principles for performing knowledge base verification through anomaly detection, but also a survey of the state-of-the-art in building practical tools for carrying out such verification. The reader of this paper is expected to be familiar with first-order logic.

1 Introduction

Reliability has emerged as a significant issue in the development of expert systems[1]. Empirical studies (Preece, 1990; Preece et al., 1992) have demonstrated that the knowledge bases of expert systems often contain potential errors which can be revealed by static and dynamic analyses of the logic of the knowledge: such analyses are called *verification*. Although verification cannot replace empirical testing of expert systems, it can make testing more productive by eliminating many errors prior to testing (see Figure 1). Verification therefore plays an important part in assuring the reliability of expert systems, and it is in the interests of all expert system builders to ensure that verification is performed on their systems prior to traditional methods of testing. Verification is closely connected with the knowledge acquisition activity, in that anomalies revealed in the knowledge base will guide the next cycle of knowledge acquisition and knowledge base refinement.

There is now an established body of work in the expert systems literature on the automatic verification of expert system knowledge bases by detecting anomalies such as redundant, conflicting, or missing knowledge. However, few attempts have been made to define formally these types of anomaly (Nazareth, 1989; Stachowitz et al., 1987), and it is therefore difficult to compare objectively the capabilities of the various tools that have been described.

This paper seeks to fill a gap in the literature on verifying expert systems. First, it presents a logical framework for comparing vertification tools by providing definitions of knowledge base anomalies, including redundant, deficient, and conflicting knowledge. Second, the paper uses the framework to survey and analyse five automatic verification programs from the literature. In particular, we consider *what* the programs verify, in terms of our logical framework, and *how* they verify, in terms of descriptions and analyses of algorithms employed. We hope that this paper will be useful to expert system researchers and practitioners in the following ways:

*This work was funded by Bell Canada.

[1] In this paper, we consider the terms "expert system" and "knowledge-based system" to be synonymous.

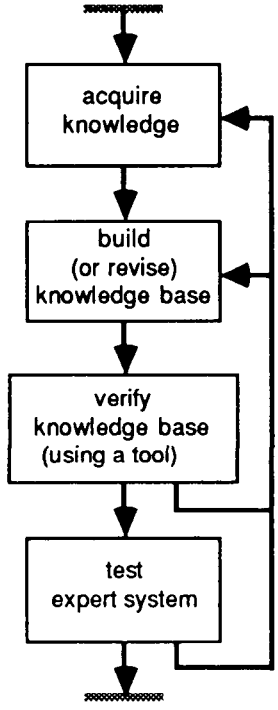

Figure 1 Flowchart for expert system development cycle emphasizing verification task

- As an analysis of the logical foundations of verifying expert systems.
- As a comparison of the functionality of five prominent verification tools.
- As a starting point for the development or selection of new verification algorithms and tools.
- As a basis for further surveys and comparisons of new verification tools.

The paper is organized as follows: section 2 presents the anomaly definitions; section 3 presents the survey of the five verification programs; section 4 summarizes and discusses the survey. In the remainder of this introduction, we briefly expand upon the context and scope of the paper.

1.1 Logical foundations of expert system verification

We choose to focus upon rule-based expert systems which are based on first-order logic because rule-based systems are the most widely-used type of expert system, and the semantics of rule-based systems based on logic are well-defined. Note that our approach can be generalized to apply to frame-based and semantic net types of expert system by re-expressing those formalisms in terms of first-order logic (Preece & Shinghal, 1991). Furthermore, we restrict this paper to discussion of knowledge bases that do not contain certainty factors.

In this paper, we consider *verification* to mean checking a rule base for anomalies such as redundant or deficient knowledge. We formally define four types of anomaly, based on the semantics of first-order logic. There are four basic types of anomaly, some of which have a number of special cases, summarized in Figure 2. Previous definitions of knowledge base anomalies have often been rather *ad hoc*, making it difficult to determine exactly what certain verification tools actually do. Therefore, our definitions of the anomalies in section 2 are presented formally to remove any ambiguity.

In each case, the intuitive meaning of the anomaly corresponds to its formal definition. Our use of first-order logic will enable us to detect each type of anomaly by considering only the syntactic form of expressions in the knowledge base. For example, consider the two rules $L_1 \wedge L_2 \rightarrow M$ and $L_1 \rightarrow M$. From an understanding of the semantics of the operators $\wedge$ (logical "and") and $\rightarrow$ ("implies"), the first rule is intuitively redundant, since it is just a specific version of the second.

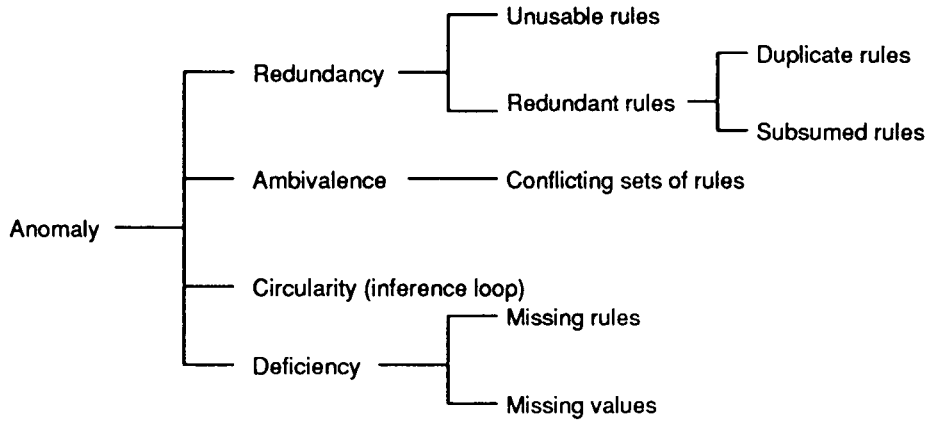

Figure 2 Four types of anomaly and their special cases

However, we can detect this by looking for that particular case of redundancy in which we have two rules with identical consequents (M in this case), such that the literals in the antecedent of one rule are a subset of the literals in the antecedent of the other (in our example, $\{L_1\} \subset \{L_1,L_2\}$).

It is important to realize that such anomalies may not represent actual *errors* in a rule base. Rather, they are *symptoms* of probable errors. For example, a redundant rule may occur because there is missing knowledge (for example, perhaps the rule $L_1 \rightarrow M$ above should have been $L_1 \wedge L_3 \rightarrow M$), in which case the symptom is redundancy but the cause is missing knowledge. It is up to the knowledge base designer to examine detected anomalies and decide on what action to take, based on their causes. Our objective in identifying and formalizing rule base anomalies is to provide a well-defined basis for developing automatic procedures for detecting them, since humans cannot be expected to perform such checking reliably on large rule bases.

1.2 Survey of verification tools

After defining the anomalies, we present a survey and analysis of five systems which have been built for verifying knowledge-based systems by anomaly detection. We base our discussions of these systems on the formal framework laid down in the first part of the paper. The five systems discussed are the Rule Checker Program (Suwa et al., 1982), developed at Stanford University in 1982, the CHECK program (Nguyen et al., 1985), developed at Lockheed Corporation in 1985, the KB-Reducer program (Ginsberg, 1988), developed at AT&T Bell Laboratories in 1987, the EVA program (Stachowitz et al., 1987), under development at Lockheed, and the COVER tool (Preece, 1989), under development within our own group. We describe the anomalies checked by each system, and the algorithms used to perform the checking. We analyse the performance of the algorithms, and describe any subsequent refinements to the original systems.

Given the advancing state of expert system verification practice, and to prevent this paper from becoming too long, our survey of tools is necessarily incomplete, and it is based on depth rather than breadth. Our primary motivation in choosing these particular five systems is that the systems provide a good cross section of verification work done over the last decade. The earlier tools, RCP and CHECK, provide the simplest anomaly checks (in general, these are also the cheapest computationally), as well as being the most frequently cited work in the field. The KB-Reducer tool was the first published system to extend significantly the range of anomalies detected (with a corresponding increase in computational complexity). The EVA project is the most extensive known industrial expert system verification effort to date, and the most comprehensive in terms of tools for both verification *and* testing (although we consider only the verification tools in this paper). Finally, we include our own tool, COVER, because it was the first known verification system to tackle the problem of detecting missing knowledge in a computationally realistic way.

In the conclusion to this paper, we return to our original set of anomalies, and summarize the extent to which the five systems detect the anomalies.

2 Logical rule base anomalies

In this paper, we assume that the reader is familiar with the basic concepts and notations of first-order logic. If not, then Mendelson (1979) provides a good introduction. Our notation for first-order logic is defined in the following section. More detailed definitions of these anomalies are given in Preece et al. (1992).

2.1 Definitions and assumptions

We assume that we are given a domain of discourse, the elements of which are the *terms* (denoted by *r, s, t*) formed from the *function symbols* (*f, g, h*) and *constants* (*a, b, c, 1, 2, 3. . .*). *Variables* (*u, v, w, x, y, z*) are symbols which can take on values from the domain of discourse. *Functions* map elements of the domain to elements of the domain. *Predicates* (*P, Q, R, S, T, U*) map elements of the domain to truth values *True* or *False*. A *literal* is an expression of the form $P(t_1,t_2, \ldots, t_m)$ or $\neg P(t_1,t_2, \ldots, t_m)$.

For real-world examples, predicate names are written in upper case `TYPEWRITER` font, while function names are written in lower case `typewriter` font; within expressions, variables start with a lower case letter, while constants start with an upper case letter (both are written in *italics*). Note that in considering terms in rule expressions, we are currently considering only a single level of embedding of function symbols so that, for example, the term $f(g(x))$ is not considered, while $f(x)$ is.

A rule base consists of a set of logical *production rules*, which we usually refer to simply as *rules*. These are of the form:

$$L_1 \wedge L_2 \wedge \ldots \wedge L_n \rightarrow M$$

where the *L*s and the *M* are literals. If the antecendent $L_1 \wedge L_2 \wedge \ldots \wedge L_n$ of a rule is true, the rule *fires*, and we infer the consequent *M*. We consider rules to have only one literal in the consequent. In this paper, when we refer to a *fact*, we mean a rule with $n = 0$. A literal *M* can be inferred from a knowledge base *K*, denoted by $K \vdash M$, if *M* is a logical consequence of *K*. Similarly, $K \vdash \{M_1, \ldots, M_k\}$ if $K \vdash M_i$ for each $i = 1, \ldots, k$.

Variables will often appear in rules, and all such variables will be assumed to be universally quantified. For example, the statement "all persons are mortal" might be expressed by the following rule:

$$\texttt{PERSON}(x) \rightarrow \texttt{MORTAL}(x)$$

This rule will usually fire due to `PERSON`(x) being true for a specific instance of x. For example, the fact `PERSON`(*Marvin*) being true, together with the above rule, would lead to the conclusion: `MORTAL`(*Marvin*). A *ground term* is a term not containing any variables. Similarly a *ground literal* is a literal not containing variables. For example, `PERSON`(*Marvin*) is a ground literal. A rule is *ground* if all of its literals are ground.

There are some constraints which cannot be expressed in the first-order order language that we have presented. For example, the arguments of predicates usually have specified limited types and ranges. The predicate `AGE_OF`(*person, age*) meaning "the single individual, *person* is *age* years old" would have the argument types *single alphabetic value* for *person* (denoting a name, such as *Marvin*), and *single non-negative integer value* for *age*. Ranges may be specified for some arguments. The range of values for *age* above could be, say, $0 \leq age \leq 120$. Similarly, the single alphabetic value argument x in `SIZE`(x) could be specified to be one of {*Small, Medium, Large*}.

These specified types and ranges serve two purposes: they are used to establish the acceptability of variable substitutions (see below); and they are used in checking the *syntax* of the knowledge base.

A *substitution* σ is a finite set of ordered pairs of the form $\{t_1/v_1, \ldots, t_n/v_n\}$, where each v_i is a variable, each t_i is a term distinct from v_i, and the variables $v_1, \ldots, v_n$ are distinct (Mendelson, 1979). A substitution is *acceptable* if it satisfies the type constraints and the ranges of values imposed on the arguments of the functions and predicates used in the knowledge base. For the rest of the paper, whenever we talk about a substitution we mean an acceptable substitution. In applying a substitution to an expression, we replace each occurrence of v_i in the expression by t_i. The expression obtained after applying the substitution is said to be an *instance* of the expression to which the substitution was applied. Let E be a finite set of expressions. A substitution σ is a *unifer* for E if $E\sigma$, the set obtained by applying σ to each expression in E, is a singleton. For example, the substitution $\{Marvin/x\}$ is a unifer for the literals PERSON(*Marvin*) and PERSON(x).

Literals that appear in the consequents of rules are called *hypotheses*. Hypotheses that appear only in the consequents of rules are called *final hypotheses*. Hypotheses that appear in the consequents of some rules and the antecedents of others are called *intermediate hypotheses*. It is assumed that there are explicit declarations of all final hypotheses—intermediate hypotheses do not need to be declared.

A literal is called a *finding* if it appears only in the antecedents of rules, and never in the consequents. This means that its value is always to be provided by the knowledge base user. A literal is declared to be *askable* if the knowledge base user is permitted to supply a value for it. Askable literals may be a superset of the findings: some intermediate hypotheses may be askable, although final hypotheses usually will not be, since it is the purpose of the knowledge base to determine these. An *environment* is a set of askable literals supplied to a knowledge base by the user of the knowledge-based system; for example, literals representing a person's height, weight symptoms, etc.

A set of literals is *impermissible* if it has been declared as such, usually because it does not make semantic sense. For example, a set of literals containing the facts that a person is male and pregnant is impermissible. For any literal L, the set $\{L, \neg L\}$ is impermissible. Any superset of an impermissible set also is impermissible. An environment or a set of hypotheses is *permissible* iff it does not contain an impermissible set. By definition, the empty (or null) set $\{\}$ is permissible. If E is an impermissible set, then for any substitution σ, the instance $E\sigma$ also is an impermissible set. For example, if the set {MALE(x), PREGNANT(x)} is impermissible, then its instance {MALE(*Marvin*), PREGNANT(*Marvin*)} also is impermissible. Impermissible sets need to be provided by domain experts, as part of the knowledge acquisition process. Note, however, that they need not all be provided in advance: often, the results from one phase of verification on an expert system will reveal the need for additional impermissible sets to be specified, for example to remove the possibility of contradictory combinations of inputs.

Suppose $L_1 \wedge L_2 \wedge \ldots \wedge L_n \rightarrow M$ is a rule in a knowledge base K. The rule is said to be *fireable* if there exists a situation in which an instance $L_1\sigma \wedge L_2\sigma \wedge \ldots \wedge L_n\sigma$ of the rule's antecedent is true. In other words, there exists a permissible environment E such that $K \cup E \vdash L_1\sigma \wedge L_2\sigma \wedge \ldots \wedge L_n\sigma$. Note that the substitution σ can be empty. A rule is *unfireable* iff it is not fireable. A literal is *inferrable* iff it is in the consequent of a fireable rule. A literal is *obtainable* iff it is either askable or inferrable. A set of literals is obtainable iff all its members are obtainable. A literal or set of literals is *unobtainable* iff it is not obtainable. In this paper, we assume that inference is *monotonic* in character: if hypothesis g is inferrable from environment E, then g is inferrable from $E' \supset E$.

In the following sections, we define four types of anomaly: *redundancy, ambivalence, circularity,* and *deficiency*,

2.2 Redundancy

An expression in a knowledge base is *redundant* iff for every permissible environment, the final hypotheses inferred are the same, regardless of the presence or absence of the expression.

2.2.1 Trivial case: unusable rules

We define two trivial cases of redundancy: *unfireable rules*, and *rules with unusable consequents*. Both are defined below.

Unfireable rule According to our definition of an unfireable rule in section 2.1, a rule $L_1 \wedge \ldots \wedge L_n \rightarrow M$ is unfireable iff it has no fireable instance. In other words, for every substitution σ, either $\{L_1, \ldots, L_n\}\sigma$ contains an impermissible set, or $\{L_1, \ldots, L_n\}\sigma$ is unobtainable. For example, assuming that {UNDERGRAD(x), GRADSTUDENT(x)} is an impermissible set, the following rule is an unfireable rule:

$$\texttt{UNDERGRAD}(x) \wedge \texttt{GRADSTUDENT}(x) \rightarrow \texttt{STUDENT}(x)$$

Rule with unusable consequent This occurs when a literal appears only in the consequent of a rule, thus appearing to be a final hypothesis, but it is not declared as such. An example of this would be a rule with a typographical error in the consequent. In the following rule, PROFESOR has been entered, instead of PROFESSOR:

$$\texttt{TEACHER}(x) \wedge \texttt{TENURED}(x) \rightarrow \texttt{PROFESOR}(x)$$

Here, we assume that PROFESOR(x) has not been declared to be a final hypothesis, and that PROFESOR(x) does not occur in the antecedent of any rule (that is, it is not an intermediate hypothesis).

2.2.2 General case: redundant rule

For a knowledge base with n rules, $I_1 \ldots I_n$, rule I_j is redundant if it is a logical consequence of rules $I_1 \ldots I_{j-1}, I_{j+1} \ldots I_n$. As an example of this, consider:

$$\begin{aligned}
\texttt{PROFESSOR}(x) &\rightarrow \texttt{HAS_DEGREE}(x, \textit{PhD}) \\
\texttt{HAS_DEGREE}(x, \textit{PhD}) &\rightarrow \texttt{HAS_DEGREE}(x, \textit{BSc}) \\
\texttt{HAS_DEGREE}(x, \textit{BSc}) &\rightarrow \texttt{GRADUATE}(x) \\
\texttt{PROFESSOR}(x) &\rightarrow \texttt{GRADUATE}(x)
\end{aligned}$$

The last rule above is redundant, since it can be inferred from the first three rules. We may view it as an inference shortcut, possibly included for reasons of run-time efficiency. Note that such shortcuts may be desirable in practice, but the designer should be aware of their existence, so that if at some time the knowledge base needs to be modified, then the knowledge base modifications can be made throughout. Thus, modifying the first three rules may require modifying the last rule also.

Subsumed rules are a special case of the general case described above. Consider two rules:

$$L_1 \wedge \ldots \wedge L_n \rightarrow L$$
$$M_1 \wedge \ldots \wedge M_j \rightarrow M$$

such that there exists a substitution σ, where $L = M\sigma$ and:

$$L_1 \wedge \ldots \wedge L_n \vdash (M_1 \wedge \ldots \wedge M_j)\sigma$$

We say that the first rule *is subsumed by* the second (or that the second rule *subsumes* the first). Since the second rule is more general than the first, the first rule is redundant:

$$\begin{aligned}
\texttt{PROFESSOR}(\textit{Marvin}) \wedge \texttt{TENURED}(\textit{Marvin}) &\rightarrow \texttt{FACULTY}(\textit{Marvin}) \\
\texttt{PROFESSOR}(x) &\rightarrow \texttt{FACULTY}(x)
\end{aligned}$$

The first rule is subsumed by the second rule under substitution $\sigma = \{\textit{Marvin}/x\}$. Note that *duplicate rules* are a special case of subsumed rules, where σ can be taken to be the empty substitution.

Recall that redundancy is an anomaly, rather than an error. In some rule-based systems, subsumed rules are used as "default" rules: where there is a pair of subsumed rules, the inference

engine will select the more specific rule first, and if this rule fails then the inference engine will try the more general rule as a default. Often the inference engine makes this selection on the basis of rule ordering, with the more specific rules appearing in the knowledge base before the general rules. In such cases, even though the redundancy is intentional, we argue that a verification tool should still warn the rule base designer of the redundancy, because the rule-ordering assumption may be implicit and may not hold if the knowledge base is used with an inference engine with a different rule-ordering assumption.

2.3 Ambivalence

Assume we are given the impermissible set $\{N_1, \ldots, N_n\}$, and the following rules where $k \leq n$:

$$\begin{aligned} L_{1,1} \wedge \ldots \wedge L_{1,j_1} &\rightarrow M_1 \\ L_{2,1} \wedge \ldots \wedge L_{2,j_2} &\rightarrow M_2 \\ &\vdots \\ L_{k,1} \wedge \ldots \wedge L_{k,j_k} &\rightarrow M_k \end{aligned}$$

Assume there exists a unifer σ which unifies each pair of literals in the following sets: $\{M_1,N_1\}, \ldots, \{M_k,N_k\}$. Assume that the set of literals:

$$\{L_{1,1}, \ldots, L_{1,j_1}, \ldots, L_{k,1}, \ldots, L_{k,j_k}, N_{k+1}, \ldots, N_n\}\sigma$$

is obtainable. Then it would be possible to infer from the knowledge base the impermissible set $\{N_1, \ldots, N_n\}\sigma$. Hence the knowledge base is ambivalent. For example, assume E = {GRADUATE(x), UNDERGRAD(x)} is an impermissible set, and we have the following rules:

$$\begin{aligned} \texttt{ENROLLED\_IN}(x,y) \wedge \texttt{GRAD\_COURSE}(x) &\rightarrow \texttt{GRADSTUDENT}(y) \\ \texttt{ENROLLED\_IN}(x,y) \wedge \texttt{UNDERGRAD\_COURSE}(x) &\rightarrow \texttt{UNDERGRAD}(y) \\ \texttt{GRAD\_COURSE}(CS661) & \\ \texttt{UNDERGRAD\_COURSE}(CS445) & \end{aligned}$$

If the environment {ENROLLED_IN(*CS*661,*Marvin*),ENROLLED_IN(*CS*445,*Marvin*)} is a permissible environment, we would be able to infer the set:

{GRADSTUDENT(*Marvin*),UNDERGRAD(*Marvin*)}

This is the impermissible set $E\sigma$ where $\sigma = \{Marvin/x\}$.

2.4 Circularity

A knowledge base has *circularity* iff it contains some set of rules such that an interminable loop could occur when the rules are fired. For example:

$$\begin{aligned} L_1 \wedge L_2 &\rightarrow L_3 \\ L_3 &\rightarrow L_4 \\ L_4 &\rightarrow L_1 \end{aligned}$$

Formally, a set of rules can cause an interminable loop if they have the following form:

$$\begin{aligned} L_{1,1} \wedge \ldots \wedge L_{1,j_1} &\rightarrow M_1 \\ L_{2,1} \wedge \ldots \wedge L_{2,j_2} &\rightarrow M_2 \\ &\vdots \\ L_{n,1} \wedge \ldots \wedge L_{n,j_n} &\rightarrow M_n \end{aligned}$$

and there exists a substitution σ such that $M_i\sigma \in \{L_{i+1,1}, \ldots, L_{i+1,j_{i+1}}\}\sigma$ for all $i = 1, \ldots, n-1$, and $M_n\sigma \in \{L_{1,1}, \ldots, L_{1,j_1}\}\sigma$, and the set $\{L_{1,1}, \ldots, L_{1,j_1}, \ldots, L_{n,1}, \ldots, L_{n,j_n}\}\sigma$ is obtainable. A common example of this problem would be a relation explicitly defined in both directions:

$$\texttt{HAS\_DEGREE}(x,y) \rightarrow \texttt{GRADUATE}(x)$$

$$\texttt{GRADUATE}(x) \rightarrow \texttt{HAS\_DEGREE}(x,y)$$

This is another anomaly which will often not indicate any error. Many knowledge bases make use of the kind of mutual implication in the above example. In such cases, the inference engine will simply check to see if it is entering a loop. However, such knowledge structures are useful in practice only if the potential loops can be "broken" by askable literals. In the above example, if the circularity is not an error, then it must be possible to provide both instances of HAS_DEGREE(x,y) and GRADUATE(x) as input to the knowledge base; otherwise, one or both of the rules will be useless in practice.

2.5 *Deficiency*

A knowledge base is *deficient* iff there exists a permissible environment for which no final hypothesis is inferred. This can happen under two circumstances: if the inference engine enters an interminable loop, or if there is missing knowledge. In this section we consider the case of a knowledge base with missing knowledge (we assume that the check for circularity, as described in section 2.4, will have been performed prior to the check for missing knowledge). For example, consider the following base, where STATUS(x,y) is the final hypothesis:

$$\begin{aligned}
\texttt{UNIV_MEMBER}(x) \wedge \texttt{ENROLLED}(x) &\rightarrow \texttt{STUDENT}(x) \\
\texttt{STUDENT}(x) \wedge \texttt{HAS_BSc_DEGREE}(x) &\rightarrow \texttt{STATUS}(x,\textit{GraduateStudent}) \\
\texttt{STUDENT}(x) \wedge \neg\texttt{HAS_BSc_DEGREE}(x) &\rightarrow \texttt{STATUS}(x,\textit{Undergraduate}) \\
\texttt{UNIV_MEMBER}(x) \wedge \texttt{HAS_BSc_DEGREE}(x) & \\
\wedge \neg\texttt{ENROLLED}(x) &\rightarrow \texttt{STATUS}(x,\textit{Staff})
\end{aligned}$$

The literals UNIV_MEMBER(x), HAS_BSc_DEGREE(x), and ENROLLED(x) are findings, and STUDENT(x) is an intermediate hypothesis. Assuming {ENROLLED(x),$\neg$UNIV_MEMBER(x)} is an impermissible set, we do not infer any hypotheses from the following permissible environments:

{UNIV_MEMBER(x), $\neg$ENROLLED(x), $\neg$HAS_BSc_DEGREE(x)}
{$\neg$UNIV_MEMBER(x),$\neg$ENROLLED(x),HAS_BSc_DEGREE(x)}
{$\neg$UNIV_MEMBER(x),$\neg$ENROLLED(x), $\neg$HAS_BSc_DEGREE(x)}

A common sympton of deficiency in a knowledge base is the presence of missing values, described below.

Special case: missing values
Assume we have a set of rules:

$$\begin{aligned}
P(x_1, \ldots, x_n, t_1) &\rightarrow M_1 \\
P(x_1, \ldots, x_n, t_2) &\rightarrow M_2 \\
&\vdots \\
P(x_1, \ldots, x_n, t_n) &\rightarrow M_n
\end{aligned}$$

and there is a value a in the range of values allowed for t_i such that for all substitutions σ and for all $i = 1, \ldots, n$, we have $P(x_1, \ldots, x_n, a) \neq P(x_1, \ldots, x_n, t_i)\sigma$. Then the above set of rules does not

cover all the possible values for the arguments of predicate P. Consider the following knowledge base, in which x in the literal SIZE(x) can have one of the values {*Small*, *Medium*, *Large*}:

SIZE(*Small*) → COST(*Low*)
SIZE(*Large*) → COST(*High*)

No rule will fire when SIZE(*Medium*) is true, thereby suggesting that there is deficiency in the knowledge base.

3 Analyses of five verification tools

Having described above four types of anomaly that may be present in knowledge bases, we will now use these anomalies as a framework for comparing the capabilities of five expert system verification tools. Sections 3.1–5 present descriptions and critiques of five programs which were built between 1982 and 1989 to perform anomaly detection in knowledge bases. In describing each system we attempt to make clear the differences in terminology between the original papers and our "cleaned-up" definitions. When an original term conflicts with our own terminology, we quote the original term so as to distinguish it from our own definition. When we describe the anomalies defined for each system, we present equivalent terminology from our own framework. This makes it possible to compare the features of the different tools more easily than by using the terminology given in the original papers (which is often applied inconsistently between systems).

3.1 The Rule Checker Program

The Rule Checker Program (RCP) (Suwa et al., 1982) was developed at Stanford University in 1982 as part of the ONCOCIN project. The RCP was used successfully to check the ONCOCIN knowledge base, and detected a number of anomalies. Many of the fundamental features of knowledge base verification tools first appeared in the RCP, namely: the ability to check a partly-complete knowledge base; the ability to check knowledge bases on the basis of their syntax only; the identification of domain-independent knowledge base anomalies; and the ability to check a knowledge base in the absence of an inference engine (the method of inference is assumed to be logical deduction).

Although the RCP is not restricted to the domain of ONCOCIN, it embodies certain assumptions about the knowledge base, namely that there are no certainty factors, reasoning is monotonic, and rules conclude a given value for a given item of data (called a *parameter*). Each rule is also assigned to a particular *context*, which tells the inference engine when to try to fire the rule.

3.1.1 What does the RCP check?

The first definitions, albeit informal ones, of the fundamental rule base anomalies appeared in the work on the RCP. The original paper defines limited checks for duplication (called "redundancy") and subsumption between pairs of rules which conclude the same value for a parameter in the same context. These are special cases of redundancy as we have defined it—where inference chains are restricted to single rules. For example:

HAS_VALUE($a1$,$b1$) $\wedge$ HAS_VALUE($a2$,$b2$) → HAS_VALUE($a3$,$b3$)

HAS_VALUE($a1$,$b1$) → HAS_VALUE($a3$,$b3$)

Note that we mimic the ONCOCIN rule form, by using the predicate HAS_VALUE($a1$,$b1$) which can be taken to mean that a parameter named $a1$ has the value $b1$ assigned to it[2]. The RCP check for ambivalence (called "conflict") is restricted to the case of pairs of rules which have subsumed

[2]Note that this assignment of a value to a parameter is not the same as an instantiation of a logical variable: $a1$ and $b1$ are both constants, representing the *names* of a parameter and a value, respectively.

Table 1 Example of the table-based rule format used by the RCP to detect anomalies. Here, the following rule pairs are in conflict: 1 and 2, 1 and 3, 1 and 4, 3 and 7. The pair 2 and 3 are subsumed. There are no cases of duplication, and one missing rule (included for comparison as rule 8)

Rule	*UniversityMember*	*Enrolled*	*HasBScDegree*	*Status*
1	Yes	Yes	—	Student
2	Yes	Yes	No	Undergraduate
3	—	Yes	No	Undergraduate
4	Yes	Yes	Yes	GraduateStudent
5	Yes	No	Yes	Staff
6	No	No	—	NonAcademic
7	No	Yes	—	Impermissible
8	Yes	No	No	?

antecedents, and which have consequents of the form HAS_VALUE(x,y) and HAS_VALUE(x,z), where $y \neq z$, where x is the name of some parameter, and y and z are the names of values for it. The RCP also detects missing rules, but the actual case which is checked for also is rather limited, as we describe below.

3.1.2 Implementation of RCP

The RCP builds a table, similar to a decision table, for each parameter which appears in a rule consequent in each context. Each row in the table represents a rule; each column in the table represents a parameter which appears in one of the rules; each entry in the table represents the value held by the parameter in the rule. By convention, the rightmost column indicates the rule consequent, and the other columns contain the antecedent. If a parameter does not appear in a particular rule, then the corresponding table entry is marked as "don't care" (—). An example RCP table appears in Table 1. The rules in this table are reformulations from the original rule syntax. For example, the original form of the rule contained in the first row of the table is as follows:

$$\text{HAS\_VALUE}(\textit{UniversityMember}, \textit{Yes}) \wedge \text{HAS\_VALUE}(\textit{Enrolled}, \textit{Yes}) \rightarrow \text{HAS\_VALUE}(\textit{Status}, \textit{Student})$$

The implicit "context" of all the rules in this table is assigning some value to the parameter *Status*. Note that, in Table 1, rule 7 describes an impermissible case, which could be declared as an impermissible set (section 2.1). Rules 1 to 7 are rules present in the knowledge base, while rule 8 is not present in the knowledge base—it is a missing rule, detected by the deficiency checking procedure, described below. Once the RCP has created each table, it can easily check the rules for anomalies. In fact, for a small rule set such as in the example, it is easy to check the table manually (although this would not be realistic for a large table).

The RCP procedure can check the rules for redundancy and ambivalence in $n(n-1)/2$ rule comparisons, where n is the number of rules in the table. The deficiency checking procedure is more complex: the RCP checks to see if every possible permutation of the parameters and values is covered by some rule in the table. To exhaustively check every case, the RCP needs to generate p^m combinations, where p is the number of parameters in the table, and m is the number of values each parameter can take. Depending on the presence of "don't cares" in the rules, each rule may cover any number of combinations from one (no "don't cares") to p^m (all "don't cares").

It is unclear how this checking procedure was implemented in the original RCP program, although an algorithm employing cardinal numbers was published by Puuronen (Puuronen, 1987). Regardless of implementation, however, this method of checking becomes intractable when p and m are large, and it is the explicit modularity of the knowledge, as defined by the rule contexts, that makes this approach feasible for checking the ONCOCIN knowledge base.

If this method were used to check a knowledge base in which no contexts were provided by the designer, the system could still use the consequent parameters to group sets of rules together (such an approach is used by the ESC tool, described in Cragun & Steudel, 1987). However, this may not always reduce the size of the tables, since many classification knowledge bases are structured such that a large number of rules have the same parameter in the consequents. Our own group has two knowledge bases in which over 90% of the rules (about 150 and 500 rules, respectively) conclude different values for the same parameter, and would share the same context.

3.1.3 Limitations of RCP

Aside from the computational problems associated with the missing rules detection procedure, the main limitation of the RCP is the narrow range of anomalies it checks for. As we have described, it uses a very restricted treatment of redundancy and ambivalence, circularity is not checked for at all, and the check for missing rules is restricted to combinations of parameters and values which are not covered by rules in a specific context. Since inference chains are not checked, the program will fail to detect many anomalies, such as the redundant third rule below:

$$L_1 \rightarrow L_2$$
$$L_2 \rightarrow L_3$$
$$L_1 \rightarrow L_3$$

Similarly, missing rules which can only be revealed by considering chains of inference will not be discovered.

3.2 The CHECK *Program*

The CHECK (Nguyen et al., 1985, 1987; Perkins et al., 1989) program originated in the mid-1980s at Lockheed Corporation as a verification facility for their in-house expert system shell, the Lockheed Expert System (LES). LES is a rule-based tool similar to EMYCIN, featuring data-driven rules (forward-chained, sometimes called demons) and goal-driven rules (backward-chained). Inference in LES is monotonic in nature. Data is represented by object-attribute-value (OAV) triples, which can be represented in our first-order logic notation as binary predicates ATTRIBUTE(*object*, *value*), for example SEX(*Marvin*, *Male*).

The CHECK program builds upon the RCP work, expanding and redefining the range of anomalies checked for. A later paper on CHECK (Perkins et al., 1989) says that the program makes a distinction between goal-driven and data-driven rules, but this distinction apparently affects only the naming of one of the anomalies (unreachable conclusion), discussed below. CHECK detects the following anomalies:

Redundant rules The CHECK definition of "redundancy" applies only to pairs of duplicate rules, as we have defined them. That is, two rules have the same consequent, and the antecedent literals of one rule are a permutation of the antecedent literals of the other, after renaming variables if necessary. For example, the pairs of rule $P(x) \wedge Q(x) \rightarrow R(x)$ and $Q(y) \wedge P(y) \rightarrow R(y)$ is reported as a case of redundancy by CHECK.

Conflicting rules CHECK defines conflicting rules similarly to redundant rules, except that the consequents are complementary literals (one is the negation of the other). Note that this means that the following rules are *not* conflicting, according to CHECK, because the two antecedents are not exact logical duplicates: $P(x) \wedge Q(x) \rightarrow R(x)$ and $Q(y) \rightarrow \neg R(y)$.

Subsumed rules This is a relaxation of the "redundant rules" definition, in that the antecedents of the pair of rules need not be duplicates: one may subsume the other. This corresponds to our definition of subsumed rules, as a special case of redundancy (see section 2.2.2).

Circular rules Although not stated formally, this is equivalent to our definition (see section 2.4).

Unreferenced attribute value Although this was originally defined for CHECK as "missing rules", it is really a check for missing values, as we have defined them (see section 2.5).

Illegal attribute values/dead-end goals Although defined as two different anomalies, these represent the same logical problem. This problem is a special case of unfireable rule, in which an antecedent clause requires a certain OAV combination, but the given value cannot be obtained for that object-attribute combination (either by asking the user or by firing a rule).

Unreachable conclusion This anomaly is similar to our definition of a rule with an unusable consequent (section 2.2.1). Note that their use of the term "unreachable conclusion" is under the implicit assumption that the rule is to be used in backward-chaining (the rule will never be used to establish a goal, because its consequent does not appear in the antecedent of any other rule, and is not a final hypothesis). This is the only case in which CHECK draws a distinction between data-driven and goal-driven rules: (Perkins et al., 1989) asserts that this anomaly is not a problem for forward-chained rules. We would argue that, in forward-chaining systems, such a rule could be termed "useless" rather than "unreachable", since it never infers a conclusion which can be reported to the user or used to fire other rules. The distinction is in their terminology only.

3.2.1 Implementation of CHECK

CHECK detects most of the anomalies defined above by building three cache tables from the rule base: the "if-if" table, the "then-then" table, and the "then-if" table. Each of these tables has a row and column for each rule in the rule base. The entries in the "if-if" and "then-then" tables are created by comparing the antecedents and consequents of each pair of rules, and entering one of the results *same*, *different*, *subset*, *superset*, and *conflict* in the appropriate table. These two tables can be used to detect most of the defined anomalies such as redundancy and conflict. For example, two rules are redundant if their entry in the "if-if" table is *same*, and the "then-then" table is also *same*.

The "then-if" table is used to detect circular inference chains. If the consequent of one rule appears in the antecedent of a second rule, then the two rules may form an inference chain, and an entry is placed in the "then-if" table. This table is checked for circular inference chains using a cyclic graph detection algorithm.

As an example, "if-if", "then-then" and "then-if" tables for the rule set below appear in Table 2:

(1) UniversityMember $\wedge$ Enrolled $\rightarrow$ Student

(2) Student $\wedge \neg$HasBScDegree $\rightarrow$ Undergraduate

(3) Student $\rightarrow$ Undergraduate

(4) Student $\wedge$ HasBScDegree $\rightarrow$ GraduateStudent

(5) UniversityMember $\wedge \neg$Enrolled $\rightarrow$ Staff

(6) Student $\rightarrow$ Enrolled

The "if-if", "then-then" and "then-if" tables can each be created and checked with $n(n-1)/2$ comparisons, where n is the number of rules in the rule base. Each comparison however, may be quite time-consuming, depending on the complexity of the rules. The pseudocode for most of the CHECK algorithms is presented in (Nguyen et al., 1985) (original algorithms) and (Perkins et al., 1989) (revised algorithms).

3.2.2 Limitations of CHECK

The most fundamental limitations of CHECK are the narrow range of anomalies checked: there is no consideration of redundancy and ambivalence over inference chains, and the check for deficiency is limited to missing values. Also, no examples of the use of CHECK on real knowledge bases are presented in the papers. ARC, a variant of CHECK tailored to checking knowledge bases for the ART tool[3], is described in Nguyen (1987).

[3]ART is a trademark of Inference Corporation.

Table 2 Set of cache tables generated by the CHECK tool for a simple rule base. (a) "if-if" table; (b) "then-then" table; (c) "then-if" table. Note that rule 2 is subsumed by rule 3 (same "then-then", superset "if-if"), rules 3 and 4 are in conflict (conflict "then-then", subset "if-if"), and there is a cycle between rules 1 and 6 (from the "then-if" entries for these rules)

	1	2	3	4	5	6
1	—	different	different	different	different	different
2		—	superset	different	different	superset
3			—	subset	different	same
4				—	different	superset
5					—	different
6						—

a

	1	2	3	4	5	6
1	—	different	different	different	conflict	different
2		—	same	conflict	conflict	different
3			—	conflict	conflict	different
4				—	conflict	different
5					—	conflict
6						—

b

	1	2	3	4	5	6
1						×
2	×					
3	×					
4	×					
5						
6	×					

c

3.3 The KB-Reducer program

The KB-Reducer program (Ginsberg, 1988), developed at AT&T Bell Laboratories, is part of a larger research effort into theory reduction and machine learning (Ginsberg, 1990). The program is inspired by the assumption-based truth maintenance system (ATMS) (de Kleer, 1986), and finds the sets of assumptions which must hold for each knowledge base hypothesis to be true. These sets of assumptions are called *labels* for the hypotheses, and are minimal disjunctive normal form expressions. Each literal in these expressions is a finding, and each conjunction is an *environment* for the hypothesis. The usage of these terms is consistent with ours, except that Ginsberg's environments contain only findings, while ours contain any askable items (which is a superset of all findings—see section 2.1). The KB-Reducer program uses the labels to detect redundancy and ambivalence within rule chains, and therefore represents a major step forward from the CHECK and RCP approaches. The KB-Reducer makes no attempt to check for deficiency. Circularity, although not checked as a specific aim of the program, is actually detected as a side-effect of the KB-Reducer algorithm, as we describe below.

The KB-Reducer program is designed for rule languages based on propositional logic, featuring OAV triples for data representation. Although the program is not limited to any particular rule language, a few assumptions are made about the nature of the inference engine which will be used by the expert system. For KB-Reducer to produce entirely meaningful results, the inference engine

needs to be monotonic, non-selective (any rule whose antecedent is satisfied can fire immediately), and strongly data-driven (all available findings are made available before any inferences are drawn). Even if these assumptions are relaxed to some extent, however, Ginsberg claims that KB-Reducer can still provide useful analyses of knowledge bases.

In addition to findings and hypotheses, KB-Reducer identifies certain literals as *default hypotheses*. These are literals which occur only in rule antecedents, and are negations of hypotheses (which are called the counter-hypotheses of the default hypotheses). The label for a default hypothesis is computed by finding the label of its counter-hypothesis, negating it, and transforming the resulting expression to disjunctive normal form. For example, consider the following knowledge base:

$$\begin{aligned} L_1 \vee L_2 &\rightarrow M_1 \\ L_3 &\rightarrow M_2 \\ M_1 \wedge M_2 &\rightarrow M_3 \\ \neg M_3 \wedge L_4 &\rightarrow \neg M_1 \end{aligned}$$

Here, L_1, L_2, L_3 and L_4 are findings, M_1, M_2, M_3 and $\neg M_1$ are hypotheses, and $\neg M_3$ is a default hypothesis. The label for M_1 is $L_1 \vee L_2$, the label for M_3 is $(L_1 \wedge L_3) \vee (L_2 \wedge L_3)$, and the label for $\neg M_3$ is $(\neg L_1 \wedge \neg L_2) \vee \neg L_3$. In applying this definition of default hypothesis, a closed world assumption is made for findings—unless the negation of a finding is explicitly present in a rule antecedent, it is assumed that it is true if the finding is not present (that is, in the example, $\neg L_1$ is assumed true if L_1 is not given as an input to the knowledge base).

3.3.1. The KB-Reduction algorithm

To apply KB-Reducer, a knowledge base must be stratified into *levels*, according to inference dependencies in the rules. A rule is at level 0 if its antecedent contains only findings; otherwise, it is at one level more than the level of the highest-level rule on which it depends. In the example, the first two rules are at level 0, while the third rule is at level 1, since it depends on the level 0 rules. The final rule is at level 2, since a rule containing a default hypothesis must be at a level higher than the level of the counter-hypothesis. This requirement means that KB-Reducer cannot be applied to rule bases with circular inference chains, but the stratification algorithm will detect circularity. In other words, KB-Reducer will always detect circularity, but if there is circularity then it will never detect redundancy or ambivalence.

The KB-Reduction algorithm works by processing rules in order of level, from level 0 up. The labels for hypotheses are built up until they are complete—all will be complete only after all rules have been processed. The working labels are called *partial labels*; the partial label for a hypotheses is updated whenever a rule is processed with the hypothesis in its consequent, and the label for a default hypothesis is computed when a rule is processed which has the default hypothesis in its antecedent.

When KB-Reducer processes each rule, it first determines the *rule label* (the set of minimal environments that satisfy the antecedent of the rule), by minimizing the conjunction of the labels for the literals in the antecedent. The partial label for the hypothesis in the consequent of the rule is then updated by forming the disjunction of its current partial label and the rule label, and minimizing this expression. KB-Reducer applies checks for redundancy and ambivalence immediately after computing each rule label. Note that, in order to report any anomalies to the user, KB-Reducer needs to record the identities of rules used to create labels, so that the user can inspect the problematical inference chains and rules, and decide what action, if any, to take to remove the anomaly. The KB-Reducer checks are defined as follows:

The redundancy check The KB-Reducer redundancy check is performed in two phases. A check for unfireable rules is done immediately after computing each rule label. If the label consists entirely of impermissible environments, then the rule is unfireable (this agrees with our definition

of unfireable rule in section 2.2.1). Note that the user of KB-Reducer is able to specify impermissible environments, called *semantic constraints* by Ginsberg (1988).

Once all labels have been computed, KB-Reducer checks for redundant rules (section 2.2.2) by examining the labels for all hypotheses. The rules whose rule labels contribute to each label are identified, and any rule which uniquely determines some environment for some hypothesis is defined to be *non-redundant*. After all non-redundant rules have been determined, any remaining rules are considered redundant. For example, consider the rules below:

$$L_1 \rightarrow L_2$$
$$L_2 \rightarrow L_3$$
$$L_1 \rightarrow L_3$$

The label for L_2 is L_1 (from the first rule), and for L_3 is L_1 (from either the second or last rules). The first rule is non-redundant because it uniquely determines an environment in a label (for L_2). The second and last rules are considered redundant by KB-Reducer. In this case, either of these rules (but not both) may be removed from the knowledge base. In general, it is necessary for the knowledge base designer to inspect the redundant sets of rules reported by KB-Reducer to determine which can be removed safely. Note that this procedure for detecting rules is described in a later paper (Ginsberg and Williamson, 1989). The original KB-Reducer redundancy detection procedure described in Ginsberg (1988) had a flaw, in that it was sensitive to the order in which rule labels were computed.

The ambivalence check The ambivalence check is applied immediately following the unfireable rules check, and after the partial label for the consequent hypothesis is updated. KB-Reducer inspects the partial labels for each hypothesis which conflicts with the consequent hypothesis of the current rule. Conflicting hypotheses include complementary literals, and hypotheses which are specified by the user as conflicting, by means of semantic constraints. For example, conflicting hypotheses for M are $\neg M$ and N, if $\{M, N\}$ is an impermissible set (semantic constraint). If there is an environment in the partial label of a hypothesis which is either a subset or a superset of an environment in the partial label of a conflicting hypothesis, then KB-Reducer reports a contradiction. If there is an environment in each of the two partial labels (which are not subsets/supersets) for which the union is not impermissible, then KB-Reducer reports a *potential contradiction*. This means that, unless there is some additional semantic constraint that rules out the simultaneous satisfaction of both partial labels, both conflicting hypotheses will be inferred. This case of contradiction corresponds to our general use of ambivalence.

KB-Reducer: an example Consider the following rule base:

(1) `UniversityMember` $\wedge$ `Enrolled` $\rightarrow$ `Student`
(2) `Student` $\wedge \neg$`HasBScDegree` $\rightarrow$ `Undergraduate`
(3) `Student` $\wedge$ `HasBScDegree` $\rightarrow$ `GraduateStudent`
(4) `Enrolled` $\wedge \neg$`HasBScDegree` $\rightarrow$ `Undergraduate`
(5) `UniversityMember` $\wedge$ `Teacher` $\rightarrow$ `Staff`

In this example, the following impermissible sets are assumed to be declared:

{`Student, Staff`} and {$\neg$`UniversityMember, Enrolled`}

Table 3 shows the labels that would be computed by KB-Reducer for each hypothesis.

To check for ambivalence, KB-Reducer inspects the labels for the hypotheses `Student` and `Staff`, since these have been declared as forming an impermissible set. From the labels, we see that rules 1 and 5 are ambivalent, because the union of the environments from their labels is the permissible environment {`UniversityMember, Enrolled, Teacher`}.

To check for redundancy, KB-Reducer locates any rules which do not uniquely contribute any environments to a label. Rules 2 and 4 both contribute only to the label for `Undergraduate`, so either rule 2 or rule 4 is redundant.

3.3.2 Performance considerations for KB-Reducer

KB-Reducer is implemented in LISP, and takes 10 CPU hours (running on a TI Explorer I workstation[4]) to check fully a 570 rule knowledge base, generating 35,000 environments to do so (Ginsberg, 1988). This seems acceptable, since the users need not be present while the system runs. Also, once a knowledge base has been reduced, when new knowledge is added it should not be necessary to re-run the entire reduction again, only those parts of the reduction affected by the changes.

The performance complexity of KB-Reducer is porportional to the number of environments that it must generate to check a knowledge base. The worst case is when, for each finding f, environments must be generated which contain either f, $\neg f$, or neither. This means that 3^n environments need to be generated, for n findings. The worst case occurs only when every combination of findings is included in the label for a hypothesis. Since a label contains minimal (non-subsumable) environments, this would require a knowledge base designer to write rules which make distinctions between 3^n data combinations, for n data items. For a small system with 25 data items, for instance, this would require a knowledge base with several million rules—certainly larger than any known hand-crafted system built to date. Therefore Ginsberg concludes that this worst case will rarely occur. Similar arguments are presented by de Kleer in support of the ATMS, and by Preece (Preece et al., 1992) in support of the COVER deficiency checking method (see section 3.5).

Rousset (1988) describes a very similar approach to Ginsberg's. We chose to focus on Ginsberg's work since it was apparently started earlier, and provides more complete descriptions of the algorithms used.

3.4 The EVA system

The Expert System Validation Associate (EVA) system (Chang et al., 1990a; Stachowitz & Combs, 1987; Stachowitz et al., 1987) is the focus of a long term research project at Lockheed Corporation to develop a general-purpose verification and validation work-bench for expert systems. EVA is a loosely-integrated set of checking tools built around a theorem prover and database. The system is implemented entirely in Prolog. The checking tools include anomaly checkers such as the structure checker, logic checker and omissions checker, and also validation tools such as the test case generator and rule refiner. In this paper we focus exclusively on the anomaly detection tools.

EVA is not tied to any particular rule language or shell. One of its features is that it provides a set of translation programs which translate the rule languages of many widely-used expert system tools

Table 3 Table showing example labels generated by KB-Reducer from a simple rule base. Parenthesized numbers in the superscripts are numbers of the rules used in generating the environments in each label

Hypothesis	*Label*
`Student`	`(UniversityMember` $\wedge$ `Enrolled)`$^{(1)}$
`Undergraduate`	`(Enrolled` $\wedge$ $\neg$`HasBScDegree)`$^{(4)}$
	`(UniversityMember` $\wedge$ `Enrolled` $\wedge$ $\neg$`HasBScDegree)`$^{(2)}$
`GraduateStudent`	`(UniversityMember` $\wedge$ `Enrolled` $\wedge$ `HasBScDegree)`$^{(3)}$
`Staff`	`(UniversityMember` $\wedge$ `Teacher)`$^{(5)}$

[4]TI Explorer I is a trademark of Texas Instruments Inc.

and shells (for example, ART, OPS5 and LES) to a canonical internal form, based on first-order logic. EVA also supports a metalanguage which can be used by knowledge base designers to specify higher-order semantic constraints (such as impermissible sets, and predicate argument types and ranges), which would not be expressible in the original rule languages. For each knowledge base written in one of the languages supported by an EVA translator, EVA reads and translates the source knowledge base and any accompanying metalanguage constraints, and stores the combined knowledge in its internal database, in canonical form. This approach, combined with the discrete nature of the checking tools, means that the EVA system is highly extensible.

3.4.1 Anomaly checks performed by EVA

We discuss the anomaly checks performed by EVA within the framework of our own definitions.

Redundancy checking Apparently as a legacy from the earlier CHECK system at Lockheed, EVA includes checks for "dead-end rules" and "unreachable facts/literals". These are the same as the "dead-end goal" and "unreachable conclusion" checks performed by CHECK, and make the same implicit assumption (in terminology) that the rules will be invoked by backward-chaining. As in CHECK, the former case is a specialization of our definition of unfireable rule, and the latter is equivalent to our definition of rule with an unusable consequent.

Our general case of redundancy is checked in EVA by theorem proving. Assume that rules are normalized so that only single literals appear in their consequents. In this case, the EVA definition of redundancy is exactly equivalent to ours; that is, if K is a knowledge base, and R is a rule in K, then R is redundant if it can be derived from $K - \{R\}$. EVA performs this check by removing R from K, replacing variables in R by Skolem constants, adding the Skolemized literals from the antecedent of R as facts in K, and trying to prove the Skolemized consequent of R. If the Skolemized consequent of R can be derived from K and the Skolemized premises of R, even in the absence of rule R, then R is redundant.

All the above redundancy checks are performed in EVA by the *structure checker* tool. In addition, the *extended structure checker* can use metaknowledge about synonym and generalization relations (ISA-links) in checking for further redundancies. For example, if the literals $L1$ and $L4$ are declared to be synonyms, EVA will detect redundancy in the following rules:

$$L1 \rightarrow L2$$
$$L2 \rightarrow L3$$
$$L4 \rightarrow L3$$

The *rule satisfiability checker* extends the redundancy checks by checking ranges, and combinations of data items/values which cannot be satisfied. Such a case would occur if a single-valued data item were required to take on two distinct values in order for a particular rule to fire. This extends the ability of EVA to detect unfireable rules, but does not enable it to address the full general case of an unfireable rule, as we have defined it.

Ambivalence checking Our general case of ambivalence is defined as "inconsistency" in EVA. A metapredicate `INCOMPATIBLE` is used to declare impermissible sets of literals. For the basic ambivalence check, the declaration `INCOMPATIBLE`(H, $\neg H$) declares any pair of complementary hypotheses to be impermissible. For the more general ambivalence check, the user is able to specify any set of literals $L1 \ldots Ln$ as an impermissible set using the declaration `INCOMPATIBLE`($L1, \ldots, Ln$).

For every declaration `INCOMPATIBLE`($L1, \ldots, Ln$), the EVA theorem prover tries to prove the goal $L1 \wedge \ldots \wedge Ln$. This proof is conducted by backward-chaining, and when the theorem prover encounters a finding (to use Ginsberg's terminology), it assumes that the finding is supplied. A proof tree so constructed for the goal is called a *proof residue*, and states what combination of findings would actually be required to prove the goal. If the theorem prover succeeds in finding one or more residual proof trees for goal, then there is ambivalence in the knowledge base. Further

details of this method are provided in McGuire (1990). Note that this method is similar to Ginsberg's ATMS-inspired approach, except that Ginsberg's method detects *all* cases of ambivalence in one pass through the knowledge base, in a manner similar to forward-chaining, whereas the proof residue method detects each potential case of ambivalence individually, by backward-chaining. The basic ambivalence check (pairs of complementary literals inferred) is applied by the EVA *logic checker*, while the general check for inferrability of unrestricted impermissible sets is applied by the *extended logic checker*.

Circularity checking For each predicate in a knowledge base represented in EVA canonical form, EVA computes a metapredicate DERIVE which states each derivation path for the predicate. Each derivation path is a list of rules, and if there are duplicates in a list, then there is a circular inference path in the knowledge base. This checks the general case of circularity as we have defined it. The circularity check in EVA is performed by the structure checker.

Deficiency checking As in the earlier CHECK system, the deficiency check in EVA is restricted to detecting special cases of deficiency, rather than missing rules in general. In the papers on EVA, only the check for missing values is explicitly mentioned. The check for missing values is performed by the EVA *omission checker*.

3.4.2 Limitations of the EVA work

While the scale of the EVA project is impressive, the designers do not always make a clear distinction in their papers (known to us) between finished work, work-in-progress, and future work. It is therefore difficult to assess how successful are many of the checks and algorithms described. This problem is exacerbated by the fact that only "toy" examples are presented in the papers: no results are included from using EVA to check large real-world knowledge bases (although the empirical performance of some of the methods is discussed for mechanically-generated knowledge bases in McGuire, 1990). Without such information, there is no way to tell whether the theorem-proving methods employed by the system (which are intractable in the worst case) are practical for checking large-scale application knowledge bases.

3.5 The COVER *tool*

The COVER tool (Preece, 1989; Preece et al., 1992; Preece & Shinghal, 1991) tool was designed to build upon the best features of earlier systems. For example, COVER has separate checks for rule-pair anomalies (as in the RCP and CHECK) and full inference chain anomalies (as in KB-Reducer and EVA). Although there is some overlap between the different checking operations COVER performs, the advantage of this approach is that the knowledge base designer has more flexibility in choosing the depth to which a knowledge base can be checked in a particular session.

The COVER tool is designed to check a canonical rule language based on a subset of first-order logic. This language incorporates many features common to rule-based expert system shells, and it appears to be straightforward to transform knowledge bases written in a variety of different commercial expert system shells into the COVER rule language. In the COVER rule language, each rule consequent assigns a value to a parameter called the rule *subject*. Data items are either single-valued *parameters* or multiple-valued *sets*. Values for parameters have determinable types and ranges. COVER assumes that rules will be used for monotonic reasoning, and that certainty factors will not be used. Note that in addition to rules, COVER rule bases may contain declarations of impermissible sets and askable parameters, as defined in section 2.1.

Although COVER has specific modules for implementing checks similar to the RCP and CHECK (see section 4 for details), we focus here on the novel features of COVER: the environment and deficiency checker.

3.5.1 Environment and deficiency checker

COVER implements the full check for missing rules, as defined in section 2.5, where a missing rule is reported if there exists a permissible environment for which no final hypothesis is inferred. The user of COVER is allowed to select a final hypothesis or set of final hypotheses for the missing-rules

check. Then, only rules and parameters that relate to the search for these final hypotheses are checked. This is a technique for reducing the search space for the deficiency check. If the knowledge base contains a single final hypothesis, which the user selects, then COVER checks the entire knowledge base for deficiency. Otherwise, the user can check a sub-module of the knowledge base by selecting the final hypothesis or hypotheses of that module.

3.5.2 Implementation of COVER

The COVER program is implemented in Prolog and C. COVER checks the syntax of the knowledge base before checking starts. Each rule in the COVER rule language is written in disjunctive normal form, and COVER normalizes the rules by splitting each conjunction into a separate rule with the same consequent, so that each rule is in the form defined in section 2.1. The original forms of the rules are stored for reference, so that any anomalies can be reported to the user more naturally.

The COVER environment checker used a procedure similar to KB-Reducer's for checking redundancy and ambivalence, and couples this with a novel method for checking deficiency. Central to the COVER procedure is the concept of a *goal environment*, which is defined as an environment that contains the minimal set of findings necessary for the inferring of a specific final hypothesis, or goal. Goal environments can usefully be thought of in terms of proof trees for goals: each goal environment is a set of the leaves from a proof tree for some goal. This corresponds to the label for a final hypothesis, in the KB-Reducer system. Note, however, that COVER environments contain askable items rather than findings (see section 2.1).

The implementation of the COVER environment checker is divided into three sub-procedures: creation of goal environments; checking of goal environments for redundancy and ambivalence; and checking of goal environments for deficiency. The user must first select which goal, or set of goals, are to be checked. As previously mentioned, this choice depends on the modularity of the system, and the extent to which the user wishes to check the knowledge base at that particular time. After the goal or goal set has been selected by the user, COVER uses a backward-chaining procedure with backtracking to find all the goal environments for each goal in the selected set. The number of goal environments e for a particular goal is dependent on the number of rules m which have the goal as their subject, and the length of inference chains in the knowledge base k. The number e is proportional to m^k. Performance of the procedure used to find the goal environments is $O(e)$.

Note that the procedure used to establish goal environments by COVER is in some ways similar to that used to establish rule labels by KB-Reducer. While creating the goal environments, COVER performs checks which are equivalent to those performed by KB-Reducer for redundancy and ambivalence. The performance of this check is $O(e^2)$. The main difference between this part of the COVER program and Ginsberg's system is that COVER uses a goal-directed strategy (because the goal set is selected by the user) and KB-Reducer uses a data-driven procedure (more specifically, it is finding-driven). This means that when COVER detects redundancy and ambivalence during the rule chain check, it does so only for the rules used in creating the goal environments for the selected goal-set. Therefore, rules which are reported as redundant in this set of rules may not be redundant when the entire knowledge base is considered. Nevertheless, a well-modularized knowledge base should consist of largely disjoint rule sets, so this will often not be a problem. Note also that during the procedure for finding goal environments, COVER automatically discovers any cyclic inference chains. However, these do not necessarily prevent COVER from forming goal environments (as was the case in KB-Reducer), since most cycles are broken by askable items which can always be supplied by the user.

To check deficiency in the set of goal environments, COVER uses a novel method to try and avoid the combinatorial explosion which renders the problem intractable for complex rule bases. Instead of generating and testing every combination of parameters and values present in the set of goal environments, COVER employs two heuristics, as follows:

- Using the information in the existing rules, COVER determines which parameters are *relevant* to one another, and restricts generation of environments to combinations of these parameters and their values. Such cases are intended to represent plausible missing rules.

- COVER generates environments in order of size, and if it finds missing environments, it need not generate environments which subsume these, because if such environments represent missing cases, they will not be minimal.

An example will illustrate this approach. Consider the following small knowledge base, in which `Undergraduate`, `Staff`, and `Professor` are goals:

```
Enrolled ∧ ¬HasBScDegree  →  Undergraduate
¬Enrolled ∧ HasBScDegree  →  Staff
  HasBScDegree ∧ Teaches  →  Professor
```

For simplicity in reading, this example is expressed in propositional logic, rather than the parameter/value form used by COVER. In the discussion that follows, we assume that an expression L, where L is one of the propositional symbols used in the rules (for example, `Teaches`) means that parameter L has the value *True*, while $\neg L$ means that L has the value *False*. Here, the goal environments are:

{Enrolled, ¬HasBScDegree}
{¬Enrolled, HasBScDegree}
{HasBScDegree, Teaches}

Some examples of covered environments are as follows:

1. {Enrolled, ¬HasBScDegree} is covered by the first goal environment above: using the rules, the goal `Undergraduate` would be inferred from this environment.
2. {HasBScDegree, Teaches, ¬Enrolled} is covered by the third goal environment above: using the rules, `Professor` would be inferred from this environment. Note that the literal ¬Enrolled is superfluous—the environment would still be covered without it.
3. {¬Enrolled} is covered by the second goal environment above: from the rules, a partial proof tree for the goal `Staff` can be constructed from this environment.
4. {¬HasBScDegree, Teaches} is covered by both the first and third goal environments above: using the rules, partial proof trees can be constructed for both the goals `Undergraduate` and `Professor`.

The following environments are the only uncovered environments in this example:

{¬Enrolled, ¬HasBScDegree}
{¬Teaches}

From these environments, using the rules, no goal can be inferred and no partial proof trees can be constructed for any goal. To detect these uncovered environments, COVER would generate the following environments, in sequence:

{Enrolled}, {¬Enrolled}
{HasBScDegree}, {¬HasBScDegree}
{Teaches}, {¬Teaches}
{Enrolled, HasBScDegree}
{Enrolled, ¬HasBScDegree}
{¬Enrolled, HasBScDegree}
{¬Enrolled, ¬HasBScDegree}
{HasBScDegree, Teaches}
{¬HasBScDegree, Teaches}

Note the effects of the two heuristics here. The first heuristic results in environments containing relevant items which, in this example, are the following:

{Enrolled, ¬Enrolled, HasBScDegree, ¬HasBScDegree}
{HasBScDegree, ¬HasBScDegree, Teaches, ¬Teaches}

Therefore, environments such as {Enrolled, Teaches} will not be generated due to the first heuristic. Also, no environment containing three items will be generated. The second heuristic leads to the generation of environments in order of size, with the smallest (one item) being generated first. When an environment is found to be uncovered, no environment containing it is generated. For example, if {¬Teaches} is found to be uncovered, there is no need to generate {HasBScDegree, ¬Teaches} or {¬HasBScDegree, ¬Teaches}.

In this simple example, the COVER approach does not lead to a substantial reduction in the number of environments generated. Some experimental data from trials of COVER on application knowledge bases are reported in Preece (1989). For a 162-rule knowledge base with 16 parameters, each with six values, 1.68×10^5 environments were generated by COVER. The RCP would need to generate $6^{16} = 1.7 \times 10^{13}$ environments to check the same knowledge base. For a 144-rule knowledge base with 49 parameters, each with three values, only 1.1×10^4 environments were generated by COVER. The RCP would need to generate $3^{49} = 2.4 \times 10^{23}$ environments to check the same knowledge base. In addition to this reduction in the search space for the deficiency check, COVER offers two further advantages over the RCP approach:

- The reported missing environments are minimal. Because they cannot determine "don't care" parameters in missing cases, the RCP would report missing cases with some value for every one of the parameters appearing in their tables. For example, for the 49-parameter rule base mentioned above, the RCP would report missing cases each with 49 parameter/value pairs. This can make it hard for the user to identify the actual deficiencies. As a simple example of this, consider the missing case {¬Teaches} above. The RCP would report all of the following cases as missing:

  {¬Teaches, Enrolled, HasBScDegree}
  {¬Teaches, Enrolled, ¬HasBScDegree}
  {¬Teaches, ¬Enrolled, HasBScDegree}
  {¬Teaches, ¬Enrolled, ¬HasBScDegree}

  COVER, by reporting the minimal case, identifies the source of the problem.

- Unlike COVER, the RCP does not provide full deficiency checking over inference chains: both are restricted to checking against the antecedents of rules which conclude values for the same parameter. Consider the following example:

  $$\begin{aligned} L &\rightarrow M \\ \neg L &\rightarrow N \\ N &\rightarrow M \end{aligned}$$

  Here, the only askables are L and $\neg L$. Checking for deficiency in the rules for M, the RCP would report $\neg L$ and $\neg N$ as missing cases. COVER, however, will not report any missing cases, because it can determine from the inference chains that the case N never arises (it has no goal environment) and $\neg L$ is covered by the second rule.

No missing cases will ever be lost by applying the COVER heuristics, because in the worst case the heuristics will simply fail to reduce the search space. In this case the performance of COVER is equivalent to that of the RCP deficiency check. Of course, in practice, the check may be intractable for some knowledge bases (particularly those with a large number of rules, long inference chains, and complex rule antecedents), but the COVER procedure can still provide *partial* deficiency checking by setting a user-defined threshold for the maximum size of environment to be generated during the checking process.

The limitations of the COVER deficiency checking procedure lie in the assumptions made by the two heuristics. The first, that of considering only relevant combinations of parameters, relies on the assumption that all such combinations will be determinable from inspection of the existing rules. If the knowledge engineer fails to relate two parameters via rules, then COVER may not detect missing rules which use a combination of these items. The question here is whether it is more reasonable to use information about the relationships between parameters as indicated in the existing knowledge base, or to make the assumption that any set of parameters may be mutually relevant. The first case entails that all relevant combinations of parameters must be determinable from the existing knowledge base; the second case can result in the reporting of a large number of semantically-meaningless rules, including parameters that would never ordinarily be present together in the same rule.

The second heuristic used by COVER is that of testing smaller environments first, and not checking any larger environments that are subsumed by the smaller ones. While this guarantees that all logically-deficient combinations of parameters and values will be found by COVER, it also results in *minimal* deficient cases being reported, unlike the RCP method, which reports missing cases including a value for ever parameter present in the table. It is possible to criticize COVER on the grounds that the minimal combinations may omit relevant item/value pairs. However, it is equally possible to criticize the other approaches on the grounds that they often include unnecessary item/value pairs. This comes down to a design choice: is it more reasonable to present minimal cases to the users, who can then decide whether additional constraints are required on any rules which may be added to fill the gaps, or should the users be expected to prune unnecessary conditions? In Preece et al. (1992) it is argued that unless minimal cases are reported, the user will often have to look through a very large number of specific missing cases, where the actual missing combinations are quite general.

The efficacy of the COVER tool is demonstrated by the inclusion of examples of its use (Preece, 1989; Preece et al., 1992; Preece & Shinghal, 1991).

4 Conclusion and discussion

In this paper we have explained four types of anomaly that we need to look for to verify expert system rule bases. Using these anomalies as a framework for comparison, we have discussed the extent to which five expert system verification tools address the problem of checking for the anomalies. To conclude, we consider a number of issues arising from this study.

Coverage of the tools Tables 4 to 7 summarize the extent to which the five tools detect the four types of anomaly. These tables facilitate comparison of the functionality of the tools used in this survey, and clearly show the progression from the simple anomaly checks of the early tools to the more sophisticated checks of the later tools.

Cost of detecting the anomalies In general, we can group the anomaly checking tasks into three categories, based roughly on the computational complexity of the algorithms used to implement them. If we view the rule base as a graph, in a manner similar to the CHECK "then-if" table (see Table 2), the *integrity checks* are those that check the connectivity of this graph. These checks include special cases of redundancy (unusable rules), deficiency (missing values) and circularity. These can be checked by algorithms with linear complexity, proportional to the number of rules in the knowledge base. The *rule checks* are those that compare pairs of rules for the special cases of redundancy (duplicate and subsumed rules) and ambivalence (conflicting pair of rules) first implemented in the RCP. The complexity of these checks is quadratic. The *environment checks* are those that check the full, general cases of redundancy, ambivalence and deficiency over inference chains, with exponential complexity.

In view of these issues, the COVER tool offers three distinct checking phases, grouped by the methods used to implement them: *integrity checking*, *rule checking*, and *environment checking*. The reasons for this separation are pragmatic. The computational cost of performing the

three categories of checks is very different in each case, and this means that the user can select the type of checking most appropriate at a given time. For example, the integrity check (with linear complexity) may take only a few minutes (on, say, a Sun Workstation), and can be run each time a few new rules have been added to a knowledge base; the environment check (with exponential complexity) often takes several hours, and would usually be run only after major modifications have been made to the knowledge base. The three types of checking are related: anomalies detected by integrity checking often cause anomalies which are detected by rule checking, and anomalies detected by rule checking always cause anomalies detected by environment checking. By performing the three separate phases of checks, the root causes of anomalies can often be made more apparent. For example, if there is a pair of conflicting rules, and environment checking is performed, then there will be ambivalence in every pair of inference chains that terminate with the conflicting pair. This is illustrated by the following example:

$$L_1 \rightarrow N \quad (1)$$

$$L_2 \rightarrow N \quad (2)$$

$$N \rightarrow M \quad (3)$$

$$N \rightarrow \neg M \quad (4)$$

The conflicting pair (rules 3 and 4) result in the reporting of ambivalence from inference chains {1,3} and {1,4}, and from {2,3} and {2,4}. Clearly, a few anomalous pairs of this kind could lead to the reporting of many anomalous chains; by running the rule checker before the environment checker, the user is able to identify and resolve rule pair anomalies before they lead to inference chain anomalies.

Another reason for separating the checks is to give the user the ability to select only those checks that he or she feels are appropriate to the application at hand. For example, in some systems circularity is widely and safely used (that is, rules representing mutual implications are common in the knowledge base, and there is no danger that the inference engine will enter an infinite loop when using such rules). In such a case, the user will wish to "turn off" the circularity check and any practical vertification tool should permit this. Empirical evidence for the utility of certain

Table 4 Comparison of the capabilities of five expert system verification tools for checking redundancy in rule bases

Tool	*Unusable rules*	*Redundant rules*
RCP	Does not check for either unfireable rules or rules with unusable consequents	Detects only redundant pairs of rules; rules are grouped by designer according to context. ESC tool provides automatic grouping of rules into contexts
CHECK	Detects unfireable rules which test a parameter against a value that the parameter can never take. Detects rules with unusable consequents	Detects only redundant pairs of rules
KB-Reducer	Implements the full test for unfireable rules. Does not detect rules with unusable consequents	Detects full, general case of redundant rules by forward-chaining ATMS-inspired method
EVA	Tests the same case of unfireable rules as CHECK. Also considers case in which, for a rule to be satisfied, some parameters would have value constraints violated	Detects full, general case of redundant rules by backward-chaining, theorem proving method
COVER	Tests the same case of unfireable rules as CHECK. Detects rules with unusable consequents	Detects full, general case of redundant rules by backward-chaining ATMS-inspired method

Table 5 Comparison of the capabilities of five expert system verification tools for checking ambivalence in rule bases

Tool	*General case*
RCP	Checks only for pairs of rules which have subsumed antecedents and assign different values to a single-valued parameter
CHECK	Checks only for pairs of rules with logically identical antecedents and conflicting consequents
KB-Reducer	Detects full, general case of ambivalence, based on declarations of impermissible sets. Uses forward-chaining ATMS-inspired approach, tightly coupled with redundancy detection
EVA	Detects full, general case of ambivalence, based on declarations of impermissible sets. Uses backward-chaining proof residue approach, performed independently from redundancy detection
COVER	Detects full, general case of ambivalence, based on impermissible set declarations. Uses backward-chaining ATMS-inspired approach, tightly coupled to redundancy detection

Table 6 Comparison of the capabilities of five expert system verification tools for checking circularity in rule bases

Tool	*General case*
RCP	Does not check
CHECK	Checks using dependency graph
KB-Reducer	Checks prior to performing redundancy and ambivalence checking (circularity needs to be removed before the later checks can be applied)
EVA	Checks by computing derivation traces for predicates, and checking to see if a predicate is in its own derivation
COVER	Checks as part of redundancy/ambivalence detection procedure

Table 7 Comparison of the capabilities of five expert system verification tools for checking deficiency in rule bases

Tool	*Missing rules*	*Missing values*
RCP	Checks by decision table-based algorithm	Subsumed by check for missing rules
CHECK	Does not check	Checks
KB-Reducer	Does not check	Does not check
EVA	Does not check	Checks
COVER	Checks by heuristic search, based on assumption sets from ATMS-inspired redundancy and ambivalence checking procedure	Checks as an inexpensive test for deficiency

anomalies is scarce at present (although see Preece 1990; Preece et al. 1992 for some early results), and much experimental research is required to gain information on the value of certain anomalies *versus* the cost of detecting them.

Of course, there is no getting away from the fact that, in the worst case, full verification of redundancy, ambivalence, circularity, and deficiency is intractable. This is necessarily true for knowledge bases based on full first-order logic, because of the problem of undecidability (Jackson et al., 1989), but it will also be true for sufficiently complex systems built using propositional logic languages, because of the exponential complexity of the checking algorithms. This problem has been addressed in two ways: by providing methods for modularizing and partitioning knowledge bases so as to check the less complex components separately (Jacob & Froscher, 1990; Preece et al., 1992); by using heuristics to focus the search for likely anomalies (Preece, 1989; Preece et al., 1992; Laurent & Ayel, 1989). Initial results are promising, and this would seem to be a fertile area for future research.

Applicability of the tools As we stated at the start of this paper, we have restricted our analysis to the knowledge bases with syntax and semantics based on first-order logic. We primarily consider rule-based systems, but the analysis can be extended in principle to frame- and semantic-net knowledge representations, provided that the syntax and semantics of these can be mapped to first-order logic (for example, see Preece & Shinghal, 1991). Similarly, although we restrict ourselves in this paper to systems with monotonic inference (primarily because this is the area in which most verification work has been done to date), some preliminary research has been done in verifying nonmonotonic systems using circumscription-based schemes (Chang et al., 1990 b). Finally, our assumption that our knowledge bases do not include certainty factors is required because of the ill-defined semantics of such schemes. However, some work has been done in this area also (Perkins et al., 1989).

There are other questions regarding the applicability of the techniques described in this paper under inference strategies other than those explicitly considered. For example, a reported case of circularity will never cause a problem in most OPS5-like systems because the *refraction* property in their recognize-act inference cycle prevents looping from occurring (Giarratano & Riley, 1989). However, provided that the tool user understands the assumptions made by the tool, the tool can still provide many useful analyses in situations where the inference engine does not conform to the assumptions. We have addressed this issue in this paper in two ways: first, by stating definitions of the anomalies declaratively in terms of logic, rather than in terms of *ad hoc* inference strategies; and second, by explicity stating the assumptions made by each tool surveyed.

It is important to realize that the real issue here is as follows: we can apply logical analysis to only those knowledge bases which have a well-defined declarative semantics. Where knowledge bases mix declarative and procedural knowledge using *ad hoc* representational formalisms, there is little that can be done to apply verification techniques (Bundy, 1988). We therefore recommend that procedural and declarative knowledge be clearly separated in all knowledge bases. One approach to doing this is to maintain a "knowledge level" description of the knowledge base (that is, a *conceptual model* (Newell, 1981; Wielinga & Schreiber, 1989)), separate from the "implementation level" system. The former is a "clean" declarative description of the knowledge, and is fully verifiable in principle, while the latter is designed to be used in problem-solving, and may be a mix of procedural and declarative constructs. Although we do not discuss these issues at length here, they are considered (with examples) in Batarekh et al. (1991).

State of the art In terms of current work, EVA and COVER are in use as research prototypes, and KB-Reducer has become part of an integrated knowledge base refinement system (Ginsberg, 1990) rather than being used as an independent verification tool. Where other systems have been developed which are similar to the surveyed tools, we have cited the appropriate papers in the text. At present, few commercial tools include verification facilities, and those that do exist are largely restricted to simple checks of the kind performed by the RCP. As new and more sophisticated verification systems appear (including, for example, deliverables from the ESPRIT VALID (1988)

project), we hope that the framework we present in this paper will help in assessing their strengths and weaknesses relative to the systems we survey here.

Acknowledgements

We thank the three anonymous referees for their careful and helpful reviews.

References

Batarekh, A, Preece, AD, Bennett, A and Grogono, P, 1991, "Specifying an expert system" *Expert Systems with Applications* **2**(4) 285–303.

Bundy, A, 1988, "How to improve the reliability of expert systems" In: DS Moralee (ed.), *Research and Development in Expert Systems IV: Proc. Expert Systems 87*, pp 3–17, Cambridge University Press.

Chang, CL, Combs, JB and Stachowitz, RA, 1990 a, "A report on the expert systems validation associate (EVA)" *Expert Systems with Applications* **1**(3) 217–230.

Chang, CL, Stachowitz, RA and Combs, JB, 1990 b, "Validation of nonmonotonic knowledge-based systems" In: A Dollas, WT Tsai and NG Bourbakis (eds.), *Proc. 2nd International Conference on Tools for Artificial Intelligence (TAI-90)*, pp 776–782. IEEE.

Cragun, BJ and Steudel, HJ, 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.

de Kleer, J, 1986, "An assumption-based TMS" *Artificial Intelligence* **28**(2) 127–162.

Giarratano, J and Riley, G, 1989, *Expert Systems: Principles and Programming* PWS-Kent, New York.

Ginsberg, A, 1988 "Knowledge-base reduction: A new approach to checking knowledge bases for inconsistency & redundancy" In: *Proc. 7th National Conference on Artificial Intelligence (AAAI 88)*, volume 2, pp 585–589.

Ginsberg, A, 1990, "Theory reduction, theory revision, and retranslation" In: *Proc. 8th National Conference on Artificial Intelligence (AAAI 90)*, pp 777–782, MIT Press.

Ginsberg, A and Williamson, K, 1989, *Checking quasi-first-order-logic rule-based systems for inconsistency and redundancy* Technical Report 11354-891229-02TM, AT&T Bell Laboratories, Holmdel, NJ.

Jackson, P, Reichgelt, H and van Harmelin, F, 1989, *Logic-based Knowledge Representation* MIT Press.

Jacob, RJK and Froscher, JN, 1990, "A software engineering methodology for rule-based systems" *IEEE Transactions on Knowledge and Data Engineering* **2**(2) 173–189.

Laurent, JP 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*. IJCAI.

McGuire, JG, 1990, "Uncovering redundancy and rule-inconsistency in knowledge bases via deduction" In: *Proc. 5th Annual Conference on Computer Assurance: Systems Integrity, Software Safety, and Process Safety* IEEE.

Mendelson, E, 1979, *Introduction to Mathematical Logic*. Van Nostrand.

Nazareth, DL, 1989, "Issues in the verification of knowledge in rule-based systems" *International Journal of Man-Machine Studies* **30**(3) 255–271.

Newell, A, 1981, "The knowledge level" *AI Magazine* **2**(2) 1–20.

Nguyen, TA, 1987, "Verifying consistency of production systems" In: *Proc. 3rd Conference on Artificial Intelligence Applications*, pp 4–8, IEEE.

Nguyen, TA, Perkins, WA, Laffey, TJ 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)*, volume 1, pp 375–378, AAAI.

Nguyen, TA, Perkins, WA, Laffey TJ and Pecora, D, 1987, "Knowledge base verification" *AI Magazine* **8**(2) 69–75.

Perkins, WA, Laffey, TJ, Pecora, D and Nguyen, TA, 1989, "Knowledge base verification" In: G Guida and C Tasso (eds.), *Topics in Expert System Design*, pp 353–376, North-Holland.

Preece, AD, 1989, "Verification of rule-based expert systems in wide domains" In: N Shadbolt (ed.), *Research and Development in Expert Systems VI: Proc. Expert Systems 89*, pp 66–77, Cambridge University Press.

Preece, AD, 1990, "Towards a methodology for evaluating expert systems" *Expert Systems* **7**(4) 215–223.

Preece, AD and Shinghal, R, 1991, "Practical approach to knowledge base verification" In: M, Trivedi (ed.), *Proc. Applications of Artificial Intelligence IX*, pp 608–619, SPIE, Bellingham WA.

Preece, AD, Shinghal, R and Batarekh, A, 1992, "Verifying expert systems: a logical framework and a practical tool" *Expert Systems with Applications* **4**(2/3).

Puuronen, S, 1987, "A tabular rule-checking method" In: *Proc. 7th International Workshop on Expert Systems and their Applications*, pp 257–268, Paris-La Défense. Agence Inf.

Rousset, MC, 1988, "On the consistency of knowledge bases: the COVADIS system" *Computational Intelligence* **4**(2) 166–170. (Also in *ECAI 88, Proc. European Conference on AI* Munich, August 1–5 1988, pp 79–84.)

Stachowitz, RA and Combs, JB, 1987, "Validation of expert systems" In: *Proc. 20th Annual Hawaii International Conference on System Sciences*, volume 1, pp 686–695.

Stachowitz, RA, Combs, JB and Chang, CL, 1987, "Validation of knowledge-based systems" In: *Proc. 2nd AIAA/NASA/USAF Symposium on Automation, Robotics and Advanced Computing for the National Space Program*, pp 1–10. Report No. AIAA-87-1685.

Suwa, M, Scott, AC and Shortliffe, EH, 1982, "An approach to verifying completeness and consistency in a rule-based expert system" *AI Magazine* **3**(4) 16–21.

VALID, 1988, Validation methods and tools for knowledge-based systems. Deliverable, ESPRIT II VALID Project 2148.

Weilinga, B and Schreiber, G, 1989, "Future directions in knowledge acquisition" In: N Shadbolt (ed.), *Research and Development in Expert Systems VI: Proc. Expert Systems 89*, pp 288–301, Cambridge University Press.