# Towards a Quality Assessment Framework for Knowledge-Based Systems*

Alun D. Preece

**Abstract**

Despite the success of knowledge-based systems in many domains, techniques for assuring their quality and reliability are still immature. A well-engineered software product is built using a process in which quality issues are addressed throughout. Key processes affecting product quality are specification, verification, and validation. The processes are characterized by the production of certain artifacts at certain times. Therefore, by looking at the artifacts produced by a specific process, we can assess the quality of the resulting product, and also we can assess the capability of the developers to produce high-quality software. This article seeks to answer precisely this question: how can the level of capability of a supplier of knowledge-based systems be evaluated? This question is of concern to all customers of knowledge-based systems and, although standards for supplier-capability of conventional software have been developed, none address the unique characteristics of knowledge-based software. To answer the question, this article describes a set of state-of-the-art techniques for specifying, verifying, and validating knowledge-based systems, highlighting the artifacts associated with each. We also assess the state-of-the-practice, indicating the limited extent to which the techniques are in use at the present time. In addition to providing answers to the motivating question, the main objective of the article is to recommend ways in which the current state-of-the-practice can be improved.

*The work described in this paper was funded by Bell Canada. All opinions expressed, however, are those of the author.

# 1 Motivation and Overview

In recent years, knowledge-based software technology has proven itself to be a valuable tool for solving hitherto intractable problems. Knowledge-based software systems are characterized by having an explicit representation of a substantial body of application domain knowledge, embodied in a knowledge base which is kept separate from the inference mechanisms that use the domain knowledge to solve problems. Most often, knowledge-based systems have been applied to application problems involving a need for flexible, high-performance, problem-solving ability in a narrow area of 'expertise'—such systems are commonly referred to as *expert systems*. Expert systems have achieved great success in domains such as telecommunications, aerospace, medicine, and the computer industry itself. Arguably, the most common tasks performed by expert systems are those involving diagnosis and planning activities. The criteria by which the success of an expert system is judged vary from application to application—some are deemed successful if they make or save large sums of money, while others succeed because they help their users to understand better their own knowledge, as applied to the tasks they perform.

As a result of this success, an increasing number of organizations are becoming active in using and producing knowledge-based systems. This has lead to great benefits for the organizations concerned, but it has also lead to serious problems. The transition of knowledge-based technology from research laboratories to software development centers highlighted the fact that, while great advances had been made in certain areas, notably knowledge representation and automated reasoning, other important issues had been neglected. One of the most significant of these issues was that of assessing the *quality* of knowledge-based software systems. The lack of work in this area became even more apparent when knowledge-based systems were compared to more conventional (that is, more well-established) software technology, where much effort had gone into developing powerful and practical methodologies for assessing and assuring the quality of software systems. As, increasingly, knowledge-based systems are integrated with conventional software to create hybrid systems, the disparity in quality assurance techniques is even more problematical.

The demand for quality assurance methods for knowledge-based systems provided the impetus for several well-funded research efforts (in addition to many more efforts of a more modest nature) to tackle aspects of this problem. Areas investigated included life-cycle models for knowledge-based systems, specification techniques, verification and validation methods and tools, and the applicability of conventional software quality assurance techniques to knowledge-based systems. Access to much of this work is provided by a number of recent publications, including [2, 14, 21, 23, 54]. One conclusion upon which most of these researchers reached agreement was that knowledge-based systems, because of their distinct characteristics, require quality assurance techniques different from those of conventional software. Interestingly, certain aspects of software quality seem to be easier to assure, at least in principle, for knowledge-based systems; other aspects, however, seem to be much harder for knowledge-based systems. While there are still many difficult issues to resolve, it is clear that we can now start offering some specific conclusions and recommendations to users and would-be users of this new technology, regarding the quality assurance procedures that they should follow.

This article arose in response to a specific question posed by one corporate user of knowledge-based systems technology. Bell Canada has recently expended considerable effort in creating a standard model for assessing the *capability* of their prospective software suppliers. This model, called *TRILLIUM* [5], draws upon and extends a number of pre-existing software quality standards,

| *Level* | *Capability* |
|---|---|
| Level 1 | Unstructured development: no use of standards, process models, or methods for process improvement between projects. |
| Level 2 | Repeatable and project-oriented development: some use of standards—for example [62], unstructured methods for continuous process improvement, broadly equivalent to mid-1980's state-of-the-art. |
| Level 3 | Well-defined process-oriented development: use of rigorous standards, organization-wide development processes, systematic process improvement, equivalent to current state-of-the-art. |
| Levels 4 and 5 | Aimed towards total process management and integration, anticipated to be mid-to-late 1990's state-of-the-art. |

Table 1: The five levels of the *TRILLIUM* supplier capability model.

notably the Carnegie Mellon University Software Engineering Institute's *Capability Maturity Model* (CMM) [43]. In the *TRILLIUM* model, capability is defined as the ability of a developer to deliver a software product is such a way as to maximize correctness and reliability, while minimizing development costs.[1] As with the CMM, the *TRILLIUM* model establishes a five-level scale against which to measure supplier capability, as shown in Table 1. It is Bell Canada's intention that all their software suppliers should be evaluated to conform to at least Level 3 in this model; the body of the *TRILLIUM* model serves to define a set of evaluation criteria against which suppliers can be measured.

The *TRILLIUM* model is directed towards conventional software: each criterion in it relates to quality assurance practices defined in the conventional software engineering literature, most of which are widely understood. The question posed by Bell Canada was of how to adapt the details of this model so that it can be used to evaluate the capability of would-be suppliers of expert systems. Such suppliers could be in-house development teams, or external contractors. This article provides a partial answer to this question, by examining a variety of integrated techniques aimed at assuring the quality of a knowledge-based system. We focus upon techniques for specification, verification, and validation, since we assert that these are the strongest indicators of the quality of the resulting software product. We also focus upon technical issues[2] since the human and managerial factors described in the other sections of the model are equally applicable to knowledge-based system development.

Recent surveys indicate that, at least in the areas of specification, verification, and validation, most current developers of knowledge-based systems would be ranked at Level 1 or Level 2 in the *TRILLIUM* model [24, 39]. Ad hoc approaches dominate, and there is little use of tools or structured methods. This is in spite of the fact that, in the last ten years, many tools and structured

[1] Here, *correctness* has the conventional meaning of compliance to specified requirements, *reliability* is synonymous with 'dependability' (that is, the software should contain minimal defects), and *minimum costs* refers to both the lowest life-cycle cost and also the shortest development time.

[2] We refer specifically to Section 6 in the *TRILLIUM* model, concerning the technical processes of software development.

methods have been developed for KBS specification, verification, and validation. Therefore, the main goal of this article is to bridge this gap, by providing a survey of state-of-the-art techniques for KBS specification, verification, and validation, and prescribing a set of techniques corresponding to Level 3 capability in the *TRILLIUM* model.

The article can be used in three ways:

- To provide pointers to the state-of-the-art techniques available for KBS specification, verification, and validation.
- To provide a set of recommendations to improve the state-of-the-practice in KBS specification, verification, and validation.
- To provide a means of assessing the level of capability of a KBS supplier, in terms of their use of rigorous specification, verification, and validation techniques during development.

The presentation of specification, verification, and validation techniques focuses upon the *artifacts* which should be produced during the course of developing a high-quality knowledge-based system. Ultimately, it is the presence or absence of such artifacts in the development process of a prospective supplier that determines the capability of that supplier.

The article is organized as follows: we will begin by examining the distinct characteristics of knowledge-based systems, since these are the cause of our difficulties in applying quality assurance techniques to these systems. In Section 3, we discuss the relationships between the software development process and the artifacts generated by it, and we present a collection of interrelated artifacts, the production of which during development of a knowledge-based system is desirable for high system quality. Section 4 proposes potential uses of the artifacts in extending the *TRILLIUM* five-level capability model to knowledge-based systems.

# 2 Why Knowledge-Based Systems are Different

Knowledge-based systems are a distinct kind of software, characterized by specific functional and structural characteristics. Viewed as 'black boxes,' typical knowledge-based systems possess the following functional characteristics:

- They solve ill-structured problems, having a great many 'special cases,' each of which can be dealt with only by the provision of specific knowledge.
- When presented with input describing situations for which they were not specifically programmed, they are capable of providing acceptable responses.
- They can handle imprecise or incomplete input, and produce output with appropriate degrees of certainty.
- They are able to support their output by producing explanations or justifications of their reasoning.

Viewed as a 'glass boxes,' knowledge-based systems have a characteristic structure, as follows:

- Knowledge of their application domains is stated explicitly in a *knowledge base*. Such knowledge may be expressed declaratively, which is most appropriate for static domain facts and relations, or it may be expressed procedurally, which is appropriate for knowledge of tasks and actions.

- The knowledge in the knowledge base is selected and applied to solve problems by a separate *inference engine*, which is a conventional program implementing a limited number of (usually fairly simple) reasoning and communication mechanisms.

The behavior of the knowledge-based system results from the interaction of inference engine and knowledge base. This behavior is usually not obvious from simple inspection of the two components, especially when the system is complex.

The characteristics of knowledge-based systems have an impact upon all aspects of their development. The majority of the development effort for these systems occurs in the phase of *knowledge acquisition*, the purpose of which is to elicit and specify the domain and task knowledge required to produce the desired problem-solving behavior. Traditionally, little attention has been paid to the identification of functional and non-functional requirements for the system in advance of knowledge acquisition, chiefly because this activity appears to be performed entirely informally in practice (if it is performed explicitly at all). Following knowledge acquisition, the task of implementing the operational knowledge base and inference engine requires considerably less effort.

Development of a knowledge-based system typically proceeds iteratively, using an evolutionary prototyping strategy. That is, a subset of the desired functionality is realized in an initial operational prototype system, which is then exercised in order to obtain feedback to drive subsequent expansion and revision of the system. This process ultimately leads to either the realization of an acceptable final system, or to the scrapping of the project in cases where the problem turns out to be intractable. It is clear that techniques for verification and validation of the knowledge-based system must be used throughout system development. Verification techniques must be used to establish, as thoroughly as possible, the internal correctness of the evolving system at each stage in development; validation techniques are used to establish if and when the system functions acceptably according to the requirements of its would-be users.

The main difference between the development cycle for knowledge-based software and conventional software lies in the distance between specification and implementation: the specification of the knowledge base is typically much closer to the final implemented form (often, they are one and the same) than the specification of a conventional system is to its final code. Also, knowledge-based system development typically de-emphasizes the distinction between system specification and design—both activities have traditionally been merged in knowledge acquisition, and it is only recently that the issue has received attention in the knowledge-based systems literature [66, 68].

The differences that set knowledge-based systems apart from other software have several important implications for quality assurance. Most of these relate to the technical aspects of the development process, although there is a resultant impact upon the human and managerial aspects also. From the technical point-of-view, the use of software engineering processes such as modularization [42], object-oriented design [6], and formal specification [59], has proven to lead to products with desirable quality-related properties such as maintainability, reusability, verifiability and, hence, correctness and reliability. In building knowledge-based systems, the traditional blurring of specification, design and implementation activities for the knowledge base prevents effective

use of software engineering practices; consequently, it is much harder to assure the quality of these systems.

The structural and functional characteristics of knowledge-based software also raise issues relating to quality evaluation. Their unique architecture requires development of different techniques for structural verification and validation [22, 28]. Their functional complexity prevents the effective use of conventional functional verification and validation; for example, it is hard to apply the equivalence-partitioning method for software testing [41] because there will be a great many such partitions for a non-trivial knowledge-based system [55]. We return to these issues in Section 3.

In conclusion, then, if we are to be able to assure the quality of knowledge-based systems (and we must, if they are to become a reliable software technology) we must address these problems, and investigate how existing software engineering techniques can be adapted to knowledge-based systems, and also develop new techniques where the existing ones are inapplicable. This is the main objective of the remainder of this article.

# 3 Artifacts for Quality

Much recent work in improving software engineering practice focuses upon qualities of the software development process [18]. Many process qualities relate directly to qualities of the resulting software products; for example, product reliability is improved when the process ensures that a test plan is generated independently of design and coding.

While a development process is dynamic, we can study it statically by examining the sequence of artifacts generated by it. Typical software artifacts include specification and design document components (for example, a stated functional requirement or the interface specification for a particular system module), source code components, and the paraphernalia associated with verification and validation (for example, test cases, test results, and review reports). In practice, any and all of these will usually be kept on electronic media, under the control of a configuration management system during development.

Consequently, if a software process is to result in high-quality products, we would expect that process to produce certain artifacts at certain times. From the point of view of evaluating the capability of a software developer, the only way to study their prior use of development processes is to examine the residual artifacts. In the remainder of this section, we identify and describe a set of desirable artifacts which we believe should be created during the development of a knowledge-based system, if that system is to possess an adequately high degree of quality. The major kinds of artifact we present are as follows:

- Problem specification documents
- Conceptual model documents
- Design model documents
- Implemented system source code
- Documented results of verification analyses
- Documented results of validation analyses

This framework of artifacts was inspired by [40].

## 3.1 Problem Specification

A problem specification should be the first artifact (or collection of artifacts) generated by a sound knowledge-based system development process. Broadly speaking, the problem specification states what the system must do and how well it must do it (functional requirements), and also what additional constraints the system must conform to (non-functional requirements). Essentially then, the problem specification for a knowledge-based system is similar to the requirements specification for a conventional software system [62], but there are important differences. The main difference stems from the fact that the greatest effort in building a knowledge-based system is in the knowledge acquisition task. It is rarely possible to specify precisely what the system must do in different situations until a great deal of knowledge has been acquired—this occurs only relatively late in development. However, the customer and supplier need to agree on the main objectives and requirements of the system much earlier: the customer needs to state its needs, and the supplier needs a set of development goals. Therefore, the chief requirement for the contents of the problem specification for a knowledge-based system is that it should satisfy the role of a *contract* between supplier and customer that will stabilize early in development [3].

Due to the immature nature of the field, there has been little examination of this kind of specification for knowledge-based systems until very recently [63]. We will examine the main artifacts included in one set of guidelines for knowledge-based system problem specifications here—a more detailed presentation of this proposal appears in [3].

**Problem description** The problem specification must contain an explicit description of the problem that the knowledge-based system is to solve. At least a preliminary statement of this must be made at the outset of development. While this seems obvious, most of the early knowledge-based systems were experimental and had only a vague notion of their goals; no explicit description of the problem was made. The problem description needs to include definitions of important concepts and terminology in the application domain, as well as a description of the tasks that the knowledge-based system is to perform. The latter should be expressed in terms of how the tasks are solved without the system (presumably by humans).

Following this description of the human system, the objectives for the knowledge-based system need to be specified. Part of this will comprise the *cooperation model* identified in the KADS knowledge-acquisition methodology [68], which indicates how tasks in the problem-solving process will be divided between the system and its users. Each task must then be specified in terms of:

- the input it requires, and the form and source of that input;
- the output it produces, and the form and destination of that output.

As a consequence of the nature of knowledge-based systems, there will rarely be a simple functional relation between the input and output of any of the tasks.[3]

**Performance constraints** As we have observed, due to the complexity of knowledge-based systems applications, it is practically impossible to specify precisely the functional relations between input and output for each system task until a great deal of domain knowledge has been acquired.

[3]If this were not the case, then the application would be too trivial to warrant a knowledge-based system solution.

However, the role of problem specification-as-contract demands that target performance requirements be stated for the system relatively early in development, even if the functions by which the system will achieve the required performance cannot be.[4] The difficulty here is that it will be hard to state realistic requirements with an appropriate degree of rigor. The approach we take to solving this problem aims at maximizing system reliability, as follows.

In accordance with the approach taken in [54], we identify two aspects of system performance:

*Minimum performance constraints* These will often be concerned with system *safety*: they state the things that that system must always do. They may be functional or non-functional in nature. An example of the former would be, "the system must never prescribe a drug overdose," and an example of the latter would be, "the system must always respond within five seconds." Where possible, we will wish to state these constraints formally, so that they can be proven mathematically. If this cannot be done, then rigorous tests must be stated in the system *test plan* for these constraints. We return to these points later.

*Desired performance constraints* The nature of most knowledge-based system applications is such that they are allowed to perform at a level which is less than perfect. For example, a particular diagnosis system may be deemed acceptable by its customer if it performs as required on 90% of some selected suite of test cases. Often, the setting of such a performance threshold level acknowledges the fact that humans perform less than perfectly on the application problems. In such cases, the desired level of performance may be set as equal to or higher than the human level. Of course, it is necessary to define the methods by which the compliance of the system with its desired performance requirements will be established. Tests must be agreed upon between supplier and client, and recorded in the test plan for the system, for use in later validation testing.

One difficulty in setting such a threshold level is that it may not be realistic. For example, after considerable development effort, it may turn out that the system fails to meet the level by a certain margin. In such a case, it may be that the customer is still willing to accept the system, at the lower performance level. Therefore, the main difference between the two kinds of performance constraint is that the desired performance may be renegotiated later in development, whereas the minimum performance will not be.

**Developing the specification** In summary, the artifacts comprising the problem specification include the following: *problem description document, statement of minimum and desired performance constraints*, and *test plan*. These are the most important components of the problem specification—others are described in [3]. Although we do not dwell upon life-cycle issues here, it should be noted that the problem specification is expected to evolve during the early iterations of a cyclic development strategy, rather than being completely fixed at the beginning. The main reason for this is that detailed knowledge acquisition and system design will usually lead to revision of an initial problem specification, to make it more correct or complete. However, as we have already discussed, we expect that the problem specification document will stabilize relatively early in development, compared to the other descriptions of the system, described in the following sections.

[4] We use the term *performance* in a general sense to mean 'the ability to fulfill a requirement' rather than in the specific sense of equating to 'efficiency'.

## 3.2 Conceptual Model

The notion of a *conceptual model* for a knowledge-based system is partly a realization of Newell's proposal for the 'knowledge level' [36], and partly inspired by domain modelling techniques from artificial intelligence [58] and software engineering [15]. The main purpose of the conceptual model is to provide an epistemological analysis of the domain knowledge prior to its implementation in an operational knowledge-based system [3]. This provides for separation of concerns in the development process: the conceptual model allows the developers to acquire, understand, and structure the knowledge without worrying about how it will be represented and used for efficient problem-solving. An enormous benefit of this approach is that the artifacts comprising the conceptual model are retained as independent documentation for system maintenance.[5] Arguably the most extensively developed techniques for conceptual modelling of knowledge-based systems are the *generic task* approach [9], the KADS methodology [68], and the work of Clancey [11, 12]. We will explore the contents of a conceptual model in terms of the artifacts which comprise it; in doing so, we will draw upon the aforementioned, and other, approaches.

A conceptual model usually contains at least two distinct types of knowledge: *domain knowledge* and *task knowledge*. The domain knowledge describes static entities and relations in the application domain. The task knowledge describes the methods by which problems are solved using the domain knowledge.[6] If there are more types of knowledge, then these will most likely arise from distinguishing between different kinds of task knowledge. A conceptual model can often be viewed as a hierarchy of levels, with one level for each distinct type of knowledge, where the higher levels organize the lower levels. For example, the KADS approach uses four distinct levels of knowledge [68]:

- the *domain knowledge*, as described above;
- *primitive inference task knowledge*, which defines the basic 'building blocks' for reasoning with the domain knowledge;
- *task knowledge*, describing how the primitive inference steps are combined into larger problem-solving mechanisms;
- *strategic knowledge*, describing the high-level methods by which tasks are selected for a problem-at-hand.

One issue upon which there is little agreement in the literature concerns the manner in which a conceptual model should be represented, and particularly the question of how formal such a model should be. Some conceptual models are entirely informal, lacking both precise syntax and semantics. The knowledge is typically recorded in the form of a 'paper model,' written in natural language. Most of the approaches described in the literature are *semiformal*, having a definite syntax but ill-defined semantics. The KADS methodology employs different languages at each of the four levels; for example, a descriptive entity-relationship modelling language is used for the domain level, and a procedural pseudocode-style language is used at the task level [68].

[5] While the system implementation will certainly be dependent on the conceptual model, the converse should not be true.

[6] A note on terminology: the task knowledge may also be 'domain' knowledge, in the sense that it may describe methods followed by humans in the domain; we do not call it such because it is often the case that different or entirely new methods will be used for the knowledge-based system. We reserve the term 'domain knowledge' for the static knowledge employed to solve problems.

More recently, the KADS-II project has been developing a representation language for KADS conceptual models which is entirely formal. This language, called $(ML)^2$, utilizes different logic formalisms to represent the four levels [65]. The use of logic for expressing formal conceptual models is also advocated in [47]. The use of a formal conceptual modelling language is still somewhat controversial. The disadvantages are that the effort involved in building the model is greatly increased, and some of the flexibility of using a non-formal language will be lost. In some cases, it may be desirable to formalize only part of the conceptual model. The substantial advantages of using a formal language are that the conceptual model will be free from ambiguity, and that it can benefit from the use of formal verification techniques (see Section 3.5).

If the conceptual model is represented formally, then it may be possible to execute it directly. Then, the design process may be largely circumvented, and the conceptual model may become the implemented system. This approach is followed by many proponents of logic-based knowledge representation [26]. The main problem with this approach is that conceptual-level descriptions of tasks usually are not efficient enough for practical operation. As an example, consider a diagnosis task which is specified as a set-covering operation (that is, to find the smallest set of disorders which completely explains a set of observed symptoms); while this task is trivial to define formally, a naive implementation of the set-covering operation is only tractable for simple problems, due to its exponential complexity [52]. The only alternative in this case would be to develop a more efficient implementation of this conceptual task, which is one role of the design model, described below.

In summary, the conceptual model consists of a set of artifacts which describe the different knowledge components of the knowledge-based system, including domain and task knowledge. Note that there is no notion of an *inference engine* at the conceptual level, because the model is not executable. (In the case where a formal conceptual model is to become the executable system, it will be necessary to design and implement a mechanism which embodies the inference rules of the formalism; for example, if first-order logic is used, then the inference engine may simply be an implementation of the resolution rule, as in Prolog [60]).

Finally, it is worth observing that the conceptual model of a knowledge-based system has no direct counterpart in conventional software engineering, because it is concerned with solving epistemological problems that arise due to the sheer complexity of the knowledge needed for a knowledge-based system application. Where knowledge modelling issues arise in conventional system development, they are usually resolved either at the requirements specification or the design stages.

## 3.3 Design Model

While the need for conceptual models in the development process for knowledge-based systems is generally agreed upon, the need for a design model is far more controversial. The strongest argument that we can offer for it is that, between the conceptual level and the implemented system, many concrete issues must be resolved. These issues typically include the specification and design of the inference engine, the design of procedures to implement certain tasks of the knowledge-based system, and the design of efficient representation structures for the domain knowledge. Difficult design choices will often be involved in resolving these issues, and sound software engineering practice dictates that these choices be recorded in the form of a collection of artifacts comprising the *design model*.

Few detailed examinations of knowledge-based system design models appear in the literature; exceptions include [3, 68, 66]. Below, we present each of the main components of a knowledge-based

system design model, drawing upon the published work and our own experience.

**Inference engine specification** As we have already said, the inference engine of a knowledge-based system is a conventional software program which implements a number of mechanisms by which the knowledge in the knowledge base is applied to solve problems. Typically, mechanisms are also provided to handle communication between the knowledge-based system and the external environment (although the decision about when to invoke these mechanisms may be taken either by knowledge, or by the inference engine itself, possibly as a default action when no further inferences can be derived from the knowledge base).

As a conventional program, the inference engine must be designed, at least at a high level of abstraction. Specifically, the design model must specify the properties that the inference engine needs to possess. At this time, the developers may decide to utilize an existing inference engine—from a commercial expert system shell, for example [34]. If so, the specified properties will be used to verify that the chosen inference engine meets the requirements (see Section 3.5). If not, the design process must proceed to detailed design of the inference engine, which will then be implemented conventionally (and verified). Formal specification languages developed for conventional software may be employed for these purposes, although this seems to have been done only rarely up till now—for an interesting pilot study, see [30].

**Knowledge base design** The design model makes concrete many issues left specified only vaguely at the conceptual level. Foremost among these are how the various knowledge-level components will be realized in the final implementation. For example, in the DESIRE design/specification language [66], interacting reasoning tasks are represented as modules. Internally, each module is composed of a collection of declarative domain knowledge constructs (such as logical implications) or, where appropriate, conventional procedural constructs. These reasoning modules are specified to interact in a rich variety of ways; for example, the output data of one module may become control information for another, or one module may issue a request for information to a second module, in which that request becomes a goal to solve.

As discussed in the previous section, not all the tasks specified at the conceptual level will be trivial to implement; we gave as an example a task that would be intractable if implemented naively. In such cases, the task must be refined into an efficient algorithm, and the design of this algorithm would be stated in the design model.

The formalization of the conceptual knowledge into a concrete design raises the question of what needs to be done if the conceptual model is itself fully formal. There are different views on this issue; one is that the conceptual model should be only semiformal, and formalization should be deferred until the design model [68]. However, there need not be any conflict here, because the concerns of the two models are very different: the conceptual model aims at describing the actual domain knowledge; the design model aims at implementing the knowledge efficiently. The needs of one are quite different from those of the other. For example, consider an expert system application, the knowledge of which can be expressed as a large number of if-then rules. At the conceptual level, we may wish to represent these rules in a purely declarative manner, using logical implications. Then, we may reap the benefits of automatic verification of the conceptual knowledge, as described in Section 3.5. However, this representation may not be efficient enough, if there are many rules and the system will need to do a great deal of searching to find applicable ones during problem-solving.

Therefore, we may choose to implement this system as a forward-chaining production rule system in the manner of the OPS5 or CLIPS systems [19]. Since such a system is highly procedural in nature, we will need to re-specify our purely-declarative rules at the design level. Both representations are equally formal, at least in principle[7], but quite different concerns motivated their selection: clarity of the knowledge at the conceptual level, and efficiency at the design level.

**Designing hybrid systems** A beneficial consequence of having an explicit design model which is similar to the design for a conventional software system is that it is easier to integrate conventional and knowledge-based components into a *hybrid system*. Such systems are becoming increasingly common, since many applications are neither 'purely conventional' nor 'purely knowledge-based.'

In summary, the design model for a knowledge-based system specifies those artifacts which document the rationale for implementing the system, based upon the conceptual model. These artifacts are crucial for the future understanding and maintenance of the system.

## 3.4 Implementation

The system implementation is the most obvious collection of artifacts generated by the development process. In general, it will consist of one or more knowledge base modules, and one or more inference engine components. If the decision was taken at the design stage to use a tool produced by some third-party (for example, an off-the-shelf expert system shell), then this will comprise part of the implementation also. In any case, the knowledge base will have been coded in whatever representation language was selected during system design. In some cases, it may be possible to generate the implementation wholly or partially using some mechanical transformation procedure [47]. This is very desirable if the transformation procedure can be guaranteed to preserve the correctness of the knowledge from the higher levels.

The quality of the implementation can be measured in several ways. Verification that the implementation is correct with respect to the design is covered in the next section; validation and testing according to the performance constraints is examined in Section 3.6. A third method by which the implementation quality can be evaluated independently is to use software *quality metrics*. However, while many metrics have been proposed and investigated for conventional software [13], little work has been done in this area for knowledge-based systems [27, 54, 4]. Metrics that have been suggested for evaluating the complexity of knowledge bases include quantifying the number of rules and potential inference chains [7, 22, 56], and determining the clustering of rules to provide a measure of the modularity of the knowledge [32]. What is missing in most of these studies to date is a principled validation of the metrics themselves—for example, to show that the metric values for different rule bases correlate with the difficulty in modifying or debugging the rule bases. Nevertheless, under certain circumstances, measurements obtained from the use of metrics may provide useful implementation-related artifacts.

[7] Questions have arisen as to the precise semantics of some commercial production rule languages [55], but they appear to be amenable to formalization [44].

## 3.5 Verification Analyses

Verification is the process of ensuring that a software system is correct with respect to its requirements [57]. Verification consists of a series of tests, each of which is designed to establish whether the system is correct with respect to certain specified requirements. To be verifiable, each requirement must be associated with some test that produces a boolean result: if the result is true, then the system complies with the requirement; if the result is false, then the system does not comply. For a development process to result in a high-quality product, verification must be performed on all the artifacts which describe the system at different levels. Verification procedures produce definite results which must be recorded; these become additional artifacts to substantiate claims that the quality of the system is adequately high.

Much of the known work in verification of knowledge-based systems has been done on automatic tools which check knowledge bases for the presence of certain anomalous structures, such as redundant or conflicting knowledge—for example, see [10, 20, 31, 37, 45, 53, 61]. Interestingly, little of this work relates verification directly to explicit system requirements; the properties checked for are independent of the application domain or required system functionality. For example, this approach assumes that the presence of conflicting knowledge in *any* knowledge base is an undesirable thing. While early attempts to define a set of generic undesirable knowledge base properties were informal [54], more recent work has lead to formalization of several commonly-accepted properties, including redundant knowledge, conflicting or contradictory knowledge, and missing or deficient knowledge [10, 33, 35, 50, 70].

Formal definitions of such properties can be defined only with respect to a formal representation language for the knowledge base. Most of the published definitions are based upon mathematical logic, which has the advantage that its syntax and semantics are well-known [8]. For example, we offer the following definitions for undesirable properties in knowledge bases expressed in propositional logic, based upon definitions appearing in [50]:

*Redundancy* It should not be possible to remove any expression from the conceptual model without changing the set of inferences that can be drawn from it. For example, consider the following knowledge base (examples here assume that the knowledge base is expressed in propositional logic, with inferences being derived by deduction):

$$p \rightarrow q \qquad q \rightarrow r \qquad p \rightarrow r$$

Here, expression $p \rightarrow r$ is redundant.

*Conflict* It should not be possible to derive incompatible inferences from the knowledge base, given a set of permissible input. For example, consider the knowledge base:

$$p \rightarrow q \qquad q \rightarrow r \qquad p \rightarrow \neg r$$

Here, both $r$ and $\neg r$ will be inferred from input $p$: the knowledge base is in conflict.

*Deficiency* There should be no sets of permissible input for which the knowledge base will fail to derive *any* inferences. For example, consider the knowledge base:

$$p \rightarrow q \qquad \neg p \rightarrow r \qquad q \rightarrow \neg r$$

Here, if $\neg q$ is a permissible input, then the knowledge base is deficient for this input (because it will not infer anything, which suggests that knowledge is missing).

All of these properties can be detected automatically in knowledge bases—the chief difficulty in doing so is the complexity of the computational procedures involved. For propositional logic, the problem of detecting the properties in general is intractable for complex knowledge bases, because of the need to generate and test all possible inferences from the knowledge [45]. For first-order logic, this problem is undecidable, for the same reason [26]. Researchers have attacked this problem in a number of ways, the most successful of which has been to isolate and solve useful special cases of the problem. The approach followed by the COVER knowledge base verification tool [50] is to provide three 'levels' of checking, according to the computational difficulty inherent to the procedures:

*Integrity checking* This involves verifying each knowledge base expression using a set of declaration tables, some of which are provided by the user, and some of which are derived from the knowledge base. This check ensures that there are no 'dead ends' in the knowledge base (rule conditions that can never be satisfied, or rule conclusions that are never used), that no rule has conditions which are trivially inconsistent (for example, $p \wedge \neg p \rightarrow q$), and any other property which can be localized to a single rule. This check can be shown to be of complexity $O(n)$, for $n$ knowledge base expressions [50].

*Expression-pair checking* Special cases of knowledge base redundancy and conflict can be detected by examining pairs of knowledge base expressions only. For example, the following pair of rules are obviously in conflict: $p \rightarrow q$ and $p \rightarrow \neg q$. This check is of complexity $O(n^2)$ for $n$ knowledge base expressions [50].

*Inference chain checking* The most general cases of all the properties listed above can be detected only by generating and analyzing all knowledge base inference chains. For example, to detect that the chain arising from the rules $p \rightarrow q, q \rightarrow r$ is in conflict with the chain from rules $p \rightarrow s, s \rightarrow \neg r$, both chains must be traced and compared. This check is very expensive, computationally, and may be impossible to perform for some knowledge bases; however, there is evidence that it is feasible for a large class of realistic knowledge-based system applications [20, 45].

This type of knowledge-based system verification may be performed on any of the artifacts which describe a knowledge base formally; depending upon the choices of the developers, this may include any or all of the following: conceptual model, design model, and implementation. Experience with such verification has demonstrated its value in revealing knowledge base errors [48]. Therefore, the existence of artifacts which document the results of this kind of knowledge base verification are of value in assessing the quality of the final system.

While the majority of work in the area of knowledge-based systems verification is concerned with checking for the generic properties discussed above, other work has focussed upon verifying that the system complies with application-specific requirements, expressed as formal properties. One approach to this is to express, where appropriate, the minimum performance requirements of the system (stated in the problem specification) formally, and then to prove that the system complies with them. This can be done in certain cases using the method described above, by showing that the knowledge base is logically consistent each minimum performance requirement. For example, if we have a safety requirement that the system must never infer both $q$ and $r$ to be true simultaneously, then our formal constraint is $\neg(q \wedge r)$. If the knowledge base contains the rules $p \rightarrow q$ and $p \rightarrow r$, then we can show that it is in conflict with the constraint using the methods described above.

An alternative approach to verifying such constraints has been proposed for knowledge bases expressed is the type of procedural language used for most production rule systems [17, 54, 67, 69]. This approach is essentially concerned with identifying pre-conditions, post-conditions, and invariants for the knowledge base rules, and with proving that the application of any applicable sequence of rules (a) cannot leave the system in an illegal state, and (b) always leads to correct conclusions, as specified. While interesting, this approach has been applied only to small knowledge bases to date. Clearly, this technique is similar to that used in the formal verification of conventional software systems; conventional approaches have also been applied to verify procedural components of knowledge-based systems, such as the inference engine [30]. This type of verification is necessary for developers of systems where there is a high risk associated with failures of the procedural components—in such cases, evaluators will need to see documentary artifacts which demonstrate that the verification has been performed.

## 3.6 Validation Analyses

Validation is the process of ensuring that a software system satisfies the requirements of its users [57]. Assuming that the requirements stated in the problem specification are correct, then verification is part of validation, concerned with establishing formalized properties of the system. In addition to verification, validation also includes empirical evaluation techniques in which experiments are performed on the system, the results of which are analyzed carefully to decide whether the system is acceptable [1]. The results of these experiments become artifacts generated by the validation process, all of which shed considerable light upon the quality of the system being validated.

The most widely used empirical validation technique is conventional software testing, in which the system is exercised using a suite of test cases, and the acceptability of its behaviour and output is measured. In this section, we will examine three aspects of testing knowledge-based systems: first, we will look at how testing can be applied both to individual system components, as well as to the system as a whole (as an analogue to the idea of module testing versus integration testing in conventional software engineering); second, we examine different criteria by which a set of test cases can be assembled; finally, we discuss the problems in deciding whether the performance of a knowledge-based system on a set of test cases is acceptable. To end this section, we briefly consider empirical validation techniques other than testing.

**Component testing for knowledge-based systems** Most often, knowledge-bases are tested only as 'black boxes'—that is, as an indivisible whole. However, there is potential for component testing or module testing for these systems, in certain circumstances. One definite validation requirement for all knowledge-based systems is to test that the inference engine satisfies the requirements specified in the system design model. As described earlier, depending upon the exact approach followed by the developers, verification of the inference engine may be accomplished either by formal methods or empirical testing. If a third-party commercial tool is used, then testing will be the only available course of action, unless the tool is supplied with some artifacts which certify its properties (for example, proofs of correctness according to some well-defined specification)—this is far beyond the current state of the field.

In addition to performing testing of the inference engine separately, it may be possible to test components of the knowledge base independently, if the problem can be decomposed in a suitable

manner. For example, it may be possible to test task knowledge by providing simulated domain-level knowledge, and vice versa. If the domain knowledge is represented formally at the conceptual or design level, it may also be possible to 'test' this component of the system directly; one approach to this would be to prove that a body of domain knowledge expressed in logic is consistent with the conditions and conclusions of a test case. As a trivial example, if the domain knowledge consists only of the rules $p \rightarrow q$ and $q \rightarrow r$, and we have a test case which says that, for 'input' $p$, the system should produce 'output' $\neg r$, then it is easy to show that the knowledge base is inconsistent with this test case, and is therefore unacceptable. If we perform testing in such a manner, then we can also test whether the implemented system behaves consistently with its higher-level counterparts, by running the same test cases on the implementation and comparing the results with those obtained at the higher level.[8]

**Obtaining a test suite** In recent years, a number of articles and reports have considered criteria and methods for creating a *representative* suite of test cases with which to test a knowledge-based system [38, 71, 54, 56]. A serious difficulty in obtaining such a set is that the diversity of situations in which a knowledge-based system must perform is such that a representative set of cases will likely be very large, even when accounting for the existence of equivalence partitions within the input domain (that is, test cases which should be treated identically by the system) [55]. While some application domains may provide a wealth of documented past cases for use in testing, many domains will not [38]. In these cases, it will be necessary for the validators to create test cases.

The problem of creating a suite of test cases is exacerbated by the fact that each test case is likely to be complex, because a typical knowledge-based system solves complex problems, presented as complex test cases (for a realistic example, see [56]). This means that, not only will it be difficult and time-consuming to create or transcribe each case, but also it may be difficult to specify what constitutes an *acceptable output* for the case. While we cannot avoid the necessity of testing, we can seek to minimize the effort involved by minimizing the number of test cases required.

While random generation of test cases has proven very useful in testing conventional software [16], the complexity of knowledge-based system testing requires a more focussed approach to test case creation. The two available approaches are *structural testing* and *functional testing*. The former involves choosing a set of test cases which exercise as many structural components of the system as possible [25]. Methods proposed for structural testing of rule-based systems are based on the notion of an *execution path* through the rule base—which is similar to the notion of an inference chain, but more general in that it applies to knowledge bases expressed using procedural representation languages as well as those expressed using declarative ones [22]. While most of the proposed approaches are still immature, some success has been reported using this approach [55, 49, 29]. Structural testing of a knowledge-based system implementation can be governed by the structural designs described in the design model. In contrast, functional testing of a knowledge-based system associates the generation of test cases with the problem specification of the system [41]. Test cases must be created for each task that the system is required to perform, and at each one of several possible levels of difficulty, where appropriate. Practical examples demonstrating the efficacy of this approach are presented in [29, 55, 56].

The most practical method for creating a set of test cases would seem to entail a combination of both structural and functional approaches [71, 54]. Recent work has addressed the problem of

[8] I am grateful to my colleague, Neli Zlatareva, for suggesting this idea.

generating test cases according to structural and functional criteria [10, 56]. It seems that tools are also necessary for measuring the adequacy of a set of existing cases; then, the deficiencies in an existing test suite can be identified, according to both structural and functional criteria, and additional cases can be generated to complete the set to the maximum extent possible.

**Judging system acceptability** An unusual difficulty in validating knowledge-based systems stems from the fact that, often, they will be allowed to perform less than perfectly, because humans performing the same tasks are themselves incapable of perfect performance. One repercussion of this fact is that it may be difficult to choose an appropriate level of performance for the system to achieve in order to be accepted. This was our motivation in allowing for *desired performance levels* in the problem specification of a knowledge-based system. A second repercussion is that it may be difficult to define a standard against which to judge the acceptability of the system.

Two standard approaches are described in the literature [38]. The first requires the existence of a *gold standard*: that is, a generally-accepted 'correct' response for every test cases. If a gold standard is available, then testing is similar to verification—each test produces a boolean result, depending on whether the output of the system exactly matches the gold standard or not. In many domains, there is no such standard; instead, a so-called *agreement* method must be employed, where the performance of the system is compared with that of other performers (humans or other systems), and the system is deemed to be acceptable if it 'agrees' with the other performers to a sufficiently high degree. Agreement should be measured using some principled approach—several suitable statistical methods are described in [38, 51]. The most well-known agreement method is based on Turing's famous test for intelligence [64]; in the version employed for validating knowledge-based systems, an independent party of human experts examines anonymous transcripts obtained from having both the system and other humans solve the same problems—if the third-party experts cannot distinguish between the problem-solving abilities of the system and the other humans, then the system is deemed to be acceptable.

**Other empirical validation techniques** In this section, we have been concerned mostly with validating the functional performance of a knowledge-based system. Finally, we acknowledge that non-functional aspects of the system require validation also, notably the user interface [46]. While the methods for performing such validation are no different, in principle, for knowledge-based systems as for conventional software, the complexity of the problem-solving tasks involved may necessitate highly complex and unusual dialogue-based interfaces, presenting great challenges to validators of the 'usability' of the system. Such validation will not be easy because of the subjective nature of the concepts; nevertheless, it must be performed using experimental techniques that are as rigorous as possible. Suitable approaches include use of controlled observation and protocol analysis.

# 4 Assessing and Improving the State-of-the-Practice

The previous section may be viewed as a descriptive survey of the state-of-the-art in specification, verification and validation techniques for knowledge-based systems. Unfortunately, few of these techniques are commonly used by current practitioners, as indicated by recent surveys [24, 39]. In this section, we now propose an adaptation of the *TRILLIUM* model to allow assessment of the

| Level 1 Capability | |
|---|---|
| Problem specification | No explicit statement of requirements. No test plan. No acceptance criteria. |
| Conceptual model | No documented conceptual model. |
| Design model | No documented design model for knowledge base. Possibly documented program design for inference engine or, more likely, rationale for choice of an existing expert system shell. |
| Implemented system | Implemented knowledge base is only complete description of knowledge. Inference engine will likely be that of an existing expert system shell. |
| Verification analyses | Verification performed by informal proofreading—no formal verification analysis conducted. |
| Validation analyses | Validation performed by ad hoc testing and informal evaluations. No permanent recording of test suite. |

Table 2: Expected artifacts produced by knowledge-based systems developers conforming to Level 1 of our adapted *TRILLIUM* model.

capability of knowledge-based system suppliers to produce high-quallity systems. This section can therefore be used in two ways:

- by knowledge-based system customers, to assess the existing capability of developers;
- by knowledge-based system developers, to improve their existing capability.

For each of the first three levels of capability, we indicate the artifacts that should be produced during system development at that level, according to the kinds of artifact described in the previous section. The three capability levels are shown in Table 2, Table 3, and Table 4.

Development at Level 1 is informal, with few artifacts generated except for the implemented system itself. This corresponds to most of the early expert system projects, which were largely concerned with exploring the technology. Development at Level 2 is semiformal; limited attention is paid to specification of the system (independent of the final implementation) and to documented verification and validation analyses. This corresponds to much of today's commercial expert system development practice. Development at Level 3 is our current goal, required in order to bring development of knowledge-based systems more closely into line with conventional software. Rigorous specification, design, verification and validation activities are performed, and all are documented thoroughly. Formal methods are used wherever possible.

It is worth noting that the above proposal is in no way tied to a specific life-cycle model for knowledge-based systems. While it is true that prototyping models are the ones most commonly used in practice, we do not want our adapted *TRILLIUM* model to prescribe a specific life-cycle model. That choice is left to individual developers, chiefly because that choice will often be influenced by external constraints. For example, defense contractors for a particular country are typically bound to follow government-imposed life-cycle standards.[9].

[9]This view is consistent with the original *TRILLIUM* model, in which all that is required for Level 3 capability is that a supplier must use a well-defined life-cycle model, whatever that model may be.

| Level 2 Capability | |
|---|---|
| Problem specification | Informal statement of requirements, test plan, and acceptance criteria. |
| Conceptual model | 'Paper model' stated semiformally. Attempt, where possible, to achieve separation of concerns by isolating domain, task, and co-operative knowledge components. |
| Design model | Architectural design for system components. Semiformal or formal designs for procedural parts of knowledge base, and for inference engine. |
| Implemented system | Implemented knowledge base and inference engine is traceable, where appropriate, to conceptual and design models. If a third-party shell or tool is exploited, steps are taken to ensure that its behavior conforms to that desired. |
| Verification analyses | Knowledge base integrity and expression-pair properties are checked, with all detected anomalies fully documented and resolved. |
| Validation analyses | Testing using documented test suite is performed according to problem specification. Semiformal evaluations of system usability are performed and documented. |

Table 3: Expected artifacts produced by knowledge-based systems developers conforming to Level 2 of our adapted *TRILLIUM* model.

The major difference between the adapted *TRILLIUM* model and the original model is that Level 3 of the latter conforms to the current state of software engineering practice, while we acknowledge that considerable effort is required to bring most current developers up to Level 3 of the model. The main reasons for this are the limited availability and lack of maturity of the tools and personnel necessary to achieve Level 3 development capability. Nevertheless, given the state of the art described in the previous section, it is reasonable to expect this situation to improve greatly within just a few years. It seems clear that, as more and more developers work towards achieving Level 3 capability, the resulting feedback to researchers will lead to further improvements in the support techniques for knowledge-based system specification, design, verification and validation.

**Acknowledgements**

The work described in this article was carried out at the Centre for Pattern Recognition and Machine Intelligence (CENPARMI), Concordia University, Montréal, Canada. The views and ideas expressed in this article owe much to the author's discussions and collaborations with his colleagues: Aïda Batarekh, Anne Bennett, Gokul Chander, Peter Grogono, Cliff Grossner, Rajjan Shinghal, Ching Suen, and Neli Zlatareva. Special thanks are due to Alan Bloch, for assembling and managing one of the largest bibliographies in the field.

| Level 3 Capability | |
|---|---|
| Problem specification | Semiformal statement of requirements, including minimum and desired functionality. Formal constraints associated with all possible minimum requirements. Test plan and acceptance criteria associated with each functional requirement that cannot be verified formally. |
| Conceptual model | Formal knowledge-level model (that is, with well-defined syntax and semantics). Where appropriate, different representation languages may be used for domain, task, and cooperative knowledge base components. |
| Design model | Formal architectural design, module interface specifications, and internal module designs for all system components, including inference engine, domain knowledge modules, task knowledge modules, meta-level control knowledge modules, and external interface components. |
| Implemented system | Implemented system is fully traceable to conceptual and design models or, preferably, is derived automatically from them using some correctness-preserving transformation procedure. A third-party tool may be used only if rigorous assurances are available of its correctness and reliability. |
| Verification analyses | Full inference chain properties are checked and all anomalies are documented and resolved. System compliance with all minimum-performance constraints is verified and documented if possible. |
| Validation analyses | Rigorous structural and functional testing is performed as per problem specification. Test suite is executed and maintained using support tools. Approved empirical methods are used for usability and utility evaluations, and results are fully documented. |

Table 4: Expected artifacts produced by knowledge-based systems developers conforming to Level 3 of our adapted *TRILLIUM* model.

# References


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

[2] M. Ayel and J.-P. Laurent, editors. *Verification, Validation and Test of Knowledge-based Systems*. John Wiley and Sons, 1991.

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

[4] W. Behrendt, S. C. Lambert, G. A. Ringland, P. Hughes, and K. Poulter. Gateway: Metrics for knowledge based systems. In J. Liebowitz, editor, *Expert Systems World Congress Proceedings* (Orlando, December 16–19, 1991), volume 2, pages 1056–1067, Pergamon Press, New York, December 1991.

[5] Bell Canada Quality Assurance. TRILLIUM: Telecom software product development capability assessment model. Technical Report Draft 2.2, Bell Canada, July 1992.

[6] G. Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, 12(12):211–221, December 1986.

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

[8] A. Bundy. How to improve the reliability of expert systems. In D. S. Moralee, editor, *Research and Development in Expert Systems IV: Proc. Expert Systems 87* (British Computer Society Specialist Group on Expert Systems, Brighton, December 14–17, 1987), pages 3–17, Cambridge University Press, 1988.

[9] B. Chandrasekaran. Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert (US)*, 1(3):23–30, Fall 1986.

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

[11] W. J. Clancey. Heuristic classification. *Artificial Intelligence (Netherlands)*, 27:289–350, 1985.

[12] W. J. Clancey. Model construction operators. *Artificial Intelligence (Netherlands)*, 53:1–115, 1992.

[13] F. Coallier, P. N. Robillard, A. Beaucage, M. Simoneau, D. Coupal, J.-B. Trouvé, and A. Grenier. DATRIX: A software workmanship evaluation tool. Draft, 1989.

[14] C. Culbert. Special issue on verification and validation. *Expert Systems with Applications*, 1(3), 1990.

[15] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, 1978.

[16] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering (US)*, SE–10(4):438–443, 1984.

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

[18] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, New York, 1991.

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

[20] A. Ginsberg. Knowledge-base reduction: A new approach to checking knowledge bases for inconsistency & redundancy. In *Proc. 7th National Conference on Artificial Intelligence (AAAI 88)* (St. Paul MN), volume 2, pages 585–589, August 1988.

[21] P. Grogono, A. Batarekh, A. Preece, R. Shinghal, and C. Suen. Expert system evaluation techniques: a selected bibliography. *Expert Systems (UK)*, 8(4):227–239, November 1991.

[22] C. Grossner, A. Preece, P. G. Chander, T. Radhakrishnan, and C. Y. Suen. Exploring the structure of rule based systems. In *Proc. 11th National Conference on Artificial Intelligence (AAAI 93)* (Washington DC, July 11–16), 1993. To appear.

[23] U. G. Gupta. *Validating and Verifying Knowledge-based Systems.* IEEE Press, Los Alamitos, CA, 1990.

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

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

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

[27] S. H. Kaisler. Expert system metrics. In *Proc. 1986 IEEE International Conference on Systems, Man, and Cybernetics* (IEEE, Atlanta GA, October 14–17, 1986), volume 1, pages 114–120, 1986.

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

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

[30] P. J. Krause, P. Byers, S. Hajnal, and J. Fox. The use of object-oriented process specification for the verification and validation of decision support systems. In *ECAI '90 Workshop on Validation, Verification and Test of KBS* (Stockholm, Sweden, August 7th, 1990), 1990.

[31] J. P. Laurent and M. Ayel. Off-line coherence checking for knowledge based systems. In *IJCAI-89 Workshop on Verification, Validation and Testing of Knowledge-Based Systems* (IJCAI, Detroit, August 19, 1989), 1989.

[32] M. Mehrotra and S. C. Johnson. Importance of rule groupings in verification of expert systems. In C. Culbert, editor, *AAAI-90 Workshop on Knowledge Based Systems Verification, Validation and Testing* (AAAI, Boston MA, July 29, 1990), July 1990. Unpublished Workshop Notes.

[33] P. Meseguer. Incremental verification of rule-based expert systems. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 92)* (European Coordinating Committee for Artificial Intelligence (ECCAI), Vienna, Austria, August 3–7), John Wiley & Sons, New York, 1992.

[34] W. Mettrey. A comparative evaluation of expert system tools. *IEEE Computer (US)*, 24(2):19–32, 1991.

[35] D. L. Nazareth. Issues in the verification of knowledge in rule-based systems. *International Journal of Man-Machine Studies (UK)*, 30(3):255–271, March 1989.

[36] A. Newell. The knowledge level. *AI Magazine (US)*, 2(2):1–20, Summer 1981.

[37] T. A. Nguyen, W. A. Perkins, T. J. Laffey, and D. Pecora. Checking an expert systems knowledge base for consistency and completeness. In *Proc. 9th International Joint Conference on Artificial Intelligence (IJCAI 85)* (AAAI, Los Angeles CA, August 18–23, 1985), volume 1, pages 375–378, 1985.

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

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

[40] D. E. O'Leary. Toward a theory of verification and validation: A theory of artifacts. Presented at the Verification and Validation of Expert Systems Workshop held at AAAI-92 (San Jose CA, July 16th 1992), 1992.

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

[42] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM (US)*, 15(12):1053–1058, December 1972.

[43] M. C. Paulk. Capability maturity model for software. Technical Report CMU/SEI-91-TR-00 & CMU/SEI-91-TR-24, Carnegie Mellon University Software Engineering Institute, 1991.

[44] R. T. Plant and D. Gold. Increasing expert system reliability through the use of a formal specification. In C. Culbert, editor, *AAAI-90 Workshop on Knowledge Based Systems Verification, Validation and Testing* (AAAI, Boston MA, July 29, 1990), July 1990. Unpublished Workshop Notes.

[45] A. Preece and R. Shinghal. Verifying knowledge bases by anomaly detection: An experience report. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 92)* (European Coordinating Committee for Artificial Intelligence (ECCAI), Vienna, Austria, August 3–7), John Wiley & Sons, New York, 1992.

[46] A. D. Preece. DISPLAN: Designing a usable medical expert system. In D. Berry and A. Hart, editors, *Expert Systems: Human Issues*, pages 25–47. MIT Press, Boston MA, 1990.

[47] A. D. Preece. Verifying expert systems using conceptual models. In *Proceedings of 5th Annual Knowledge-Based Software Assistant (KBSA) Conference* (Syracuse, NY, September 24–28, 1990), pages 280–289, Rome Air Development Center (RADC), New York, Griffiss AFB, NY, 1990.

[48] A. D. Preece, R. D. Bell, and C. Y. Suen. Verifying knowledge-based systems using the cover tool. In F. H. Vogt, editor, *Personal Computers and Intelligent Systems (Information Processing 92, Volume III): Proceedings of the IFIP 12th W orld Computer Congress* (Madrid, Spain, September 7–11), pages 231–237, Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1992.

[49] A. D. Preece, C. Grossner, P. G. Chander, and T. Radhakrishnan. Structural validation of expert systems: experience using a formal model. In A. D. Preece, editor, *Validation and Verification of Knowledge-Based Systems (AAAI-93 Workshop Notes)* (AAAI, Washington DC, July 12), pages 19–26, 1993. AAAI Press Technical Report.

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

[51] J. A. Reggia. 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, 1985. Also in Miller, Perry L., ed., *Selected Topics in Medical AI*, New York, Springer, 1988, pp. 222–230.

[52] J. A. Reggia, D. S. Nau, and P. Y. Wang. Diagnostic expert systems based on a set-covering model. *International Journal of Man-Machine Studies (UK)*, 19:437–460, 1983.

[53] M.-C. Rousset. On the consistency of knowledge bases: the COVADIS system. *Computational Intelligence (Canada)*, 4(2):166–170, May 1988. Also in *ECAI 88, Proc. European Conference on AI* (Munich, August 1–5, 1988), pages 79–84.

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

[55] J. Rushby and J. Crow. 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, February 1990. 93 pages.

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

[57] Software Engineering Technical Committee of the IEEE Computer Society. *IEEE Standard for Software Verification and Validation Plans.* The Institute of Electrical and Electronics Engineers, New York, November 14 1986. IEEE Std 1012-1986.

[58] J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, Reading, MA, 1984.

[59] J. Spivey. *Understanding Z, a Specification Language and its Semantics.* Cambridge University Press, 1989.

[60] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Boston MA, 1986.

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

[62] The Institute of Electrical and Electronics Engineers. *Software Engineering Standards.* IEEE Press, New York, 3rd edition, 1989.

[63] W. T. Tsai, K. G. Heisler, D. Volovik, and I. A. Zualkernan. A critical look at the relationship between AI and software engineering. In *1988 IEEE Workshop on Languages for Automation: Symbiotic and Intelligent Robots* (IEEE, College Park MD, August 29–31, 1988), pages 2–18, 1988.

[64] A. Turing. Computing machinery and intelligence. *Mind*, 59:236–248, 1950.

[65] F. van Harmelen and J. Balder. $(ML)^2$: a formal language for KADS models of expertise. *Knowledge Acquisition*, 4(1):127–161, 1992.

[66] I. van Langevelde, A. Philipsen, and J. Treur. Formal specification of compositional architectures. Technical Report IR-282, Faculteit der Wiskunde en Informatica, Vrije Universitat Amsterdam, Netherlands, 1991.

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

[68] B. J. Wielinga, A. T. Schreiber, and J. A. Breuker. KADS: a modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1):5–54, 1992.

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

[70] N. Zlatareva and A. Preece. An effective logical framework for knowledge-based systems verification. *International Journal of Expert Systems: Research and Applications*, 1993. Under review.

[71] N. Zlatareva and A. Preece. State of the art in automated validation of knowledge-based systems. *Expert Systems with Applications (US)*, 7(2), 1993. To appear.