Validation of Knowledge-Based Systems: The State-of-the-Art in North America

Alun D Preece

1 Historical Perspective

Validation provides evidence upon which users decide whether or not a knowledge-based system (KBS) is reliable. Thus, validation ultimately determines the success or failure of a KBS project. Validation techniques include two classes of test: verification tests yield boolean results (for example, whether the knowledge base (KB) of the system is consistent, according to some formal definition of consistency); evaluation tests do not yield boolean results – their results are subject to interpretation (for example, whether a system performs "acceptably", given that a certain percentage of test cases were processed as expected by the system).[1] The importance of validation techniques has lead to a great deal of activity in the research and development of these techniques.

Before 1982, the field of KBS validation focused upon pioneering expert systems projects, including MYCIN and XCON [Gaschnig:83]. Typically, validation involved running a set of test cases on a prototype system, and evaluating the output of the system. The chief issues arising during this period were: what standard should an expert system be validated against, and what constitutes an adequate set of test cases? These issues are still being investigated today; nevertheless, one conclusion quickly emerged: that validation is a hard problem, and that software tools were required to assist as much as possible in conducting verification and evaluation.

In 1982, the Stanford Heuristic Programming Project (HPP) developed what is generally believed to be the first verification tool for KBS: the Rule Checking Program (RCP) for the ONCOCIN expert system [Suwa:82]. The important feature of the RCP was that it checked *domain independent* properties of a KB: redundant, conflicting and missing rules. These properties are *anomalies*: they do not necessarily signify faults in a particular system – they may be intentional or harmless – but they are suspicious and require careful examination. The RCP prompted the start of a great deal of work in the area

[1]These definitions are in accordance with those presented in [Laurent:92].

of KB verification by *anomaly detection*, leading to the development of the following major North American verification tools:

- CHECK, ARC, and EVA, developed by Lockheed Corporation, 1985–1990 [Nguyen:85, Nguyen:87, Chang:90b].
- KB-Reducer (KBR) versions 1–3, developed by AT&T Bell Laboratories and Boeing Corporation, 1987–1993 [Ginsberg:88, Ginsberg:93, Dahl:93].
- COVER, developed by Concordia University, Montréal, and Bell Canada, 1989–1992 [Preece:92c].

The above list is not intended to be complete. It is confined to those systems which have either been used industrially, or had a significant impact on other work. In particular, KBR and COVER tools have demonstrated themselves to be capable of detecting faults in fielded industrial applications.

The anomaly-detection tools consider only the declarative semantics of the knowledge; typically, the KB is modelled as a theory in an appropriate logic, from which the desired properties such as the existence (or non-existence) of conflicts, redundancy, circular dependency, and deficiency can be derived formally. Recently, there has been recognition that many KBS are highly procedural in nature. Partly, this can be attributed to the success of the OPS5-like production rule languages, which support a procedural style of programming, attractive for the development of task-oriented KBS applications [Giarratano:89]. The declarative semantics of these systems are too approximate for anomaly detection to be very effective.

In view of this, a number of approaches have recently been proposed for verifying production rule-based systems, based upon their procedural semantics. Success has been achieved using techniques adapted from other types of software development, including:

- Washington State University's adaptation of SWARM, originally developed for verifying concurrent software [Gamble:91].
- SRI's adaptation of their EHDM and SNARK theorem-provers for verification of formal specifications [Rushby:89, Waldinger:91].
- Honeywell Inc's adaptation of Dijkstra's program verification techniques [Wood:90].
- IBM's adaptation of their "Cleanroom" methodology [Highland:92].

It should be noted that, of the above, only the last demonstrates success on an industrial-scale application – the others have been applied only to small "toy" systems.

Since 1982, the majority of activity in the KBS validation field has concerned formal verification. Less work has been done in the areas of testing and evaluation. Prior to 1987, the main body of work here involved the development of statistical techniques for quantifying the performance of a KBS [O'Keefe:93, O'Keefe:87, Reggia:85]. Testing techniques have received a relatively small amount of attention until recently, when there has been a number of attempts to apply testing techniques from software engineering to KBS [Kiper:92, Preece:94, Rushby:90, Kirani:92]. Evaluation methodology, concerning the design of principled validation experiments, has also been a topic of concern among the empirical validation research community [Adelman:91, O'Keefe:93].

In recent work in the validation area, two significant trends seem to be emerging. Firstly, a number of efforts are being directed at evaluating the validation techniques themselves. In a project funded by the U.S. Nuclear Regulatory Commission (NRC) and the Electrical Power research Institute (EPRI), SAIC Inc. has evaluated the applicability of 134 validation techniques from conventional software to KBS [Miller:93a]. SAIC has also conducted an experiment to compare the effectiveness of a number of different KBS verification tools and techniques [Miller:93b]. Additional data on the relative effectiveness of testing techniques for KBS has been obtained in experiments conducted by the Universities of Minnesota and Penn State [Kirani:92, Zualkernan:93]. Data upon the utility of verification by anomaly detection has been obtained by Concordia University, Montréal [Preece:92a]. Given more studies of this kind, it should be possible to build a clear picture of the relative power of the various validation techniques that have been developed to date.

The second trend in the North American validation field is emerging in direct response to the evolving nature of KBS themselves: increasingly, KBS are being constructed, not as stand-alone systems, but as part of hybrid, heterogeneous software systems. Modern complex software applications often mix a number of software technologies, including knowledge-based, object-oriented, connectionist, case-based, database, and conventional imperative (often legacy code) software. As an added complication, a growing number of systems employ a distributed architecture. Consequently, the North American community is beginning to investigate the validation implications of mixing these technologies. Some example efforts in this direction are as follows:

- Object-oriented and frame-based systems: SAIC Inc. (under investigation) and [Lee:93].
- Distributed systems: [O'Leary:93].
- Heterogeneous software architectures: [Landauer:93].

2 KBS Verification

2.1 Theory of KB Verification

We define four types of anomaly which are likely to indicate actual errors in knowledge bases. For the moment, we will consider knowledge bases that are composed of rules and facts (rule bases). In the following definitions, an *environment* is a set of input data supplied to the knowledge-based system; a *final hypothesis* is a conclusion from the knowledge-based system; a set of data or hypotheses is declared to be *impermissible* if the domain experts say that the set represents a situation that cannot occur in the real world. In the examples, rules are written in first order predicate calculus, *x* is a variable, and *a* is a constant.

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. The most general case of this is a redundant rule: to illustrate, for a knowledge base with *n* rules, I_1 … I_n, rule I_j is redundant if it is a logical consequence of rules I_1 … I_{j-1}, I_{j+1} … I_n. For example, the third rule is redundant below:

```
PROFESSOR(x) ∅ HAS_PHD_DEGREE(x)
HAS_PHD_DEGREE(x) ∅ GRADUATE(x)
PROFESSOR(x) ∅ GRADUATE(x)
```

Ambivalence A knowledge base is *ambivalent* iff for some permissible environment, we can infer an impermissible set of hypotheses. For example, assume a permissible environment is {TEACHING_ASSISTANT(*a*), ENROLLED(*a*)} and we are given the impermissible set {STAFF(*x*), STUDENT(*x*)}. Then the rules below would allow us to infer {STAFF(*a*), STUDENT(*a*)}, which being an instance of an impermissible set is also considered to be impermissible.

```
TEACHING_ASSISTANT(x) ∅ STAFF(x)
TEACHING_ASSISTANT(x)  ENROLLED(x) ∅ STUDENT(x)
```

Circularity A knowledge base has *circularity* iff it contains some set of rules such that a loop could occur when the rules are fired. A common example of this problem would be a relation explicitly defined in both directions:

```
HAS_DEGREE(x) ∅ GRADUATE(x)
GRADUATE(x) ∅ HAS_DEGREE(x)
```

Deficiency A knowledge base is *deficient* iff there exists a permissible environment for which a hypothesis that should be inferred is not inferred. Deficiency is typically due to missing knowledge. For example, consider the following knowledge base, in which R(*a*) is a final hypothesis, and where P(*a*) and Q(*a*) are input data:

```
P(a) ∅ R(a)
¬P(a)  Q(a) ∅ ¬R(a)
```

If either of {¬`P`(a), ¬`Q`(a)} or {¬`Q`(a)} are permissible environments, then we cannot infer any hypotheses for these cases. Therefore, some knowledge is missing.

2.2 KB Verification

In terms of their impact upon the literature and in practical use, the most influential verification tools have been the ONCOCIN RCP, the Lockheed tools (CHECK, ARC and EVA), KBR, and the COVER tool. We focus here upon the more recent tools (EVA, KBR, and COVER), since these have subsumed the capabilities of the older tools (RCP, CHECK and ARC).

2.2.1 EVA: The Expert System Validation Associate

Building upon earlier Lockheed efforts in the verification area [Nguyen:85, Nguyen:87], the scope of the EVA project (1986–93) was ambitious: to build an integrated set of tools to check the redundancy, consistency, completeness, and correctness of any KBS written in any KBS language [Chang:90b]. To achieve this objective, the EVA developers needed to accomplish the following tasks:

- Develop a set of operational definitions for the concepts redundancy, consistency, completeness, and correctness, such that these definitions would apply to any KBS.
- Develop a canonical knowledge representation formalism, into which target KBS applications could be converted, and upon which the various verification checks could be performed.
- Implement efficient software tools to perform the desired checks on the canonical knowledge representation.

Not surprisingly, the EVA project did not solve all these problems in general (they are still unsolved). Nevertheless, it did achieve its goals in a limited manner. First-order logic was chosen as the canonical knowledge representation formalism, and Prolog was chosen as the implementation platform for the toolset. An extensive set of definitions were drawn up for the properties redundancy, consistency, completeness, and correctness, with many special cases. In doing this, the EVA developers were probably the first to acknowledge the role played by metaknowledge in the verification process. Each of the logic checkers in EVA has two forms: basic and extended. The basic form needs no metaknowledge but is severely restricted in power; for example, the basic check for conflicting rules will detect that instances of the terms `MALE`(x) and ¬`MALE`(x) can be asserted simultaneously from consistent input data. The extended form uses metaknowledge to extend the checking capabilities; for example, the extended check for conflict could use the metaknowledge that `MALE`(x) and `PREGNANT`(x) are incompatible, and detect potential simultaneous assertions of instances of these terms.

The definitions of KB anomalies used by EVA cover both rule-based knowledge and the declarative aspects of frame-based knowledge. Unfortunately, the definitions provided in the literature for the properties detected by EVA have not been formalised completely in first-order logic. This has lead to criticisms of the EVA approach as being ad hoc [Rushby:88].

A prototype Prolog implementation of EVA was developed, which uses the Prolog theorem-proving mechanism to check for the defined properties. An efficient, resource-constrained version of the checking algorithm was also developed as part of the off-shoot DEVA project, with funding from the U.S. DARPA agency. The EVA kernel was extended to allow it to check a number of popular North American expert system languages, including ART, KEE and CLIPS (an OPS5-like language) [Childress:91]. Only the declarative features of these languages were captured by the EVA canonical form, however. Currently, Lockheed is not developing EVA further, and it is not in use within or outside the corporation. One of the main reasons for this is that the Prolog platform lacked portability; the possibility of an external company reimplementing the toolset in C is under consideration at present.

Redundancy Checking in EVA Redundancy is checked in EVA by theorem proving. 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, asserting 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 clearly redundant.

The above redundancy check is 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 propositions p and s are declared to be synonyms, EVA will detect redundancy in the following rules:

$p \varnothing q$
$q \varnothing r$
$s \varnothing r$

Ambivalence Checking in EVA In EVA one declares impermissible sets of literals, $L_1 \ldots L_n$ by means of a metapredicate declaration `INCOMPATIBLE`($L_1 \ldots L_n$). For every declaration `INCOMPATIBLE`($L_1 \ldots L_n$), the EVA theorem prover tries to prove the goal $L_1 \quad \ldots \quad L_n$. This proof is conducted by backward-chaining, and when the theorem prover encounters a primitive input, it assumes that the input is given. A proof tree so constructed for the goal is called a *proof residue*, and states what combination of inputs would actually be required in order to prove the goal. If the theorem prover succeeds in finding one or more residual proof trees for the goal, then there is ambivalence in the knowledge base. Further details of this method are provided in [McGuire:90]. 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.

Circularity Checking in EVA For each predicate in a knowledge base represented in the 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. The circularity check in EVA is performed by the structure checker.

Deficiency Checking in EVA As in the earlier CHECK system, the deficiency check in EVA is restricted to detecting some of the symptoms of deficiency, rather than missing rules in general. In the papers on EVA, only a check for missing values is explicitly mentioned, although it is hinted that other symptoms of deficiency are detected. The check for missing values is performed by the EVA *omission checker*.

Other Pertinent Features of EVA EVA features a *semantic checker*, which is used to prove that invariant conditions will never be violated by the knowledge base. The specifications of the invariant conditions make use of a range of metapredicates, including the `INCOMPATIBLE` declarations mentioned earlier.

The EVA *uncertainty checker* uses semantic constraints (declared via metapredicates) to check the validity of certainty factors in a limited manner. For example, it is able to detect if combinations of uncertain evidence will lead to 'fuzzy' contradictions (such as the inferring of conclusions like *risk is high* (80%), *risk is low* (90%)).

Recent work on EVA considers the design of a checker for nonmonotonic knowledge bases [Chang:90a].

2.2.2 KB-Reducer versions 1–3

The first version of KB-Reducer (KBR1) was developed by Allen Ginsberg at AT&T Bell Labs in 1987. The tool was motivated by two requirements: to provide automatic consistency and redundancy checking of knowledge bases during maintenance in the field, and to provide an appropriate "compiled" form for knowledge bases, as a basis for automatic KB refinement [Ginsberg:90]. KBR1 was developed to check rule-based systems written in propositional logic. The KBR algorithm is based upon the assumption-based truth maintenance system (ATMS) developed by de Kleer [deKleer:86], which finds the set 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.

KBR1 is designed for rule languages based on propositional logic, featuring conventional OAV triples for data representation. For KB-Reducer to produce meaningful results, the inference engine needs to be monotonic, non-selective (any rule whose antecedent is satisfied can immediately fire – no conflict resolution strategy is used), 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, KBR 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 only occur 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:

$a \Delta b \varnothing p$
$c \varnothing q$
$p \quad q \varnothing r$
$\neg r \quad d \varnothing \neg p$

Here, a, b, c, and, d are findings, p, q, r, and $\neg p$ are hypotheses, and $\neg r$ is a default hypothesis. The label for p is $a \Delta b$, the label for r is $(a \quad c) \Delta (b \quad c)$, and the label for $\neg r$ is $(\neg a \quad \neg b) \Delta \neg c$.

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, $\neg a$ is assumed true if a is not given as an input to the knowledge base).

The KB-Reduction algorithm requires that a knowledge base can 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 the level above 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.

The KB-Reduction Algorithm The 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 unfirable rules is done immediately after computing each rule label. If the label consists entirely of impermissible environments, then the rule is unfirable. (Note that the user of KB-Reducer is able to specify impermissible environments, called semantic constraints in Ginsberg's paper [Ginsberg:88].)

Once all labels have been computed, KB-Reducer checks for redundant rules 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:

$p \varnothing q$

$q \varnothing r$
$p \varnothing r$

The label for q is p (from the first rule), and for r is p (from either the second or last rules). The first rule is non-redundant, because it uniquely determines an environment in a label (for q). 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 redundant rules is described in a later paper of Ginsberg's [Ginsberg:93]. The original KB-Reducer redundancy detection procedure, described in [Ginsberg:88] was incorrect, as it was sensitive to the order in which rule labels were computed.

The Ambivalence Check The ambivalence check is applied immediately following the redundancy 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 (this includes complementary literals, and hypotheses which are specified by the user as conflicting, by means of semantic constraints). If there is an environment in the partial label of one hypothesis which is either a subset or a superset of an environment in the partial label of the 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 their 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, then both conflicting hypotheses will be inferred.

Performance Considerations for KB-Reducer KB-Reducer is implemented in LISP, and takes 10 cpu hours to fully check a 570 rule knowledge base (running on a LISP workstation), generating 35, 000 environments to do so. This is 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 proportional to the number of environments that it must generate in order 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 only occurs 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 larger than any hand-crafted system built to date. Therefore, Ginsberg concludes that this worst case will rarely occur. A similar argument is presented by de Kleer in support of the ATMS, and by Preece [Preece:93] in support of the COVER deficiency checking method.

Subsequent Development of KB-Reducer KBR2 was developed in 1990, in collaboration with Boeing, extending KBR1 to check knowledge bases written in a subset of first-order logic [Ginsberg:93]. In 1992, KBR3 was developed at Boeing to check knowledge bases containing equations [Dahl:93]. In addition to the technical results of the KBR project, a number of practical results have been obtained. KBR3 is in use at Boeing, and seven of their KBS have been analysed using the tool to date. (Perhaps ironically, KBR was never used operationally at AT&T, where its initial development took place). A variety of faults were detected in these systems, illustrating the utility and practicality of this type of verification tool. Two specific observations are worth noting here: first, despite the fact that, in theory, the KBR algorithm is intractable [Ginsberg:88], none of the Boeing applications verified so far have exhibited the worst-case complexity [Dahl:93]; second, despite the first observation, the tool is very expensive to run, and the developers note that an incremental approach to verification would render the tool more practical. One techniques for incremental verification is offered by the COVER tool, described below.

2.2.3 COVER: COmprehensive VERifier

The first version of COVER was developed in the U.K., and used to check a medical KBS application [Preece:90]. Subsequent development was carried out at Concordia University, Montréal, in the context of a research contract with Bell Canada Inc. Like EVA, COVER is Prolog-based, and performs verification using a goal-directed theorem-proving strategy. Knowledge bases are first converted semi-automatically into the Prolog-based knowledge representation used by COVER; converters have been developed for four representation languages to date, but the COVER representation is capable of capturing only declarative aspects of the KB.

One of the goal's of Bell Canada's expert systems evaluation project was to define *formally* a comprehensive set of KB anomalies. These formal definitions were used as a basis for the checks performed by COVER [Preece:92c], and as a basis of comparing several of the North American verification tools [Preece:92b].

An important part of the development of COVER has been to apply it to a diverse collection of KBS applications. The tool has been used to find faults in KBS developed by Bell Canada, NASA, 3M Corporation, and the U.K. health services [Preece:92a], to solve a variety of tasks including diagnosis and planning. Study of these verification efforts has yielded data on the relative utility of the various verification checks. In addition to confirming the value of this approach to verification, these results suggest that the most effective checks are the simplest: the integrity checks were found to detect the highest percentage of faults in the systems surveyed.

The COVER verification operations are divided into three groups, related by the methods used to implement them: *integrity checking operations, rule checking operations*, and *rule-chain checking operations*. The reason for this separation is purely pragmatic: the computational cost of performing the three categories of checks is very different in each case; moreover, integrity anomalies which represent actual errors will usually result in rule and chain anomalies, while rule anomalies will certainly result in rule chain anomalies. Separating each category affords more flexibility to the user of COVER, and can make the root cause of anomalies much clearer. For example, if there are two duplicate rules in a system, and COVER is asked to check first for rule chain anomalies, then there will be redundancy in every inference chain which uses the consequent of the duplicated rule, as the following example shows:

$p \varnothing q$ (1)
$q \varnothing r$ (2)
$q \varnothing r$ (3)

Here, the duplicate rules (2 and 3) result in the reporting of the redundant inference chains {1, 2} and {1, 3}. In a real system, there could be many anomalous inference chains reported because of a few rule pair problems; by running the rule checker before the rule chain checker, the user will be able to detect and correct problems with pairs of rules before running the expensive inference chain checking procedure.

Implementation of COVER The COVER program first performs syntax checking on the rule base, as it is read in. 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 clausal form. Also, the conditions in the conjunctions are ordered at this stage, to make subsequent checks more efficient. The original rule identifiers are preserved in this transformation, so that anomalies can be reported to the user when found.

The integrity checker builds a number of important cache tables, including a table which references all data items against the values they can legally take (either by firing a rule or by asking the user). Building this table is linked with the check for unobtainable data items, unused askable declarations, and useless rules. Once complete, the table is used in the check for unsatisfiable conditions, dead-end rules and missing values. The performance of the integrity check is $O(n)$, where n is the number of rules in the rule base.

The rule checker compares each pair of rule antecedents and consequents. This is efficiently implemented because all antecedents are conjunctions and their conditions were ordered during syntax checking. This performance of this check is $O(n^2)$.

The implementation of the rule chain checker is divided into three sub-procedures: creating goal environments, checking goal environments for redundancy and ambivalence, and checking goal environments for deficiency. The user must first select which goal, or set of goals, are to be checked. This could be the top-level goal of the entire knowledge base, or a lower-level goal, depending on the modularity of the system, and the extent to which the user wishes to apply the checking at that particular time. This feature is included to give the user additional control over the checking process, and is purely a pragmatic consideration. After the goal or goal set has been selected by the user, COVER uses a meta-interpreter similar to a backward-chaining inference engine to establish the set of all environments which lead to the establishing of some value for the goal. Each such environment is called a *goal environment* in COVER. The number of goal environments e for a particular goal g is dependent on the number of rules s which have a particular data item as their subject, and the length of inference chains in the knowledge base c. e is proportional to s^c. Performance of the procedure used to establish goal environments is $O(e)$.

Note that the procedure used to establish goal environments by COVER is similar to that used to establish rule labels by KB-Reducer, except that COVER uses a goal-directed strategy (because the goal set is pre-selected by the user) and KB-Reducer uses a data-driven procedure (more specifically, it is finding-driven). Goal environments in COVER are equivalent to rule labels in KB-Reducer for rules which have a goal in the selected goal set as their subject. Note also that, during the procedure for finding goal environments, COVER automatically discovers any cyclic inference chains. These need to be removed for goal environments to be created properly – in this requirement, COVER is equivalent to KB-Reducer.

Having established the goal environments, COVER checks them for redundancy and ambivalence, using the same comparison procedures as those used for the rule antecedents in the rule check. This works because each goal environment can be considered to be a conjunction of literals, similar to rule antecedents in their normalized form. The performance of this check is $O(e^2)$.

To check deficiency in the set of goal environments, COVER uses a novel method to control the combinatorial explosion which would otherwise render the problem intractable [Preece:93]. Instead of generating and testing every combination of data items and values present in the set of goal environments, COVER uses the information in the existing rules to determine which data items are *relevant* to one another, and restricts its generation of environments to only these 'plausible' cases. Also, COVER generates environments in order of size, and if it finds small missing cases, it need not generate environments which subsume these.

The limitations of the COVER deficiency checking procedure lie in the assumptions made by the two heuristics. The first, that of only considering relevant combinations of data items, 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 data items in the rule, then COVER may not detect missing rules which use a combination of these items. However, this is really a fundamental limitation of all anomaly checking programs: they can only work by syntactic inspection of the existing knowledge base; they have no access to semantic knowledge about the real world. The philosophy behind the first heuristic used by COVER is that it is more reasonable to use information about the relationships between data items as indicated in the existing knowledge base than to make the assumption that any set of data items may be relevant to one another, which can lead to the reporting of an unnecessarily large number of semantically-meaningless rules.

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 data items and values will be found by COVER, it also results in *minimal* deficient cases being reported, unlike the methods of RCP and ESC, which report missing cases including a value for every data item (parameter) present in the table. It is possible to criticise COVER on the grounds that the minimal combinations may omit relevant item/value pairs, however, it is equally possible to criticise the other approaches on the grounds that they often include unnecessary item/value pairs. The question here is whether it is more reasonable to present minimal cases to the knowledge engineer and expert(s), who can then decide whether additional constraints are required on any rules which may be added to fill the gaps, or whether the users should be expected to prune unnecessary conditions.

Concluding Note The data obtained from use of declarative verification tools such as COVER and KBR reveal a subtle relationship between anomalies and faults in a KBS. Anomalies are merely symptoms of probable faults: when an anomaly is detected, some effort is usually required decide if it indicates a fault and, if so, what that fault is. Not every anomaly indicates a fault. The main reason for this phenomenon is that few realistic KBS applications have a pure declarative semantics; many of these systems have some procedural aspect, which interferes with the declarative semantics [Preece:92a]. The following approaches address this problem by treating a KBS as a procedural system.

2.3 Procedural Verification

Many KBS operate in a procedural manner, performing tasks defined imperatively. This is especially true of systems using production rules, which use a global data structure, *working memory*, to represent the current state of problem-solving. The state of working memory at any given time determines which rules are able to fire at that time; when a rule fires, its "actions" are reflected by a change in the state of working memory. Procedural verification techniques exploit the procedural semantics of this type of KBS in order to prove properties concerning the reachability or non-reachability of certain working memory states from valid initial states. For example, it is possible to prove that no state can be reached which will violate pre-defined *invariants*, such as the `MALE`(x) and `PREGNANT`(x) semantic constraint described above.

In this spirit, a number of successful attempts have been made to verify KBS applications [Gamble:91, Rushby:89, Wood:90]. However, all of the systems verified in these attempts have been simple, "toy" problems (such as "tic-tac-toe" [Rushby:89] and the supermarket "bagger" example [Gamble:91, Rushby:88]). To establish the efficacy of this approach to verification, more complex examples must be explored. This is especially necessary since, even with software support, a great deal of effort was required to prove properties of the simple systems.

Experience to date has shown that obtaining formal requirements specifications against which to verify KBS is a serious difficulty in applying the above techniques [Gamble:93]. Unlike the declarative approach, procedural verification cannot be performed without such specifications. (The declarative approach can still find inconsistencies and redundancies in the absence of any formal specifications, since these properties are generic to any logic-based KB.)

In spite of the above limitations, this approach provides a promising means of verifying KBS which operate in a largely procedural manner. Moreover, a clear advantage of this approach is that it provides an integrated means of verifying hybrid software systems which contain both knowledge-based and conventional imperative components. Evidence for this was provided by an IBM project in which a pseudo-procedural KBS was verified using an adaptation of IBM's "cleanroom" validation methodology [Highland:92].

3 KBS Testing

Validation entails testing: the decision as to whether a KBS meets some criterion is based upon evidence obtained by running the system through a set of tests. The results of validation will only be credible when the test-set is sufficiently representative of the domain of problems that the system will be required to solve when in use. The fundamental problem in testing knowledge-based systems is the same as that for conventional software: systems cannot be tested on all input cases because the input domain is intractably large [Rushby:88]. The problem, then, is to choose a reasonable subset of the input domain that provides the maximum assurance of system reliability. The strongest proposal along these lines is *thorough testing* [Goodenough:75], which will find all faults in a program by formulating a set of test cases which are called a *thorough test set*. Formulation of a thorough test set depends on identifying a test selection criterion *C* which guarantees:

a) that all test sets *T* satisfying *C* give the same results when run on program *F* (in this case, we say that *C* is *reliable*); and
b) that if *F* is faulty, then some *T* satisfying *C* will fail when run on *F* (we say that *C* is *valid*).

There are several serious problems with applying this method in practice [Weyuker:80]. The most serious problem is that the difficulty of formulating *C* is equivalent to a formal proof-of-correctness for *C*. In fact, problems with applying the theory of thorough testing led researchers to the technique of *partitioning* the input domain. In the following sections, we consider several ways of doing this.

3.1 Function-Based Partitioning

Validation establishes that the system meets its requirements with respect to functionality; therefore, all such requirements should be specified as precisely as possible during the early stages of development [Batarekh:91].

3.1.1 Knowledge-Based System Requirements

Knowledge-based system requirements can be identified and grouped according to whether they are "AI software technology-based" or "conventional software technology-based". In an influential report, Rushby has observed that, although many important elements of AI software can be only weakly specified at best due to their reliance on human knowledge or skill, many other requirements for these systems are no different in principle from those of conventional software [Rushby:88]. As examples of such *service* requirements, he cites input and output formats, user interface requirements, and processing rate, and he points out that these can and should be specified as precisely and formally as for conventional software. The issues unique to validating knowledge-based systems are concerned with the essentially "AI" aspects of these systems (that is, those utilizing AI techniques), which Rushby terms the *competency* requirements. He proceeds to identify two sub-types of competency requirements: minimum and desired.

The *minimum* requirements seem to have much in common with *safety* properties; that is, they state not only what the system must always do, but also what it must never do. The idea is that, even where it is too hard to specify precisely the optimal solution to a problem-solving task, it may well be feasible to specify a minimally-valid solution. The hope here is that such requirements will be specifiable with greater precision than the desired properties, thus permitting them to be validated with a greater degree of reliability. This in turn would promote acceptance of knowledge-based systems since, whatever else they may or may not do in practice, at least there would be a high degree of assurance that disasters will not follow from their introduction in safety-critical applications.

If we can specify the service requirements and minimum competency requirements with a fair degree of precision, this leaves only the desired competency requirements. Rushby suggests two possibilities in this case: if there is a deep model for the domain (for example, a simulation of an electrical device), it can be used to generate tests against which to compare the knowledge-based system; in the absence of a deep model, comparison with human experts is suggested as the only recourse. Rushby stresses that, in any case, the comparison standard should be identified at the outset of development, as the target for system developers to meet.

3.1.2 Equivalence Class Partitioning

The specification of a knowledge-based system provides a generally specified functional relation $F : \mathbf{I} \; \varnothing \; \mathbf{O}$, where F is the knowledge-based system and $\mathbf{I}$ and $\mathbf{O}$ its input and output domains, respectively. Since it will usually be infeasible to test every input/output pair because of the sheer size of most knowledge-based system input domains, we seek *equivalence classes* which partition the input space into classes of input which are expected to produce similar output. We can then use these to design tests which will determine if the system behaviour on the input classes is acceptable, based on the required output classes. We first identify each equivalence class I_k to be a set of input $I_k \wp \mathbf{I}$, which the system is expected to map to output class $O_j \wp \mathbf{O}$. Then, we select at least one sample input $i_k \quad I_k$ from each equivalence class, and we test that it does indeed map to an output $o_j \quad O_j$. These sample pairs (i_k , o_j) constitute our test cases.

The *revealing subdomain* method [Weyuker:80] from the software engineering literature is an implementation of the equivalence class idea. Intuitively, a revealing subdomain is a class of input the elements of which behave identically: either all produce correct output or none does. There are two stages to defining revealing subdomains: the problem-partitioning and solution-partitioning stages. In the *problem-partitioning* stage, as described in [Weyuker:80], the problem specification is used to identify classes of input that should be treated identically by the system. The *category-partition* method is one technique for doing this in conventional programs [Ostrand:88]. Essentially, this method entails identifying distinct functional units from the program-independent requirements documents, and determining the essential characteristics of the input for each unit. In the *solution-partitioning* stage, the solution specification for the system is studied in order to determine sets of input which will be treated identically by the program. This is related to structural testing of the system, in that it is likely to involve studying paths through the system.

Following these two stages of analysis, the revealing subdomains are defined to be the intersection of the two sets of input classes. That is, a revealing subdomain is a set of inputs which *should* (according to system requirements) and *will* (according to system design) be treated identically by the system. For example, in a medical diagnosis system, a set of input cases (sets of patient symptoms) which indicate a common disease and which would result in the same set of rule firings by the knowledge-based system, would constitute an equivalence class. The definition of equivalence class is equivalent to the definition of a revealing subdomain given previously. Unfortunately, in practice we cannot define a revealing subdomain *I* such that, if the system correctly processes input *i*   *I*, then we can be assured that the system will correctly process all *i*   *I*. Such a formulation would be as hard as a full proof-of-correctness for the program [Weyuker:80]. Instead, the best approach seems to be to define revealing subdomains which reveal the presence of specific errors in the system. We want to be able to say that, if an error *E* is present, and *E* affects the processing of input *i*   *I*, then *I* is defined such that *E* affects all *i*   *I*. Now, if the system correctly processes a single input *i*   *I* we can be assured that *E* is absent; however, if *i* is processed incorrectly then we cannot say if this is due to *E* or to some other unspecified error.

Preliminary evidence for the applicability of the revealing subdomain method of functional testing to knowledge-based systems is presented in [Rushby:90]. A rule-based knowledge-based system is analysed and a complete set of revealing subdomains is distinguished; test cases are generated for each subdomain based on the distinguishing characteristics of each subdomain, and errors are detected in the system. Although the system used is fairly simple, it is a real application. This evidence for the efficacy of the revealing subdomain method is not conclusive, but further research is encouraged.

3.1.3 Risk-Based Partitioning

The *heuristic testing* approach advocated in [Miller:90] provides a practical synthesis of ideas from the approaches described above. The principle in heuristic testing is that system *reliability* is a more important issue than *bugs* in the system. In other words, rather than seeking to eradicate all bugs in a knowledge-based system, we seek assurance that any bugs remaining will be harmless. Assuming that, in a complex knowledge-based system, we will never be sure that we have removed every bug, it is desirable to seek assurance that remaining bugs are unlikely to cause serious harm.

The heuristic testing method is so-named because it identifies fault classes which should be validated in order of priority: the idea is that it is a reasonable heuristic to look for each class of fault in order of seriousness. Miller identifies ten classes; we focus on five of the more interesting ones to give a flavour for the method. The five are listed in the order of priority suggested by Miller:

- **Basic safety** As the name implies, these faults, if present in the system, would have a direct or indirect causal relationship with harmful actions. These actions are identified by structured interview with domain experts, by looking for actions which, if properly or improperly performed or recommended by the knowledge-based system, could lead to harm (an example given is opening an airlock door on a spacecraft when it is not known that all occupants are wearing spacesuits).
- **Essential function** This class can be considered equivalent to those minimum competency requirements not covered by the class of basic safety requirements. They represent a lower bound on system competency not involving basic safety.
- **Robustness-failure** These faults are caused by deviant input conditions. These include familiar problems such as typographical errors, overloaded input, values out-of-bounds or of the wrong type, and so forth.
- **Secondary function** These are faults in non-critical system functions, which would seem to include functionality in the *desired competency* areas, and may also include functions such as user explanation and help facilities.
- **Error-Metric** This class can be regarded as a "safety net" for faults which are missed out from the other fault classes. Metrics can be created based upon statistical analyses of the structural properties of the knowledge-based system, and the existence faults may be hypothesised based on the results of applying the metrics. For example, a metric may measure the complexity of an individual rule based upon the logic therein; a reasonable hypothesis based upon this metric might be that errors are more likely in highly complex rules than in simple ones. The difficulty,

however, lies in finding good metrics. In the example, a single complex rule may be conceptually easier to understand than a logically-equivalent inference chain of simple rules. Metrics in software engineering are based upon a great deal of experience in writing many thousands of programs. It will be several years before the same breadth of experience is available for designing good metrics for knowledge-based systems. In the meantime, some researchers have produced metrics for quantifying knowledge base modularity [Jacob:90, Mehrotra:93], rule base complexity [Chen:93], and the metrics for knowledge-based system search space complexity, discussed in Section 3.2.

One advantage of the heuristic testing approach is that its use is not predicated upon the existence of complete functional specifications for the system. Many of the fault classes can be (at least partially) constructed by interviewing domain experts and/or consulting system code.

3.2 Structure-Based Partitioning

The idea in structure-based testing is to validate the system by exercising as many of its structural components as possible. In conventional software, structural testing is usually based upon viewing imperative programs as flow graphs: a minimal criterion is to exercise every statement (node) in the program (graph); a better criterion is to exercise all outcomes of each decision point (traverse each edge in the graph) at least once. This is called the *minimally thorough* structural testing criterion [Huang:75]. However, it is still too weak to be useful in practice, because many faults will be too subtle to manifest themselves in every execution of a given statement. The context for the execution of the statement, in terms of the state of the program variables at that moment, needs to be accounted for [Rushby:88]. *Path testing* addresses this problem by attempting to execute every path through the program. However, this method is not foolproof, since two different test cases may follow the same path and yet one may reveal an error while the other does not.

The first problem in applying structural testing techniques to knowledge-based systems is to formulate an adequate notion of *path* for these systems. The purpose of structural testing for rule-based knowledge-based systems is to ensure that test cases are run on the rule-based system which exercise each path through the rule base [Rushby:90]. There are three tasks involved:
1) generate paths from the rule base according to certain constraints;
2) create test cases for each path;
3) run the test cases on the rule base.

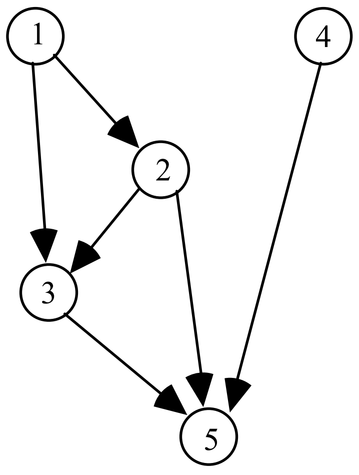
There is a trade-off at the heart of this issue: the weaker the constraints used in path generation, the easier generation will be, but this may result in wasteful testing of invalid paths; the stronger the constraints, the more difficult generation will be, but testing will be more efficient. The very strongest constraints will result in exhaustive generation of all possible valid paths, although this may not be computationally feasible for very complex rule bases. In the following sections, we consider four proposals for path testing of rule-based knowledge-based systems.

3.2.1 EVA approach

The EVA system [Chang:90b] features a structural test case generator based on a dependency graph (DG) of the rule base. The first step in this procedure is to create a connection graph of knowledge base rules. Consider the following simple rule set (based on an example in [Kiper:92]):

$p \varnothing q$ (1)
$q \varnothing r$ (2)
$q \quad r \varnothing s$ (3)
$p \varnothing s$ (4)
$r \quad s \varnothing t$ (5)

In the DG, the rules are prepresented as nodes, and an edge is drawn from node *i* to node *j* if a literal in the consequent of rule *i* unifies with a literal in the antecedent of rule *j*. The DG for the above rule set is shown below.



$$\begin{tikzcd}
1 \arrow[dr] \arrow[dd] & & 4 \arrow[ddd] \\
& 2 \arrow[dl] \arrow[dd] & \\
3 \arrow[dr] & & \\
& 5 &
\end{tikzcd}$$

Note that the presence of an edge from node *i* to node *j* to another does not necessarily mean that firing rule *i* would lead to the firing of rule *j*. As Rushby observes in [Rushby:90], there would be an edge between the following rules, even though firing the first rule would actually prevent the second from firing:

$p \varnothing q \quad \neg r$
$q \quad r \varnothing s$

EVA uses the DG to generate a *rule flow diagram*, in which nodes represent rules and edges represent rule firing sequences. This implies that additional sequencing information needs to be employed, because the DG does not record the fact that more than one rule may *jointly* enable some rules. In the graph above, either rules 2 and 3 or rules 2 and 4 jointly enable rule 5, but this information is not apparent from the graph alone.

EVA creates a set of *paths* from the flow diagram such that each rule appears in at least one path. Test cases are then generated for each path; each of these is a data set which satisfies the conjunction of the antecedents of all rules on the path. These are generated using the constraint solver described in Section 3.3. Working from the example rule set graphed above, EVA would create four paths, as follows: (1, 2, 5), (1, 3, 5), (1, 2, 3, 5), (4, 5). The conjunctive expressions for generating test cases for these are as follows:

(1, 2, 5)	$p \quad q \quad r \quad s \quad t$
(1, 3, 5)	$p \quad q \quad r \quad s \quad t$
(1, 2, 3, 5)	$p \quad q \quad r \quad s \quad t$
(4, 5)	$p \quad r \quad s \quad t$

In actual fact all of the above paths can be exercised with the single input *p*, due to the highly redundant nature of the simple rule set. (Consider the antecedent of rule 5: *p* directly implies *s* from rule 4, and *p* indirectly implies *r* via rules 1 and 2; the combination of rules 1, 2 and 3 is subsumed by rule 4. It is not clear from the published reports whether the EVA method retains enough sequencing information to realise this, but it seems clear that the sequencing information is important in practice.

3.2.2 Kiper's approach

The graphing method proposed by Kiper [Kiper:92] records enabling relations between rules. The meaning of an edge between two nodes *i* and *j* in one of Kiper's graphs is that, as a result of firing *all* the rules on the path leading to *i*, rule *j* is now firable (though whether *j* fires immediately after *i* depends on the conflict resolution strategy employed in practice by the inference engine). The Kiper graph of the earlier set of rules is shown below (this is taken from [Kiper:89]).

$$\begin{tikzcd}
1 \arrow[d] & 4 \arrow[d] \\
2 \arrow[d] & 2 \arrow[d] \\
3 & 5
\end{tikzcd}$$

However, it is not clear from Kiper's method whether this is the only possible graph, or what the precise meaning of the sharing of rule 2 between the two separate chains is. Kiper's paper states the additional condition that an edge from rule *i* to rule *j* means that rule *j* was not firable before rule *i* was fired. This is inconsistent with the example shown above, where the "enabling" of rule 2 by rule 4 in the second chain is puzzling in view of Kiper's additional condition: rule 2 *was* firable before rule 4 fired. To understand this, it is necessary to note that, in the first chain of the graph, rule 1 is seen to enable rule 2. When rule 1 is firable, so is rule 4 (they have the same conditions), but since neither of these rules depends on any other rules, it seems to make sense to draw the graph in the way shown. However, following Kiper's definition (without the additional condition), it would appear to be strictly possible to draw the second chain thus: *rule* 2 ∅ *rule* 4 ∅ *rule* 5 In any case, these graphs will be difficult to compute in practice – a point also observed in [Rushby:90].

3.2.3 Rushby and Crow's approach

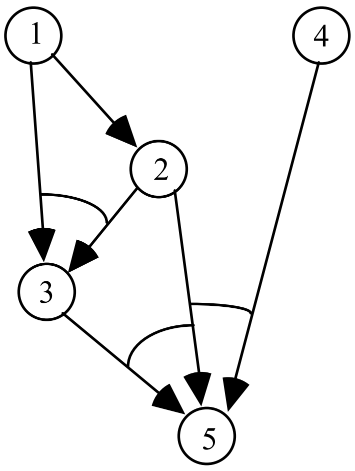
Essentially, this is an extended version of the EVA DG method, described in [Rushby:90]. It is designed to improve over the EVA approach without incurring the extreme computational expense of Kiper's method. Once again, rules are represented as nodes, and the first condition on edge formation is that an edge may be drawn from node *i* to node *j* if a literal in the consequent of rule *i* unifies with one in the antecedent of rule *j*. However, Rushby and Crow place two additional constraints on the drawing of edges:

- The antecedent of rule *j* should at least be logically consistent with the antecedent of rule *i*; as a stronger constraint, the antecedent of rule *j* should be logically consistent with the antecedents of the path of *n* rules terminating with *j* (we set a bound *n* to control the computational difficulty of implementing this constraint).
- Drawing an edge from node *i* to node *j* should not introduce a cycle into the graph.

The first of these constraints is supposed to avoid problems such as the one described under the EVA approach, above:

$$p \varnothing q \quad \neg r$$
$$q \quad r \varnothing s$$

However, this will not be the case, unless we tighten the constraint further: the antecedent of rule *j* must be logically consistent with the *conjunction* of the antecedent and consequent literals of rule *j*. The other improvement offered by this method over the EVA approach is that the graph is post-processed to eliminate paths which do not correspond to potential rule chains: an arc is drawn between edges from nodes which *jointly* satisfy a common successor node. This is illustrated below, using the same ruleset as the earlier examples.



Since many knowledge bases contain legitimate (and even desirable) cycles, the restriction on cycles imposed in Rushby's proposal is rather restrictive. Tools such as COVER use the notion of input items (or *askables*) to break cycles in cycle-detection procedures [Preece:92c]. Once detected, the cycles are reported so that the knowledge base designer can decide if they are legitimate. A path generator must be able to work with cycles, and it appears that the notion of input items would serve adequately here, too.

The other problem with Rushby's proposal is that the details of what constraints are needed to control path generation are rather sketchy. In particular, the post-processing method (for drawing arcs between the edges as illustrated above) is not formalised. Nevertheless, Rushby comes closest to the heart of the trade-off between constraints and computability, and the constraints proposed are certainly towards the weak end of the scale (stronger than used in EVA but weaker than used by Kiper). These ideas beg further investigation.

3.2.4 COVER approach

Verification tools such as COVER can be used to provide rule execution paths. As described in [Preece:92c], COVER attempts to exhaustively compute all execution paths for a rule base, in order to check them for redundancy and conflict. A product of this analysis is the set of environments for each rule base output, associated with the chains of rules that establish the outputs from the environments. For the example rule set used above, COVER would compute the environments and associated rule-chains shown below:

Output:	*Environment*:	*Rule chains*:
q	$\{ p \}$	(1)
r	$\{ p \}$	(1, 2)
s	$\{ p \}$	(4) and (1, 2, 3)
t	$\{ p \}$	(1, 2, 3, 5) and (1, 2, 4, 5)

Not only does this analysis produce the execution paths for the rule base, but also it generates the sets of test cases required to exercise those paths, in the form of environments. Note that COVER would also detect that either rule 3 or rule 4 (but not both) is redundant in this rule set – the redundancy would probably be removed before doing a dynamic analysis, to eliminate testing of unnecessary paths.

This analysis is not unique to COVER: EVA, KB-Reducer [Ginsberg:88] and other inference chain verifiers also perform variants of it. The issue is how practical this procedure is. Although the computational complexity is exponential, and therefore the procedure is intractable in the worst case, empirical evidence produced from trials of COVER and KB-Reducer on substantial real-world knowledge bases suggest that it is perfectly feasible in practice.

As we have seen, there is a trade-off between computability and the strength of the definition of paths in rule bases. If we set strong constraints on path formulation, we face hard computations. If the computations prove intractable, we can relax the constraints, allowing some invalid paths to be generated but permitting the analysis to proceed where it would otherwise have to be abandoned. We need to experiment to determine how such bounds might be determined. We can also study ways to make the analysis more tractable, by designing knowledge bases that are more amenable to testing strategies (by the use of modularization, for example).

Even if we can overcome the problem of computing all paths for a knowledge base, a second significant problem in applying exhaustive path testing to knowledge-based systems is the size of the solution space. For example, the size of the MYCIN search space has been estimated to be about 900 paths (5.5^4) [Buchanan:87], using the metric b^d, where b is the average branching factor (b = 5.5 for MYCIN) and d is the average depth of search (d = 4 in MYCIN). Therefore, about 900 test cases would be required to exhaustively cover this space using structural testing, and every one of these cases would have to be checked for correctness. However, the solution of a real-world problem by MYCIN would involve the tracing of more then one path through the knowledge base, and so the actual complexity is even worse than our simple metric suggests.

A more realistic quantification of search space complexity for a knowledge-based system is presented in [Shwe:89]. Here, the ONCOCIN knowledge base is viewed as an augmented transition network (ATN) and a computation formula is derived for the number of paths through the network. We do not go into the details here, except to observe that as the size of ATNs based on subsets of the ONCOCIN rule base grows from 4 to 29 rules, the lower bound on the number of paths grows from 30 to 8163. This has serious implications for the creation of test cases, as discussed below.

3.3 Test Case Generation Tools

The issue here is whether to obtain the test cases by selecting real past cases or by generating artificial cases. Use of the *selective* approach ideally requires a large number of well-documented historic cases, representing a wide variety of problem types and the experience of different experts [O'Keefe:87]. In some domains, such as telecommunications, such cases are readily available from logs of past sessions [Grogono:91]. However, in other domains, such past cases may be rare or non-existent. In contrast, the *generative* approach involves the creation of synthetic test cases. Although such cases can be constructed manually by experts, to minimise bias it is better to generate them automatically using randomised generation procedures, ideally guided by knowledge about likely cases in the real world [Shwe:89].

Each approach has its advantages and disadvantages. Real cases will probably reflect the actual types and range of problems encountered in practice, and have known correct (or at least acceptable) solutions. However, they may be hard to obtain, especially if the domain is highly esoteric, and may be incomplete, in the sense that many hard or unusual problems are not covered by the available historic cases. Generated cases not only solve the problem of lack of past cases, but also have the advantage that they can be tailored to reflect appropriate problem types and ranges, as well as covering unusual situations. However, human experts will need to be consulted to ensure that realistic cases are generated, and also to provide acceptable solutions for the cases (if these cannot be provided in any other way). Unless several experts are consulted, this may result in a biased set of generated cases.

Given the advantages and disadvantages of each approach, we concur with O'Keefe's suggestion that a synthesis of both types of test case is probably the best approach in general [O'Keefe:87]. This can be achieved by including a good cross-section of real cases, together with synthesised cases to "fill the gaps". In practice, most of the systems described in the literature for generating test cases for knowledge-based systems use a synthesis of techniques from the random, structure-based and function-based approaches. We consider three systems: the Generic Testing Method [Miller:90], the structural and functional test case generators of the EVA tool [Chang:90b], and the domain-specific ScriptGen tool for the ONCOCIN knowledge-based system [Shwe:89]

3.3.1 Generic Testing Method

Miller's Generic Testing Method (GTM), as described in [Miller:90], is specifically designed for generating test cases for rule-based knowledge-based systems. Nevertheless, in theory it is applicable to knowledge-based systems using alternative knowledge representations, and even to conventional software systems. The method is intended to be applied in the context of the heuristic testing strategy: test cases are generated specifically for each of the applicable fault classes, some of which were discussed in Section 3.1. It is assumed that test cases generated using the GTM will be supplied to the system by test driver software, using testing scripts to simulate actual operating conditions (for example, by simulating the interfaces of a knowledge-based system embedded in some other hardware or software system). Note that the GTM is limited to addressing the problem of generating test case *input* data only – the acceptable outputs have to be determined for each case, and some method of checking the correctness of the system outputs must be implemented.

The GTM is a structure-based test case generation method. The basic idea is very simple: that errors in rules are more likely to be subtle than gross. For example, given a rule condition $A < C$, where A is some attribute (data value) and C is a constant, it is more likely that there is a small error in the chosen value of C or a mistaken choice of comparison operator (perhaps it should have been $\leq$ instead of $<$) than a major error in choice of A or magnitude of C. The argument supporting this assumption is that most of the gross errors will have been detected in verification and knowledge base inspection. Therefore, for each specific rule condition, the GTM creates test cases of three kinds, where appropriate: the case where the condition is exactly true; two cases where the condition is "minimally" true and false, respectively; and two cases where the condition is "extremely" true or false. (Here, the terms "minimally true/false" and "extremely true/false" are used somewhat loosely: the difference is that, in the "minimal" case, a slight change in the numeric constant could affect the success or failure of the test, while in the "extreme" case a slight change would have no effect.) Selection of numeric data values in the test cases is determined by knowledge of an appropriate step size for the attribute. Limits for the extreme values may also be determined by relevant knowledge. Non-numeric attributes are treated as a special case of = operator tests, unless their values are ordered, in which case any of the comparison operators may be used and appropriate test cases may be generated.

For rules with multiple conditions, there is an exponential increase in the number of test cases required for exhaustive testing: as we have seen, one condition requires at most five tests, two conditions requires at most 25 tests, three conditions requires at most 125 tests, and n conditions requires at most 5^n tests. Since this rapidly becomes unreasonable in practice, Miller suggests weakened versions of the full GTM strategy, involving one test (the exactly-true or minimally-true case) or three tests (exactly-true and minimally true) for each condition.

The GTM is extended to generate test cases for rule chains by a procedure similar to backward chaining, working backwards through rules whose conclusions satisfy the conditions of the later rules in the chain. All primary inputs discovered along the chain become part of the test case, and their values are assigned using the basic GTM procedure. The places to start the backward-chains (that is, the final conclusions of the chains) are determined by the fault class that the GTM is being employed to detect. For example, to test for *basic safety* faults, test cases need to be generated that lead to conclusions concerned with basic safety. In this sense, the GTM can be considered a *structural* (path) testing method, driven by a *functional* testing strategy (heuristic testing).

3.3.2 EVA Test Case Generator Tools

The Lockheed EVA system [Chang:90b] consists of a broad range of independent knowledge-based system verification and validation tools. Several of the verification facilities were described in detail in Section 2; here, we focus upon two of the validation tools: the *structure-based test case generator* and the *function-based test case generator*. Both of these tools are based upon a *constraint solver* program, written in Prolog. Given a conjunction of literals C, the solver performs three steps:

1) identify the sets of dependent variables X and independent variables Y appearing in C;
2) generate a set of assignments to all variables in X that satisfy the dependencies specified in C;
3) randomly assign values to variables in Y, according to their types and ranges.

Upon backtracking, the constraint solver will generate alternative assignments to variables in X and Y, thereby creating a set of test cases.

The EVA function-based test case generator consists of the constraint solver plus a set of constraints specifying the characteristics of functional test cases. The constraint solver is domain-independent, while the constraints are application-dependent and need to be specified anew for each knowledge-based system validated using EVA. For example, the constraints might be based upon a revealing subdomain analysis, as described in Section 3.1. The constraint solver appears capable of generating both the input and output parts of test cases, if the relationships between input and output can be specified simply enough. However, it seems most likely that, in any realistic domain, only the *minimum output requirements* could be specified in this way, for if it were possible to write a simple conjunctive expression describing the full input-output relation, then the problem would probably be too simple to warrant a knowledge-based system solution. If the input-output relation cannot be specified in the constraint solver, then human experts or a simulation model will have to provide the required outputs as previously discussed.

The EVA structure-based test case generator was examined in Section 3.2. It automatically obtains conjunctive expressions for the constraint solver by finding a path (inference chain) through the rule base that requires testing, and creates a conjunction of input items that satisfy the antecedents of each rule on the path. The process is repeated for each path that needs to be tested. As we saw, there are two problems with this method. Firstly, the definition of *path* is so weak that attempts will be made to test many invalid paths (presumably, in such cases the constraint solver will fail to solve the conjunctions). Secondly, rule enabling relations do not appear to be adequately accounted for by the method, leading to inefficiencies and inaccuracies in test case generation.

3.3.3 ONCOCIN ScriptGen

The ScriptGen system [Shwe:89] is a fully-realised demonstration of the *parallel model* approach to generating knowledge-based system test cases. This approach is based on the idea that test case generation is a knowledge-based activity, and therefore an effective test case generator requires its own knowledge base [Mars:87]. This knowledge base cannot be the same as that used by the knowledge-based system, since then the system would only be capable of testing the cases that it "knows about"; errors and omissions in the system would not be revealed. For the same reason, the parallel model needs to be constructed using the knowledge of different experts from those used in building the knowledge-based system.

There are two types of parallel model, depending upon whether the model can create the correct (acceptable) outputs for generated test cases, or the input sets only. In the former case, the parallel model is similar to a deep model for the domain; in the latter case, the parallel model is more akin to a functional specification for the system. In neither case is it a replica of the system itself. Knowledge in the parallel model will probably be at a more abstract level than that in the system knowledge base, to keep the amount of work in constructing the parallel model to a reasonable level. Otherwise, the parallel knowledge base would be of comparable size and complexity to the original.

The chief disadvantage of this approach is the effort involved in building the parallel model. Furthermore, it will be necessary to validate the parallel model itself, for obvious reasons. This is another argument for building a parallel model that is far less complex than the original system. Moreover, if there are bugs in the parallel model, then these will manifest themselves when the system behaves "incorrectly", according to the erroneous expectations of the model (this assumes that bugs in the parallel model will be different to any in the original system – a reasonable assumption since different experts supplied the knowledge). The parallel model approach to validation will be of most value when the effort of building and validating the parallel model is small relative to the effort of the alternative approaches (having human experts painstakingly check the outputs of many blindly-generated random test cases, or having human experts manually design and check a set of test cases). The parallel model approach may be of greatest value when testing needs to be repeated several times with different test cases; the only additional effort each time will be that of checking the correctness of the system outputs, if the parallel model is not capable of doing this automatically.

The primary motivation for building ScriptGen to generate test cases for validating ONCOCIN lies in the nature of the task performed by this expert system. ONCOCIN provides recommendations to physicians concerning the administering of treatment *protocols* to patients receiving cancer therapy. These protocols specify standardised cycles of radiation and drug therapies to be administered to patients over time periods lasting several months. Therefore, each test case for ONCOCIN is a complex temporally-distributed sequence of varying parameters, following a specified protocol. These cases are called *scripts*, and a typical test script might contain values for 20 parameters over 10 visits: 200 values in total. Such scripts are hard to create manually (especially if bias is to be minimised and coverage is to be maximised), and even harder to check as to the required system recommendations. ScriptGen was developed to perform the following tasks:

- Assist experts in generating test scripts by allowing the user to select *goals* to test for a particular protocol, and then generating plausible sequences of visit and parameter data to simulate realistic protocols with ONCOCIN.

- Assist experts in checking the acceptability of ONCOCIN recommendations by providing script *annotations* that describe the purpose of tests at each stage in the simulated protocol.

Although randomness is employed in generating the data in the scripts, considerable protocol knowledge is used to constrain and guide the generation (knowledge of *how* testing is to proceed), and to provide the annotations (knowledge of *what* is being tested). The test goals are specified in ScriptGen as a *template*, such that any script generated for protocol *P* from template *T* is expected to produce the same ONCOCIN recommendations. That is, ScriptGen employs a domain-specific implementation of the equivalence partitioning method described in Section 3.1. Scripts within a partition can be generated at random or to test boundary values for the partition. If two scripts generated from the same partition result in different recommendations, then the method (random or boundary-value) used to generate them will suggest the likely source of error. Discordant output from boundary-value cases suggest small errors in the constants used in rule antecedent conditions, while those from randomly-generated cases suggest gross errors in system knowledge base, such as missing rules or incorrect rules.

ScriptGen was evaluated by an experiment in which ten errors were seeded into the ONCOCIN knowledge base [Shwe:89]. A domain expert suggested likely errors for this task. Test cases generated by ScriptGen successfully detected five of these errors; subsequent analysis of these results revealed the types of errors that ScriptGen is well-suited and ill-suited to detecting. Interestingly, one of these results indicates that ScriptGen should only be used after the knowledge base has been subjected to verification by anomaly detection, because ScriptGen fails to detect redundancy (in fact, it can fail to reveal errors in the presence of redundancy). In view of the results of this evaluation, ScriptGen can be considered an *error-based* testing method, since the equivalence partitioning method employed is directed towards revealing the presence of specified types of error in ONCOCIN. It does not guarantee the absence of unspecified errors.

Concluding Note As a final observation on the methods for generating test cases for validating knowledge-based systems, there seems to be a dichotomy in the available literature between domain-dependent systems which have been used in practice (for example, ScriptGen) and generic systems for which we could find no evidence of application as yet (GTM and EVA). Clearly, this work is still in its infancy, but it shows promise.

4 KBS Evaluation

The sub-field of KBS evaluation is neither as well-delineated nor as well-explored as the sub-field of formal verification. There are at least two reasons for this fact: the first is that, by definition, evaluation addresses less well-defined aspects of KBS, where there is little uniformity between systems in different application domains; the second reason is that most of the work in the validation domain has been carried out in university computer science departments, where formal methods are a more common research topic than empirical techniques.

One way in which we can assess the current state of KBS evaluation is to consider the main problems with which the sub-field has been concerned. Primarily, these problems are: the choice of standard against which the system will be evaluated, the choice of test-set to which the system will be subjected, and the choice of overall strategy in which evaluation will be conducted. We have already considered issues in choosing the test set, above, so this section focuses upon the other two questions.

4.1 Choice of Evaluation Standard

Ideally, we would always prefer to evaluate the performance of a KBS against an objective standard, often referred to as a "gold standard". For this to be possible for a given application, we would need to be able to obtain a set of *correct* (or, at least, *acceptable*) solutions to each problem that the system will be required to solve during evaluation. While this is possible in some relatively well-defined domains, it is not possible in many – if not most – domains in which KBS technology is applied. When no objective standard is available, two alternatives have been proposed [O'Keefe:87]:

- Define the opinion of a human expert (or that of the consensus between a group of human experts) to be the "gold standard". This may be appropriate if the system is designed to emulate the abilities of the expert(s) in question; otherwise, since all humans are fallible, this may be a poor choice of standard.
- Instead of treating the human expert(s) as an infallible "oracle", have a third-party group of experts compare – without knowing which is which – the solutions from humans and machine to the same set of problems; the machine can be considered to perform acceptably if there is no discernible difference in its performance.

When a gold standard is available, a number of standard statistical metrics have been adapted to measure the extent to which the results of the system agree with the standard results. We briefly consider four quantitative methods from the literature that are suitable for measuring agreement between a knowledge-based system and the comparison standard (usually human experts): the kappa statistic, the accuracy coefficient, the mean probability score, and the *paired-t* test.

4.1.1 Kappa Statistic

A widely-used method for evaluating the agreement between a knowledge-based system and human expert is the *kappa statistic* [Reggia:85] and also its weighted variant [Cohen:68]. In its simplest form:

$$k = \frac{(p_0 - p_c)}{(1 - p_c)}$$

where p_0 is the proportion of observed agreements between system and expert, and p_c is the proportion of agreements that could be attributed to chance. The result $k = 1$ indicates perfect agreement, while $k = 0$ is the level of agreement expected by chance alone. $k < 0$ would indicate a system which performs worse than chance.

4.1.2 Accuracy Coefficient

The kappa statistic does not address the issue of uncertainty in the knowledge-based system conclusions. The accuracy coefficient [Reggia:85] takes this into account:

$$Q = \frac{1}{n(1 - p_c)} \sum_{i=1}^{n} (p_i - p_c)$$

where n is the number of test cases, p_c is the level of conclusion certainty equivalent to pure chance, and p_i is the level of certainty assigned by the knowledge-based system to the correct solution for the i th test case (this is zero if the system failed to suggest that solution as a conclusion). Here, $Q = 1$ indicates perfect performance, $Q = 0$ indicates no skill (chance alone would account for this), and $Q < 0$ indicates a level of performance worse than chance.

4.1.3 Mean Probability Score

Another method used to assess system accuracy under uncertainty is the *mean probability score* (MPS) statistic [Levi:89]. For example, assume that the correct solution for the three test cases is x, and the system offers the following conclusions (the bracketed numbers are probabilities representing the level of certainty that the knowledge-based system has for each conclusion): x [0.3], y [0.6], x [0.9]. The MPS for the conclusions of the system for solution x would be:

$$\frac{((0.3-1)^2+(0.6-0)^2+(0.9-1)^2)}{3}=0.287$$

An MPS of 0 represents perfectly correct performance, 1 indicates perfectly incorrect performance, and $(c-1)^2$ indicates a level of performance due to chance alone, where the probability of a chance agreement is *c*).

4.1.4 Paired-t Test

The paired-t test [O'Keefe:87] can also be used to compare agreement between human experts and knowledge-based systems. For a set of test cases, we can compute the difference between human and system results as $D_i = X_i - Y_i$, where X_i are system results, and Y_i are human results (or gold standard results). Over *n* test cases, we have differences $D_i \cdots D_n$, for which the following confidence interval can be produced:

$$\bar{d} \pm \frac{t_{n-1,\alpha/2} S_d}{\sqrt{n}}$$

where $\bar{d}$ is the mean difference, S_d is the standard deviation, and $t_{n-1,\alpha/2}$ is the value from the *t* distribution with *n* degrees of freedom. We can accept the system as valid if zero lies in the confidence interval.

Additional statistical methods are considered in [Adelman:91, O'Keefe:87]. In general, if complex statistical procedures are used to validate knowledge-based systems, a statistician should participate in designing the validation experiment and interpreting the results.

4.2 Choice of Evaluation Strategy

Early evaluation efforts were planned either in an ad hoc manner as in XCON [Gaschnig:83], or used a variation on the blind-peer ("Turing test") approach as in MYCIN [Buchanan:84]. Neither is capable of delivering a reliable KBS, as experience shows [Gaschnig:83]: the initial evaluations of XCON and MYCIN failed to reveal flaws in these systems which would cause them to fail when fielded (XCON due to inadequate coverage of the validation tests; MYCIN due to a lack of field evaluation). The current viewpoint identifies two distinct types of validation activity, according to their placement within the knowledge-based systems development cycle [Preece:90]. *Laboratory validation* tests the system in an artificial setting, concentrating mainly upon the technical performance aspects of the system requirements. *Field validation* tests the system *in situ*, permitting the assessment of organisational and ergonomic requirements. The purpose here, as observed in [Rushby:88], is to systematically address each factor that could cause the users to reject the system – an example is that of the MYCIN system, which was evaluated only for its clinical utility after its diagnostic ability was validated as being expert-level [Buchanan:84]. The remainder of this section considers the design of laboratory and field validation procedures in more detail.

4.2.1 Laboratory Validation Procedures

Although some developmental validation tests will often be conducted informally, the laboratory setting provides the validators with the maximum control over experimental conditions. Therefore, at least some of the laboratory validation tests should be the most rigorous type of validation performed on a knowledge-based system. For this reason, a recent tutorial [Adelman:91] recommends the use of carefully designed experiments for laboratory validation. As defined by Adelman, the essential characteristics of an *experiment* are as follows:

- **Participants** If the knowledge-based system is to be tested with human users, then a test population of users needs to be recruited. For the results of the experiment to be considered reliable, the users should be selected at random from a population of intended users of the knowledge-based system. If the knowledge-based system is designed to be embedded in a larger software system, then it will not be necessary for human users to participate in the validation (although the inputs to the knowledge-based system from the external system will have to be simulated).
- **Independent variables** These are the experimental conditions. For example, if the performance of a group of humans using the system to solve test cases in the laboratory is to be compared with a control group who did not use the system, then one independent variable is whether a specific individual used or did not use the system.
- **Tasks** These are the test cases that the system will be used to solve. Criteria for choosing these are discussed at length in Section 3.
- **Dependent variables** These are the measurements that determine the acceptability of the results of the validation, as described in Section 4.1.
- **Control procedures** These are the procedures followed by the validators to control the experiment. The reliability (that is, the likelihood that the experiment can be repeated with the same results) and validity (acceptability of the results) of the experiment depend for the most part on these procedures. Factors that need to be considered to assure reliability include:
  - *Effect of competing hypotheses*: we must establish a causal link between the independent variables and the changes in dependent variables, and rule out any spurious factors.

- *Randomisation*: we do not want the results of the experiment to be sensitive to choice of participants or tasks, so these should be selected at random.
- *Statistical validity*: using methods such as increasing sample size and careful choice of test hypothesis, statistical errors need to be controlled for. A *Type I error* is committed when a valid system is rejected, and a *Type II error* is committed when an invalid system is accepted.
- *External validity*: consideration of the accuracy of experimental conditions with respect to users and tasks will increase the likelihood that the results of the laboratory experiment will generalise to the target setting.

Not all laboratory validation procedures need be conducted at the above level of rigour. In practice, one would choose a procedure commensurate with the objectives of validation. Alternative, less-rigorous procedures are considered for field validation below, but all of these are also applicable to laboratory validation.

4.2.2 Field Validation Procedures

The purpose of field validation is to demonstrate that the introduction of the knowledge-based system improves performance of the tasks required of the system. Sometimes this will be easy to demonstrate, such as when a knowledge-based system performs real time fault detection in isolated equipment (for example, a satellite). In other situations, such as the introduction of an knowledge-based system for diagnostic decision support in a hospital, it will be necessary to show that the users' diagnostic performance is improved by their use of the system.

Typically, in field situations, it will be harder to control the experimental conditions and thereby assure reliability and validity than in the laboratory. Therefore, if it is not possible to perform rigorous experiments as described above, the following are possible alternative methods, in decreasing order of reliability and validity [Adelman:91]:

- **Case study** Case studies are the next-best alternative to fully-controlled experiments in which lack of experimenter control is balanced by careful methodological procedures to minimise the loss of reliability and validity. For example, it may be impossible to reliably measure a dependent variable where experts in the domain disagree over the correct answers to test cases. In such cases, various *agreement* methods may be employed, or it may be possible to take several alternative measurements of the effect of the independent variables and see if they correlate well.
- **Time series design quasi-experiment** This design is useful where no control group is available. Instead, the single group is essentially used as its own control by taking a series of both pre-test and post-test

performance measurements over time. Thus, the effect of competing hypotheses which explain performance variations over time (such as seasonal variations or increasing user experience) can be controlled.
- **Non-equivalent control groups quasi-experiment** This design uses a control group and takes a single pair of pre-test/post-test observations per group. The weakness of the design lies in the fact that random sampling is not used to select the groups (hence, the word non-equivalent). Therefore, statistical tests applied to the observations from this study must control for possible selection differences in the groups.

There are many more possibilities for empirical validation procedures than the ones listed above, but these give a flavour for validators' options in procedure design. The most important point arising from this is that validators need to be aware of the reliability and validity implications of any empirical procedures that they might wish to use.

5 Conclusion

Despite the considerable amount of activity in the field, KBS validation is still an immature field. This assertion is based primarily on the following observations:
- While certain issues have been well-explored, other important issues have been neglected. This can be seen most clearly in the disparity between work in verification and evaluation, where both the theoretical and practical aspects of verification for rule-based systems are now well-understood, but almost nothing is known about what constitutes a thorough testing strategy for these systems.
- While many novel techniques and tools have been proposed in the last decade, there have been few attempts made to evaluate the relative effectiveness of these. No known experiment has been conducted to compare state-of-the-art verification tools with state-of-the-art evaluation techniques, in terms of their effectiveness in detecting different kinds of fault in KBS.
- The field suffers from the problem of trying to "hit a moving target": when work began, the majority of KBS under development were rule-based; now, object-based, model-based and hybrid systems are becoming common. Validation technology is still largely directed at the older, rule-based type of system.
- The technology has not yet become adopted by practitioners. Recent surveys show that ad hoc techniques dominate in industry, and developers cite validation problems as among their greatest difficulties [Hamilton:91, O'Leary:91].

To a certain extent, the trends mentioned in Section 2 seem to be emerging in response to the limitations in the present state-of-the-art in validation. The current NRC/EPRI-funded work at SAIC includes the following deliverables:

- A survey of the effectiveness of conventional software verification and evaluation techniques for validating KBS [Miller:93a]. The survey found that only the knowledge base component of a knowledge-based system does not seem to be covered adequately by conventional software validation techniques. The results of the survey are also potentially useful for validation hybrid systems including conventional and knowledge-based components.
- An experiment comparing a number of KBS verification techniques [Miller:93b]. This experiment is one of the first to provide concrete statistical measurements for the utility of knowledge-based verification techniques. One finding was that a group using an anomaly detection tool significantly out-performed its control group in identifying and locating faults in two nuclear industry KBS applications.
- A set of recommendations for the validation of object-oriented software. Preliminary findings suggest that a taxonomy of anomaly types can be identified for this class of software which is compatible with the taxonomy of anomalies for KBS software (this finding is supported by research into validation of frame-based KBS, which may be seen as a special case of object-based systems [Lee:93]). A convergence of these technologies may provide a means of addressing the validation problems of hybrid software systems in the future.

While the NRC/EPRI project is currently the largest validation initiative in North America, the evolving nature of the field is reflected in the work of the other active groups, including:

- Boeing's use of verification technology in the form of KBR3 [Dahl:93].
- Aerospace Corporation's efforts in validating hybrid software [Landauer:93].
- The joint NASA/IBM effort in training industry to use existing validation techniques [French:93].
- Recent interest in evaluating KBS testing strategies from a number of universities in the U.S. and Canada [Preece:94, Kirani:92, Zualkernan:93].

Taken as a whole, these current efforts are addressing the limitations of the field, making industry aware of the available technology, strengthening the areas in which present technology is weak, and adapting to the changing nature of KBS themselves.

Bibliography

[Adelman:91] Adelman, L. (1991). Experiments, quasi-experiments, and case studies: A review of empirical methods for evaluating decision support systems. *IEEE Transactions on Systems, Man and Cybernetics*, 21(2):293–301.

[Batarekh:91] Batarekh, A., Preece, A. D., Bennett, A., and Grogono, P. (1991). Specifying an expert system. *Expert Systems with Applications*, 2(4):285–303.

[Buchanan:87] Buchanan, B. G. (1987). Artificial intelligence as an experimental science. Technical Report KSL 87-03, Knowledge Systems Laboratory, Stanford University, Stanford, CA.

[Buchanan:84] Buchanan, B. G. and Shortliffe, E. H. (1984). The problem of evaluation. In Buchanan, B. G. and Shortliffe, E. H., editors, *Rule-Based Expert Systems: the MYCIN Experiments of the Stanford Heuristic Programming Project*, chapter 30, pages 571–588. Addison-Wesley, Reading MA.

[Chang:90a] Chang, C. L., Stachowitz, R. A., and Combs, J. B. (1990). Validation of nonmonotonic knowledge-based systems. In Dollas, A., Tsai, W. T., and Bourbakis, N. G., editors, *Proc. 2nd International Conference on Tools for Artificial Intelligence (TAI-90)*, pages 776–782. IEEE, IEEE.

[Chang:90b] Chang, C. L., Combs, J. B., and Stachowitz, R. A. (1990). A report on the Expert Systems Validation Associate (EVA). *Expert Systems with Applications*, 1(3):217–230.

[Chen:93] Chen, Z. and Suen, C. Y. (1993). Application of metric measures: from conventional software to expert systems. In Preece, A. D., editor, *Validation and Verification of Knowledge-Based Systems (AAAI-93 Workshop Notes)*, pages 44–51. AAAI. AAAI Press Technical Report.

[Childress:91] Childress, R. L. and Valtorta, M. (1991). EVA and the verification of expert systems written in OPS5. In O'Leary, D. E., editor, *AAAI-91 Workshop on Verification and Validation of Knowledge Based Systems*. AAAI.

[Dahl:93] Dahl, M. and Williamson, K. (1993). Experiences of using verification tools for maintenance of rule-based systems. In Preece, A. D., editor, *Validation and Verification of Knowledge-Based Systems (AAAI-93 Workshop Notes)*, pages 114–119. AAAI. AAAI Press Technical Report.

[deKleer:86] de Kleer, J. (1986). An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162.

[French:93] French, S. W., Culbert, C., and Hamilton, D. (1993). Experiences in improving the state of the practice in verification and validation of knowledge-based systems. In Preece, A. D., editor, *Validation and Verification of Knowledge-Based Systems (AAAI-93 Workshop Notes)*, pages 86–93. AAAI. AAAI Press Technical Report.

[Gamble:93] Gamble, R. F. (1993). A perspective on formal verification. In Preece, A. D., editor, *Validation and Verification of Knowledge-Based Systems (AAAI-93 Workshop Notes)*, pages 131–134. AAAI. AAAI Press Technical Report.

[Gamble:91] Gamble, R. F., Roman, G.-C., and Ball, W. E. (1991). Formal verification of pure production system programs. In *Proc. 9th National Conference on Artificial Intelligence (AAAI 91)*, pages 329–334. AAAI Press.

[Gaschnig:83] Gaschnig, J., Klahr, P., Pople, H., Shortliffe, E., and Terry, A. (1983). Evaluation of expert systems: Issues and case studies. In Hayes-Roth, F., Waterman, D. A., and Lenat, D. B., editors, *Building Expert Systems*, chapter 8, pages 241–280. Addison-Wesley, Reading MA.

[Giarratano:89] Giarratano, J. and Riley, G. (1989). *Expert Systems: Principles and Programming*. PWS-Kent, New York.

[Ginsberg:88] 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, pages 585–589.

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

[Ginsberg:93] Ginsberg, A. and Williamson, K. (1993). Inconsistency and redundancy checking for quasi-first-order-logic knowledge bases. *International Journal of Expert Systems: Research and Applications*, 6(2):321–340.

[Goodenough:75] Goodenough, J. B. and Gerhart, S. L. (1975). Toward a theory of data selection. *IEEE Transactions on Software Engineering*, SE–1(2):156–173.

[Grogono:91] Grogono, P., Preece, A., Shinghal, R., and Suen, C. (1991). Techniques for the evaluation of expert systems in telecommunications. In *Proceedings of 1991 Bell Canada Quality Engineering Workshop*. Bell Canada.

[Grossner:93] Grossner, C., Preece, A., Chander, P. G., Radhakrishnan, T., and Suen, C. Y. (1993). Exploring the structure of rule based systems. In *Proc. 11th National Conference on Artificial Intelligence (AAAI 93)*.

[Hamilton:91] Hamilton, D., Kelley, K., and Culbert, C. (1991). State-of-the-practice in knowledge-based system verification aand validation. *Expert Systems with Applications*, 3:403–410.

[Highland:92] Highland, F. and Kornman, B. (1992). A design language for knowledge-based application development. In Miller, L. A., editor, *AAAI-92 Workshop on Verification and Validation of Knowledge Based Systems*. AAAI.

[Huang:75] Huang, J. C. (1975). An approach to program testing. *ACM Computing Surveys*, 7:113–128.

[Jacob:90] Jacob, R. J. K. and Froscher, J. N. (1990). A software engineering methodology for rule-based systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):173–189.

[Kiper:89] Kiper, J. D. (1989). Structural testing of rule-based expert systems. In *IJCAI-89 Workshop on Verification, Validation and Testing of Knowledge-Based Systems*. IJCAI.

[Kiper:92] Kiper, J. D. (1992). Structural testing of rule-based expert systems. *ACM Transactions on Software Engineering and Methodology*, 1(2):168–187.

[Kirani:92] Kirani, S., Zualkernan, I. A., and Tsai, W.-T. (1992). Comparative evaluation of expert system testing methods. Technical Report TR 92-30, University of Minnesota, Department of Computer Science.

[Landauer:93] Landauer, C. and Bellman, K. (1993). Designing testable, heterogeneous software environments. In Preece, A. D., editor, *Validation and Verification of Knowledge-Based Systems (AAAI-93 Workshop Notes)*, pages 35–37. AAAI. AAAI Press Technical Report.

[Laurent:92] Laurent, J.-P. (1992). Proposals for a valid terminology in KBS validation. In Neumann, B., editor, *Proc. 10th European Conference on Artificial Intelligence (ECAI-92)*, pages 829–834. John Wiley.

[Lee:93] Lee, S. and O'Keefe, R. M. (1993). Subsumption anomalies in hybrid knowledge bases. *International Journal of Expert Systems: Research and Applications*, 6(2):299–320.

[Levi:89] Levi, K. (1989). Expert systems should be more accurate than human experts: Evaluation procedures from human judgment and decisionmaking. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-19(3):647–657.

[Mars:87] Mars, N. J. I. and Miller, P. L. (1987). Knowledge acquisition and verification tools for medical expert systems. *Medical Decision Making*, 7:6–11.

[McGuire:90] McGuire, J. G. (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 COMPASS-90).*

[Mehrotra:93] Mehrotra, M. and Wild, C. (1993). Multi-viewpoint clustering analysis. In Preece, A. D., editor, *Validation and Verification of Knowledge-Based Systems (AAAI-93 Workshop Notes)*, pages 52–63. AAAI. AAAI Press Technical Report.

[Miller:90] Miller, L. A. (1990). Dynamic testing of knowledge bases using the heuristic testing approach. *Expert Systems with Applications*, 1(3):249–269.

[Miller:93a] Miller, L. A., Groundwater, E., and Mirsky, S. M. (1993). Survey and assessment of conventional software verification and validation methods. Technical Report NUREG/CR-6018; EPRI TR-102106; SAIC-91/6660, Science Applications International Corporation; U.S. Nuclear Regulatory Commission; Electric Power Research Institute.

[Miller:93b] Miller, L. A., Hayes, J. E., and Mirsky, S. M. (1993). Knowledge base certification. Technical report, Science Applications International Corporation; U.S. Nuclear Regulatory Commission; Electric Power Research Institute. Prepublication version.

[Nguyen:87] Nguyen, T. A. (1987). Verifying consistency of production systems. In *Proc. 3rd Conference on Artificial Intelligence Applications*, pages 4–8. IEEE Computer Society.

[Nguyen:85] Nguyen, T. A., Perkins, W. A., Laffey, T. J., and Pecora, D. (1985). Checking an expert systems knowledge base for consistency and completeness. In *Proc. 9th International Joint Conference on Artificial Intelligence (IJCAI 85)*, volume 1, pages 375–378. AAAI.

[O'Keefe:87] O'Keefe, R. M., Balci, O., and Smith, E. P. (1987). Validating expert system performance. *IEEE Expert*, 2(4):81–90.

[O'Keefe:93] O'Keefe, R. M. and O'Leary, D. E. (1993). Expert system verification and validation: a survey and tutorial. *Artificial Intelligence Review*, 7(1):3–42.

[O'Leary:91] O'Leary, D. E. (1991). Design, development and validation of expert systems: A survey of developers. In Ayel, M. and Laurent, J.-P., editors, *Validation, Verification and Test of Knowledge-Based Systems*, pages 3–20. John Wiley.

[O'Leary:93] O'Leary, D. E. (1993). Verification and validation of multiple agent systems: combining agent probabilistic judgments. In Preece, A. D., editor, *Validation and Verification of Knowledge-Based Systems (AAAI-93 Workshop Notes)*, pages 27–34. AAAI. AAAI Press Technical Report.

[Ostrand:88] Ostrand, T. J. and Balcer, M. J. (1988). The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686.

[Preece:92a] Preece, A. and Shinghal, R. (1992). Verifying knowledge bases by anomaly detection: An experience report. In Neumann, B., editor, *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 92)*, New York. European Coordinating Committee for Artificial Intelligence (ECCAI), John Wiley & Sons.

[Preece:92b] Preece, A. D., Shinghal, R., and Batarekh, A. (1992). Principles and practice in verifying rule-based systems. *Knowledge Engineering Review*, 7(2):115–141.

[Preece:92c] Preece, A. D., Shinghal, R., and Batarekh, A. (1992). Verifying expert systems: a logical framework and a practical tool. *Expert Systems with Applications*, 5:421–436. Invited paper.

[Preece:90] Preece, A. D. (1990). Towards a methodology for evaluating expert systems. *Expert Systems*, 7(4):215–223.

[Preece:93] Preece, A. D. (1993). A new approach to detecting missing knowledge in expert system rule bases. *International Journal of Man-Machine Studies*, pages 161–181.

[Preece:94] Preece, A. D., Grossner, C., Chander, P. G., and Radhakrishnan, T. (1994). Structural validation of expert systems: Experience using a formal model. In Liebowitz, J., editor, *Expert Systems Second World Congress Proceedings*, Oxford. Pergamon Press.

[Reggia:85] Reggia, J. A. (1985). Evaluation of medical expert systems: A case study in performance assessment. In *Proc. 9th Annual Symposium on Computer Applications in Medical Care (SCAMC 85)*, pages 287–291. Also in Miller, Perry L., ed., *Selected Topics in Medical AI*, New York, Springer, 1988, pp. 222–230.

[Rushby:88] Rushby, J. (1988). Quality measures and assurance for AI software. NASA Contractor Report CR-4187, SRI International, Menlo Park CA. 137 pages.

[Rushby:90] Rushby, J. and Crow, J. (1990). Evaluation of an expert system for fault detection, isolation, and recovery in the manned maneuvering unit. NASA Contractor Report CR-187466, SRI International, Menlo Park CA. 93 pages.

[Rushby:89] Rushby, J. and Whirehurst, A. (1989). Formal verification of AI software. NASA Contractor Report CR-181827, SRI International, Menlo Park CA.

[Shwe:89] Shwe, M. A., Tu, S. W., and Fagan, L. M. (1989). Validating the knowledge base of a therapy planning system. *Methods of Information in Medicine*, 28(1):36–50.

[Suwa:82] Suwa, M., Scott, A. C., and Shortliffe, E. H. (1982). An approach to verifying completeness and consistency in a rule-based expert system. *AI Magazine*, 3(4):16–21.

[Waldinger:91] Waldinger, R. J. and Stickel, M. E. (1991). Proving properties of rule-based systems. In *Proc. IEEE Conference on Artificial Intelligence Applications 1991*, pages 81–88. IEEE Press.

[Weyuker:80] Weyuker, E. J. and Ostrand, T. J. (1980). Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, SE–6(3):236–246.

[Wood:90] Wood, W. T. and Frankowski, E. N. (1990). Verification of rule-based expert systems. *Expert Systems with Applications*, 1(3):317–322.

[Zualkernan:93] Zualkernan, I. A. and Lin, Y.-Y. (1993). An analysis of output-based partition testing for heuristic classification expert systems. In Preece, A. D., editor, *Validation and Verification of Knowledge-Based Systems (AAAI-93 Workshop Notes)*, pages 8–15. AAAI. AAAI Press Technical Report.