Factors in software quality for knowledge-based systems

R T Plant

The paper considers the need for quality knowledge-based software systems. The paper defines quality in terms of a manufacturing process and then relates the creation of software to manufacturing. It then considers the aspects that go towards the creation of a quality knowledge-based system: the specification, the development methodology, and the validation and verification processes that check that a product meets its specification. The paper does this by considering a set of benchmarks against which the level of quality can be measured and then how each of the aspects contributes to heighten each of the factors that contribute to the level of quality. The paper proposes that the specification, methodology, and validation criteria can all heighten the level of correctness and that if they are used collectively then the correctness of a system and hence the quality level can be raised significantly.

software quality, knowledge-based systems, specification, methodologies, validation, verification, software testing

This paper discusses the need for software quality in the area of knowledge-based systems. These are systems that attempt to perform at the same level of performance as a human expert over a given domain. However, they differ from the traditional procedural software systems in the type of domain they attempt to model and the techniques through which these models are created. The primary consideration that distinguishes knowledge-based systems from their more traditional counterparts is that their domains cannot be fully specified before the creation of the system and thus cannot be argued about in the formal styles available to the conventional software developer.

This inability to specify fully the systems has impeded the creation of adequate development models for knowledge-based systems, whose declarative nature does not lend them to creation through the conventional software development methodologies. These two factors have been the primary cause for the creation of poor-quality knowledge-based systems that are of low reliability and of consequently limited use. Therefore the areas of specification and methodology are considered in conjunction with other techniques to raise the level of quality of knowledge-based systems.

Department of Computer Information Systems, University of Miami, Coral Gables, FL 33124, USA

DEFINITIONS OF QUALITY

The term quality has been defined in many different ways, for example:

> 'Quality is a judgement by customers or users of a product or service; it is the extent to which the customers or users believe the product or service surpasses their needs and expectations.'[1]

This defines quality in terms associated with a customer's or user's perception of its worth, with implicit reference to other products of a similar nature.

It is possible to go further and define quality in a more specific way:

> 'Quality is conformance to requirements. Deviation from specification implies a reduction in quality.'[2]

Here the definition has associated with it a context based on a manufacturing approach.

This definition can be used to consider the problem of quality as related to software, as the production of software can be viewed as a manufacturing process. A definition that helps do this is:

> 'Testing is a measurement of software quality.'[3]

Hetzel goes further and provides a useful working definition of testing:

> 'Testing is any activity aimed at evaluating an attribute or capability of a program or system.'[3]

Thus a relationship between testing and quality can be seen. To have a quality product, which in this case is software, it needs to be ensured that it meets its requirements. This can be considered a validation and verification process and can be approached through the use of testing, where the use of more accurate testing mechanisms leads to increases in the level of system correctness and hence increases in quality.

QUALITY-ASSURANCE MEASURES

To assess the quality level a software system has reached, there need to be benchmarks against which the quality level can be measured. The benchmarks can be used for both knowledge-based and conventional systems; how-

 0950-5849/91/070527-10 © 1991 Butterworth-Heinemann Ltd

ever, the imprecise nature of knowledge-based systems makes it more difficult for these systems to have metrics applied to them and an analysis of the results made. For example, knowledge-based systems cannot be precisely specified and as such the correctness of a system becomes difficult to assess, as the correctness of a system can only be assessed when the system is measured against its specification. However, the use of quality metrics is still of importance for knowledge-based systems, and it is useful to survey some of the approaches available.

A set of quality factors has been defined by Garvin[2], who sets out his 'eight dimensions of quality' as:

- performance
- features
- reliability
- conformance
- durability
- serviceability
- aesthetics
- perceived quality

He uses these factors to determine the quality of a product, e.g., a program, in relation to other products that perform a similar or identical role, and as such the factors are at a higher level than those necessary to assess the quality of an individual knowledge-based software system. They would be useful, however, for assessing the variance between two or more expert system shells, for example.

A set of benchmarks that attempts to measure the level of quality a software system has attained, by measuring a set of attributes associated with the software has been proposed by Carpenter and Murine[4], who put forward a software quality–assurance (SQA) methodology; these attributes have been termed 'quality factors'. The 12 factors proposed by Carpenter and Murine[4] are:

- correctness
- reliability
- efficiency
- integrity
- reusability
- useability
- maintainability
- testability
- flexibility
- portability
- interoperability
- intraoperability

Carpenter and Murine define these factors and state that there are metrics available to measure them (they do not, however, define these metrics). They also state a useful axiom that the weighting associated with each factor is not going to be equal in value, a problem that has to be addressed with all metric-based quality models, whether applied to knowledge-based or conventional software.

In an important paper, Boehm also attempted to define software quality in terms of seven software characteristics[5]:

- reliability
- portability
- efficiency
- human engineering
- testability
- understandability
- modifiability

As Conte notes, however, 'precise definitions of these subjective characteristics are very difficult'[6]. Conte makes three points to illustrate the difficulty of associating quality measurements with software. First, some of the characteristics are potentially contradictory. Second, there are significant cost-benefit trade-offs that must be considered in attempting to maximize any particular characteristic. Third, it may be difficult to define a particular metric to measure a particular characteristic.

An alternative set of SQA measures has been given by Dunn[7]:

- ensuring compliance to defined standards
- tracking corrective action
- reliability analysis
- measurements
- customer (or user) feedback
- pareto analysis
- vector surveys and vendor surveillance
- product qualification
- quality improvement

In this set of factors, several important characteristics of any effective QA methodology can be identified: standards. In many organizations, the software is created by following a series of guidelines or 'house-standards' that lay down definitive guidelines for the specification, design, implementation, and maintenance standards required by that organization. The aim of these standards is to promote the production of quality software and in doing so to ensure that all developers use the same approaches, thus avoiding a plethora of styles, methodologies, and strategies for development. In many applications the software created is of a critical nature, for example, defence, aeronautics, and energy systems; it is therefore in these areas that the most rigorous standard definitions have been developed (see Table 1). Software quality standards have been reviewed[8,9].

These standards and reports define the approaches and procedures to be taken when performing a certain aspect of the software development task. The reports, however, focus on the task of developing conventional, procedural, algorithmic software, and as such their applicability to the creation of knowledge-based software is limited. An example of this is the DOD-2167A military standard, which:

> 'Establishes uniform requirements for software development throughout the system life-cycle'

and even though the standard states that it does not intend to discourage use of any particular software development method, such as rapid prototyping, the findings

Table 1. Standard definitions and reports on software quality

NATO
AQAP 14 1981
'Software quality control requirements'
AQAP-14 1984
'Guide for the evaluation of a contractor's software quality control system for compliance'
ANSI/IEEE
824 1983
'Standard for software configuration management'
730-1984
'Standard for software quality assurance plans'
829 1985
'Standard for software documentation'
983 1986
'Guide for software quality assurance planning'
DOD
MIL S-52779A 1979
'Software quality assurance program requirements'
MIL-HDBK 334-1981
'Evaluation of a contractor's software quality assurance program'
DI R 3521-1892
'Software quality assurance plans'
NRC
EPRI NP 5236/1987
'Approaches to the verification and validation of expert systems for nuclear power plants'
EPRI NP 5978/1987
'Verification and validation of expert systems for nuclear power plant applications'
NSAC-39/1981
'Verification and validation for safety parameter display system'
NUREG/CR-4640/1987
'Handbook of software quality assurance techniques applicable to the nuclear industry'
NUREG-0653/1980
'Report on nuclear industry quality assurance procedures for safety analysis computer code development and use'

by practitioners in the field of knowledge-based systems development are that:

> 'Knowledge-based systems applications are almost never based on a solid definite set of requirements; they are usually somewhat ill defined, and they almost always change significantly during development. In any event, even if there were stable requirements the knowledge-based contents are inherently not easily decomposable into separate and independent functional components. Rather knowledge-based elements typically are employed for multiple functions and purposes.'

and thus it is Miller's conclusion that:

> 'The standard 2167A life cycle thus seems ill suited for knowledge-based system development.'[10]

The creation of knowledge-based systems therefore needs to be regarded as a special case within software development, and as such these systems require special metrics and effort to achieve satisfactory quality levels. This has necessitated the IEEE and AIAA to develop new standards solely for knowledge-based systems[11].

The tracking of corrective actions taken by a developer entails the auditing of error maintenance. This is an aspect of software quality that can be of significant benefit, yet one that is too often neglected. It has been shown by Boehm that 25% of all software defects can be attributable to defects in the documentation deliverable to customers[12]. Thus the tracking of error correction and formalization of update procedures can significantly raise the level of software quality. The relation of this corrective action tracking to knowledge-based systems does, however, run across several problems from the standpoint of documentation. The level of documentation is either ineffective, such as that found in several of the military standards, where the focus is intended for algorithmic procedural systems, or of limited applicability to error tracking, as the development methodologies typically do not enforce formalized document requirements. Thus, when considering an aspect of software quality for knowledge-based systems that could be of substantial benefit, the employment of rigorous documentation standards into a development methodology could be advantageous.

The use of software metrics enables a series of measurements to be collected, by which the developer can form a parametric model of the system under consideration. The model can then be used as a gauge for alterations to the system and to view whether improvements in quality have been achieved. Central to these metrics is the measurement of software failures, and through this system reliability can be considered. The area of software reliability has been covered extensively[13-15], however, the focus has been on the reliability of conventional, procedural, deterministic, algorithmic systems and as such is of limited applicability to the domain of knowledge-based systems. There has been a move in current research towards developing a theory of knowledge-based system reliability[16,17]. This work is in the early stages, but should be of significant benefit to knowledge engineers on maturity.

A technique that can be used to raise the level of quality of a system as quickly as possible is that of pareto analysis. The principle of this is based around the idea of 'the vital few versus the trivial many', e.g., most problems in a system emanate from a relatively small number of significant faults, and once these have been solved there will remain only a small number of problems of a trivial nature[18]. Lakelin's rule can be applied to the application of the pareto principle with regard to software[19], in that 80% of the errors result from 20% of the faults in the code[20]. To isolate the errors and judge their significance, a pareto diagram can be constructed, which is a bar chart in which the frequency of error types is plotted[1]. This assumes the availability of testing techniques, for which knowledge-based systems require alternative strategies from those associated with the testing of conventional systems. These will be considered in the following section.

It can be seen from the models discussed above (and in other software quality models[21,22]) that the keys to achieving favourable quality factors in knowledge-based systems are the employment of validation, verification, and testing techniques in association with a rigorous development methodology that uses specification wherever possible, and these aspects of QA are considered in the remainder of the paper.

VALIDATION, VERIFICATION, AND TESTING

Validation and verification have been defined as follows:

> 'Validation: the process of evaluating software at the end of the software process to ensure compliance with software requirements.'[12]
> 'Verification: the process of determining whether or not the products of a given phase of the software development cycle fulfil the requirements established during the previous phase.'[12]

Thus these definitions closely relate to those that define quality.

It follows, therefore, that one of the keys to effective evaluation of the software and consequently to having valid and verified knowledge-based software is to have effective testing techniques available.

Several approaches to the testing of knowledge-based systems can be used:

- traditional approaches
  - dynamic testing
  - static testing
- formal approaches
  - specification
  - proof systems
- artificial-intelligence approaches
  - prototyping
  - certainty factors
- quantitative validation
  - paired t-tests
  - Hotelling's one-sample T^2 test
  - simultaneous confidence intervals

Each of these areas is now considered.

Testing

The testing of a program can be defined as the application of test data (input) to the program to examine the correctness of the output with respect to the function of the program over that input.

Output = Program(Input)

This testing function can be used in several ways. The most obvious way to achieve correctness would be to test exhaustively every possible input against its output. This is of course not feasible for any but the most trivial of expert or knowledge-based system.

There is therefore a limit to the number of tests that can be performed and, in the interest of maximizing the return on the time spent testing, criteria must be looked for with which to test by. There are two primary testing strategies that are used by knowledge engineers: dynamic testing and static testing.

Dynamic testing refers to those techniques that necessitate observation of the behaviour of a system in execution, while static testing is that which depends only on scrutiny of the program or system text.

Examples of dynamic testing include:

- sensitivity analysis
- regression testing
- statistical analysis
- random testing

Examples of static testing include:

- structured walk-through
- mathematical validation
- anomaly detection
- fault tree analysis

First, consider dynamic testing where sensitivity analysis uses test data to determine if similar input data produce significantly divergent results, a potential indication of instability or fragile behaviour. This technique is closely related to the use of multiple sets of identical input data that attempt to search multiple paths through the system for possible redundancy or problems in conflict resolution. The underlying assumption associated with sensitivity analysis testing is that 'small variations in input should produce small consequent variations in output'[23]. Thus knowledge engineers have to be critical of their test data and the resultant output, performing a suitable statistical distribution of the test data to meet their sensitivity requirements. A variation of sensitivity testing that can be performed on knowledge-based systems is the variance of confidence factors, if used in a system, and examining the effect that this has on the stability of the system.

The dynamic testing strategy of regression can be used to effect after the location of an error, in that it demands, in the use of strict regression, that all previous test cases be reapplied. This is of course an expensive overhead to impose on the testing scheme and it is often the case that critical subsets of data are requested[24]. This approach has been considered in relation to knowledge-based systems by Downs[25], while other researchers are working on regression testing tools, such as Scamboros's scenario-based test-tool for examining knowledge-based expert systems[26]. Tools such as this will reduce the cost of regression testing, encourage its use in relation to sensitive and critical testing, and thus increase the quality of the systems.

The use of statistical analysis of a software systems performance is another mechanism through which a system can be tested, and this has been examined in relation to knowledge-based systems[27]. Three techniques are generally used: paired t-tests, Hotelling's one-sample T^2 test, and simultaneous confidence intervals[28]. These tests allow the knowledge engineer to compare the difference between the results of the system and another source, such as the performance of a human problem-solver. This technique, usually in its simplest form where a direct comparison between the results produced by the system and the results from an expert are made, has been extensively used in the area of knowledge-based system testing[29].

The random application of test data to a system to ascertain its correctness has been shown by Currit, Dyer and Mills to be as much as 30 times as effective as other testing mechanisms[30], such as structural testing. They indicate the reason for this conclusion to be the enormous variation in the rate at which errors lead to failures. The random test data approach is one of a set of approaches known as case-based approaches, each of which attempts to test a system's efficiency by placing a focused emphasis on the cases presented. For example, test data could be compiled to test the structure of the inference engine, data to test the systems functionality, etc. Several case-based approaches of this type have been proposed:

- functional
- structural
- data
- random
- extracted
- extreme

One of the most useful is that of extreme case testing, which helps the knowledge engineer examine the boundary conditions of the knowledge-based system under test. This is one of the most difficult aspects of testing for knowledge-based systems as the absence of a full specification makes it difficult to be precise when reasoning about boundary conditions.

While dynamic testing tends towards addressing the problems associated with validation, the static testing mechanisms focus on the problem of verification, with the aim of placing the system under review to locate inconsistencies and omissions.

A widely used static testing strategy is that of the structured walk-through. This entails the detailed examination of the specification, the code, or a model of the system, depending on the level required or the problem to be addressed. Various strategies can be taken towards the walk-through, including the creation of a software quality circle[31]. The use of the walk-through can be applied best when applied to a high-level model of the system behaviour, rather than, for example, examination of the Lisp code, as the models allow the deep knowledge to be more clearly represented[32]. Further, while the declarative nature of many of the representations used in knowledge-based systems facilitates advantages such as modularity, it has the disadvantage that the reasoning may be difficult to follow easily, thus not facilitating structured walk-throughs. The use and creation of trace and debugging tools in the development environments is an approach towards easing this problem.

A technique that is related to structured walk-through is that of anomaly detection, which entails examination of the system for consistency and completeness. This can be performed at a variety of levels, but is usually based on code inspections, which in a knowledge-based system causes significant problems due to the lack of redundant code that usually has no typing mechanisms or few facilities for control or data structuring. To overcome this problem, there is a move towards toolsets to assist in the checking process; one such system is the 'Lockheed Expert System' shell[33].

The use of knowledge-based systems in critical- and safety-oriented environments has promoted the use of testing techniques that aim to reduce the occurrence of failures. One technique that is concerned with this is that of software fault tree analysis. This attempts to show that the logic or design of a system will in some way produce failures that are critical or not safe. This is based on the principle of hazard analysis and has been documented by Leveson[34-36]. The use of software fault tree analysis can be at many levels of abstraction, from the code upwards, and is of significant benefit to systems developers in areas such as knowledge-based systems where it is difficult to apply directly existing reliability models[17].

Testing strategies outlined above each examines a different aspect of the system and are collectively valuable in raising the level of system correctness. However, note that even when all the strategies are used together this does not guarantee total correctness. Further discussion of testing can be found elsewhere[23,37,38].

Prototyping

The prototyping approach to software development is not strictly a testing mechanism[39]. It can be used, however, to test ideas and aspects of the system design that could not be practically tested in a full-size implementation, or aspects that are difficult to theorize about/test without a working system. This includes experimenting with different representations, inference architectures, shells, certainty factors, etc. If the prototyping mechanism is used constructively as part of a complete methodology then the conceptual testing at this stage can considerably benefit the level of correctness achieved in the final system and hence the quality of that system. However, should the prototyping approach be abused, such as when the prototyping system is continued on to be the final system, then this can lead to a poorly structured, ill-designed system which can be extremely difficult to reason about, leading to a system of limited quality that may be difficult to maintain and result in a system with diminishing quality levels. Alavi presents a useful assessment of the prototyping approach to information systems development[40].

Formalized specifications

The ability to show that a product meets its requirements was stated earlier in the paper by Garvin as a means of demonstrating quality, assuming that the product requirements are satisfactory. A means by which this can be achieved for software is through the use of formalized specification techniques.

There are currently three approaches to specification:

- the use of a logic programming language such as Prolog[41] or ML[42]

- using an executable functional specification language such as Miranda[43] or KRC[44]
- the use of a formal specification language such as Z[45] or VDM[46]

Each of these formalized styles of development has led to successfully specified systems in conventional domains, and each has been or could be applied to knowledge-based systems domains to differing degrees. Now each approach is briefly considered.

The use of logic as a means of specification is well known to computer scientists and is documented in seminal papers[47–49]. These papers give a basis to the movement away from 'trial and error' programming to the development of programs that can be proved to have the desired capabilities — hence quality systems. Thus from the use of formal logic in specifying and proving programs correct to the use of a logic programming language as a specification language is but a short step. A keen proponent of Prolog and logic programming as a vehicle for programming and specification is Kowalski who states:

> '[Formal logic] is ideally suited to the representation of knowledge and the description of problems without regard to the choice of programming language. Its use as a specification language is compatible not only with conventional programming languages but also with programming languages based entirely on logic itself.'[50]

Thus the use of logic as a vehicle for specifying knowledge-based systems can be seen. Kowalski amplifies this:

> 'In many cases, when a specification completely defines the relations to be computed, there is no syntactic distinction between specification and program. Moreover the same mechanism that is used to execute logic programs, namely automated deduction, can also be used to execute logic specifications. Thus, all relations defined by complete specifications are executable. The only difference between a complete specification and a program is one of efficiency. A program is more efficient than a specification.'[50]

It is therefore possible to achieve a specification in logical terms for conventional and knowledge-based systems based on the assumption that all the relations to be computed can be completely defined, a process that is significantly more challenging for knowledge-based systems than for the relatively well defined domains of conventional applications.

An aspect of logical specification techniques that cannot be ignored and that was mentioned earlier is that of prototyping. To achieve quality systems, the benefits gained through specification must not be negated by employing a 'trial and error' approach to development. Thus care must be taken in the creation of the systems, developing and following a suitably rigorous methodology.

A further class of executable specifications are those based on functional programming languages. A functional program, or as it is sometimes termed 'script', is a series of recursive equations that are based on the mathematical idea of a function f, where for a given input x to that function, the output f x is always the same. Functional programs have several advantageous properties not in conventional imperative languages. For example, the equations have referential transparency, equivalent equations possess the property of extensionality, the underlying recursive nature of the equations lends them to proof through inductive means, and the programs can be transformed through refinement transformations. Thus functional applicative languages are powerful yet flexible forms through which domains can be specified.

Functional specifications fall into the category of executable specification systems and are subject to the same prototyping implementation-dependent problems as logic specifications. A practitioner in the field who uses Prolog states:

> 'Specifying and modeling the deeper levels of the system got increasingly more difficult to keep crisp and consistent and avoid an ever growing collection of ad-hoc procedures usable at one and only one place.'[51]

Similar arguments can be applied to executable specifications as they are also restrictive in the type of specification that easily adheres to the functional notation. Turner states:

> 'A functional language when considered as a specification language suffers from the restrictions inherent in being recursive: only computable functions can be denoted, so there are some useful and interesting specifications that can not be expressed within it.'[43]

An implication of Turner's statement is that the incomplete, heuristic, nonfunctional nature of many knowledge-based systems may make a full specification in a functional language a difficult process.

The third type of specification is that of the formal approach and is characterized by such languages as VDM[46] and Z[45]. These languages attempt to give a mathematical framework around which specifications can be developed. A formal specification is a declaration of what the system is required to do and not an algorithm of how to do that task. Thus formal specifications are not themselves executable, but there is ongoing research to develop techniques that will enable specifications, through a series of formal (hence provable) 'data refinement' steps to be turned into more concrete and ultimately executable forms. Formal methods have in the past been criticized for only being applicable to 'toy' examples. However, research has been performed to alleviate these criticisms and both VDM and Z have been used in documented large-scale projects, such as the formal specification of IBM's CICS system in Z[52].

The use of formal techniques in knowledge-based systems has not, however, been so straightforward. The major constraining factors are associated with the difficulty of specifying domains that are incomplete, nonfinite, and poorly defined in nature, unlike their conventional counterparts. This is not to say that the formal methods cannot be used at all in developing specifications for knowledge-based systems. It is possible to use these techniques for certain aspects of the system that are

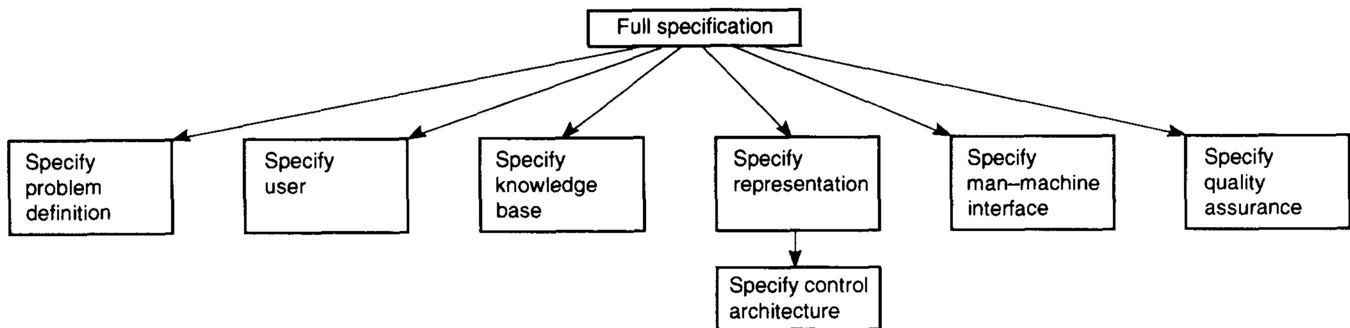

Figure 1. Multiple specialized specifications that can be combined to form composite specification

not inherently knowledge-based. For example, the knowledge base[53], the user interface[54,55], and the representation[56] can be specified formally in a language such as Z. Note that these specifications are of the static aspects of the system rather than the dynamic aspects, but that their use can considerably raise the level of correctness for a system and consequently its quality.

Later it will be suggested how these approaches to specification can be used through a development methodology for knowledge-based systems to raise collectively the correctness of such systems.

METHODOLOGY

A methodology that the author advocates is based around the use of multiple specialized specifications that when combined together can be thought of as a composite specification.

Consider Figure 1. The knowledge engineer can specify seven areas of the system. The first is the specification of the problem definition, the production of which is extremely difficult for nontrivial knowledge-based systems due to their inherent lack of procedural, deterministic, algorithmic structure.

The second area where specification is needed is that of the intended user; this can be performed through the creation of a behavioural model.

The third part is the specification of the knowledge base. It is possible to model this aspect, as the knowledge elicited from the domain expert/knowledge source is finite. Through the use of transformational processes this can be specified formally. The specification of the knowledge base is vital if the system is to be maintained, while the representational independence of the specification promotes clarity and flexibility.

Fourth, it is vital that a suitable representation is selected, and this is done by analysing the representational needs of the knowledge base. Once selected, the syntax and semantics of the representation can be specified. Having specified the representation it is then possible to select an appropriate control architecture, the operation of which can also be specified.

The fifth aspect that needs to be specified is the man—machine interface. This can be fully specified through the use of formal techniques, and several examples of such specifications have been documented[54,55]. The specification of the interface allows the knowledge engineer and user to have an unambiguous frame of reference, through which interactions with the system can be viewed.

The sixth component of the composite specification is the need to specify the validation and verification requirements, the definition of which will enable the knowledge engineer to judge whether the levels of quality reached are adequate for system use.

Therefore several aspects of an expert system can be formally specified, each of which is fundamental to its construction. It is briefly described how, from an initial specification of the problem definition, these points can be rigorously reached and combined together to form a concrete specification from which the system can be implemented.

The methodology as a whole can be introduced by considering Figure 2. The development commences with an initial specification, which acts as an informal software requirements document. This gives a broad outline of the systems parameters and boundaries, to be used by the knowledge engineer as the basis of both the knowledge elicitation phase and the creation of the user model.

In the knowledge elicitation phase the most suitable knowledge elicitation technique with which to extract knowledge from the domain expert is selected. The knowledge engineer then uses this extracted knowledge as the basis of the elicited representation, an unprocessed representation that usually has a textual form. The elicited representation, however, is too coarse in nature to act as the specification for an implementation, and so it is necessary for the representation to undergo a refinement process. The result of this is a more adequate representation, termed the primary representation. It is adequate in the sense that an adequate level of completeness and consistency has been reached, to allow major knowledge processing of the representation to be performed. An example of such a representation is a decision table.

The first process is to transform the primary representation into a formal representation of the knowledge base, this being a mathematical specification written in the Z specification language. The second process is an analysis that examines what constituent characteristics are present in the primary representation, before attempting to match these with the characteristics of the 'classical' representations such as frames, production systems, and semantic networks. From this matching process, a specification of a suitable representation lan-

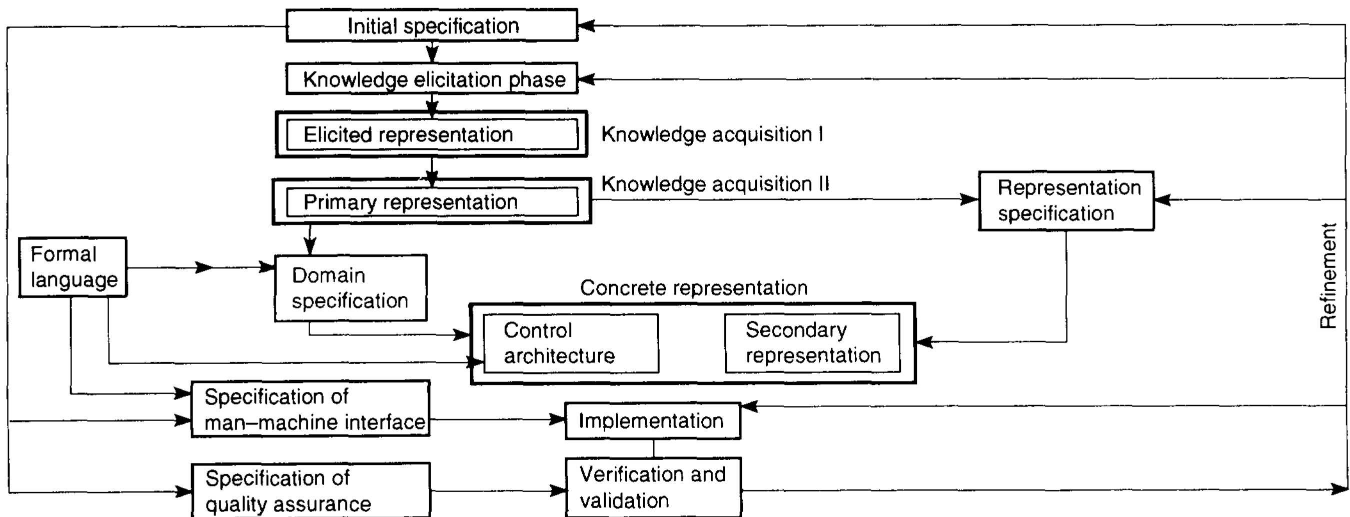

Figure 2. Development methodology for knowledge-based systems

guage can be produced. This is known as the representation specification. Following this, the domain and representation specifications are drawn together to form the secondary representation in which the domain knowledge from the domain specification is then represented in the form advocated by the representation specification. This, plus the specification of the control architecture, forms the concrete specification. This acts as a specification for the implementation of the knowledge base, which when combined with the man—machine interface specification (which allows the human—computer interaction considerations to be understood) provides the basis for implementing the whole system. This methodology has been discussed in greater detail[57].

INFLUENCE OF THESE TECHNIQUES ON QUALITY FACTORS

The techniques described above are each designed to focus on certain aspects of software quality. However, the maximum benefit of the techniques can only be gained when they are used in unison. The most effective means of achieving this is through a methodology such as that described above.

The increased quality of knowledge-based systems through the suggested approach to development can be seen when considering the effect that the use of these techniques would have on Carpenter's quality factors.

The most dramatic effect would be on the correctness of the systems. This would be due to the ability of the knowledge engineer to measure the correctness of the system against its specifications and through the methodology show that the systems development follows from one specification to another. The reliability metrics discussed could also be used to measure expert system reliability and hence correctness. Again the use of multiple independent specifications will enhance the knowledge engineer's ability to locate and correct errors, so raising the system reliability. The structured development process in relation to the specifications will encourage an efficient system to be created, but the independence of the specifications will not detract from, or prevent, the knowledge engineer from inspecting the implementation with regard to efficiency. Further, if a functional approach is used then the program could undergo transformation processing to raise its efficiency while still adhering to the specification.

The integrity of a system can also be improved through the incorporation of integrity constraints into the specification. The ability to reason about a system's integrity without specifications is difficult as the knowledge engineer may not be aware of all the integrity gaps for a nontrivial system.

The reusability of the system will improve as the subfunctions of a system will be defined through the specifications and thus they can be reasoned about effectively without the fear of unforeseen side-effects. The useability of a system is also improved as the prospective user can read the system specifications.

A major quality increase occurs indirectly, that of system maintainability. The degree of effort required to update a rigorously specified and developed system is substantially lower than a traditionally or nonspecified system.

The different approaches to testing have been considered in this paper and when used against a formalized document, such as a specification, against which test results can be compared, then substantial benefits can be gained through use of appropriate tests at appropriate places in development.

The ninth software quality factor, flexibility, is similar to that of maintainability in that the effort needed to modify a specified operational program is minimal, compared to an unspecified program.

The implementational independence of the development philosophy gives it the ability to remain abstracted from portability considerations and facilitates implementation regardless of environment.

The final two software quality factors, interoperability and intraoperability, are also far easier to undertake as the developer can examine the subfunctions and the

effect that coupling them with another function will have, either internally or externally.

CONCLUSION

It was the aim of this paper to show that if specification techniques are used in conjunction with a rigorous development method that uses rigid validation and verification techniques, the quality level of knowledge-based systems can be raised significantly. It has been noted that the area of quality assurance for knowledge-based systems demands alternative or amended strategies from those associated with conventional systems and that not all areas are as theoretically developed as required. Thus the aim of current and future research should be to build the necessary theoretical foundations for formal quality assurance to be carried out and further that the techniques developed be integrated so that they all contribute to the quality of the systems developed.

REFERENCES

1 **Gitlow, H, Gitlow, S, Oppenhiem, A and Oppenhiem, R** *Tools and methods for the improvement of quality* Irwin (1989)

2 **Garvin, D** 'What does 'product quality' really mean?' *Sloan Manage. Rev.* (1984)

3 **Hetzel, W** *The complete guide to software testing* QED Information Sciences (1984)

4 **Carpenter, C L and Murine, G E** 'Measuring software product quality' *Quality Progress* (May 1984)

5 **Boehm, B W, Brown, J R and Lipow, M** 'Quantitative evaluation of software quality' in *Proc. 2nd Int. Conf. Software Engineering* San Francisco, CA, USA (October 1976) pp 592–605

6 **Conte, S D, Dunsmore, H E and Shen, V Y** *Software engineering metrics and models* Benjamin/Cummings (1986)

7 **Dunn, R H** 'Software quality assurance: a management perspective' *Quality Progress* (July 1988)

8 **Schulmeyer, G G** 'Standardization of software quality assurance' in **Schulmeyer, G G and McManus, J I (eds)** *Handbook of software quality assurance* Van Nostrand Reinhold (1987)

9 **Smith, D J and Wood, K B** *Engineering quality software* Elsevier (1989)

10 **Miller, L** 'A realistic industrial strength life cycle model for KBS' in *AAAI Workshop on Knowledge-Based Systems Verification, Validation and Testing: Workshop Notes* (29 July 1990) Boston, MA, USA

11 **Freedman, M** *Smalltalk Newsletter* AIAA: Artificial Intelligence Technical Committee (February 1990)

12 **Boehm, B W** 'An experiment in small scale application software engineering' *IEEE Trans. Soft. Eng.* Vol 7 No 5 (May 1981) pp 482–493

13 **Kopetz, H** *Software reliability* Springer-Verlag (1980)

14 **Littlewood, B** *Software reliability: achievement and assessment* Blackwell (1987)

15 **Musa, J A, Iannino, A and Okumoto, K** *Software reliability: measurement, prediction, application* McGraw-Hill (1987)

16 **Hollnagel, E** *The reliability of expert systems* Halstead (1989)

17 **Plant, R T** 'Reliability of expert systems' in *Proc. TIMS/ORSA National Meeting* Las Vegas, NV, USA (7–9 May 1990)

18 **Juran, J** *Quality control handbook* McGraw-Hill (1979)

19 **McCabe, J T and Schulmeyer, G G** 'The pareto principle applied to software quality assurance' in **Schulmeyer, G G and McManus, J I (eds)** *Handbook of software quality assurance* Van Nostrand Reinhold (1987)

20 **Beardsley, J F and Associates, International Inc.** *Quality circles: member manual* San Jose, CA, USA

21 **Pettijohn, C L** 'Achieving quality in the development process' *AT&T Tech. J.* (March/April 1986)

22 **Grady, R B and Caswell, D L** *Software metrics: establishing a company-wide program* Prentice Hall (1987)

23 **Rushby, J** 'Quality measures and assurance for AI software' *NASA report* 4187 NASA (1989)

24 **Goodenough, J B and Gerhart S L** 'Towards a theory of test data selection' *IEEE Trans. Soft. Eng.* Vol 1 No 2 (June 1975) pp 156–173

25 **Downs, T** 'An approach to the modeling of software testing with some applications' *IEEE Trans. Soft. Eng.* Vol 11 No 4 (April 1985)

26 **Scamboros, E T** 'A scenario-based test-tool for examining expert systems' in *Proc. Int. Conf. Systems, Man, and Cybernetics* IEEE (1986) pp 131–135

27 **O'Keefe, R M, Balci, O and Smith, E P** 'Validating expert system performance' *IEEE Expert* Vol 2 No 4 (Winter 1987) pp 81–90

28 **Bowker, A H** 'A representation of Hotelling's T^2 and Anderson's classification statistic W in terms of simple statistics' in **Olin (ed)** *Contributions to probability and statistics* Stanford University Press (1960)

29 **Gashnig, J, Klahr, P, Pople, H, Shortliffe, E and Terry, A** 'Evaluation of expert systems: issues and case studies' in **Hayes-Roth, F, Waterman, D A and Lenat, D B (eds)** *Building expert systems* (1983)

30 **Currit, A P, Dyer, M and Mills, H D** 'Certifying the reliability of software' *IEEE Trans. Soft. Eng.* Vol 12 No 1 (January 1986) pp 3–11

31 **Poore, J H** 'Derivation of local software quality metrics (software quality circles)' *Soft. Pract. Exper.* Vol 18 No 11 (1988) pp 1017–1027

32 **Neches, R, Swartout, W R and Moore, J D** 'Enhanced maintenance and explanation of expert systems through explicit models and their development' *IEEE Trans. Soft. Eng.* Vol 11 No 11 (November 1985) pp 1337–1351

33 **Stachowitz, R A, Chang, C L, Stock, T S and Coombs, J B** 'Building validation tools for knowledge-based systems' in *Proc. First Annual Workshop on Space Operations, Automation and Robotics (SOAR 87)* (Houston, TX, USA, August 1987) NASA Publication 2491 (1987) pp 209–216

34 **Leveson, N G and Harvey, P R** 'Analyzing software safety' *IEEE Trans. Soft. Eng.* Vol 9 No 5 (May 1983) pp 569–579

35 **Leveson, N G** 'Software safety in computer controlled systems' *Computer* Vol 17 No 2 (February 1984) pp 48–55

36 **Leveson, N G** 'Software safety: why, what and how' *ACM Comput. Surv.* Vol 18 No 2 (June 1986) pp 125–163

37 **Weyuker, E J** 'An evaluation of program-based software test data adequacy criteria' *Commun. ACM* Vol 31 No 6 (June 1988) pp 668–675

38 **Plant, R T** 'The validation, verification and testing of knowledge-based systems' *Heuristics: J. Knowl. Eng.* (Spring 1990)

39 **Boehm, B W** *Software engineering economics* Prentice Hall (1984)

40 **Alavi, M** 'An assessment to the prototyping approach to information systems development' *Commun. ACM* Vol 27 No 6 (June 1984) pp 556–563

41 **Clocksin, W F and Mellish, C S** *Programming in Prolog* Springer-Verlag (1981)

42 **Gordon, M J C** *Edinburgh LCF Vol III* (Lecture Notes in Computer Science No 78) Springer-Verlag (1979)

43 **Turner, D A** 'Miranda: a non-strict functional language with polymorphic types' in *Proc. IFIP Conf. Functional Programming Languages and Computer Architectures* Springer-Verlag (1985)

44 **Turner, D A** 'Recursive equations as a programming lan-

guage' in **Darlington, J, Henderson, P and Turner, D A (eds)** *Functional programming and its applications* Cambridge University Press (1982) pp 1–28

45 **Spivey, J M** *Understanding Z* Cambridge University Press (1990)

46 **Jones, C** *Software development: a rigorous approach* Prentice Hall (1980)

47 **McCarthy, J** 'Towards a mathematical science of computation' in *Information Processing, Proc. IFIP Congress* North-Holland (1962) pp 21–28

48 **Floyd, R W** 'Assigning meanings to programs' in *Proc. Amer. Math. Soc.* Vol 19 (1967) pp 19–32

49 **Hoare, C A R** 'An axiomatic basis for computer programming' *Commun. ACM* Vol 12 (1969) pp 576–583

50 **Kowalski, R A** 'The relation between logic programming and logic' in **Hoare, C A R and Shepherdson, J C (eds)** *Mathematical logic and programming languages* Prentice Hall (1984) pp 11–27

51 **Leibrandt, U and Schnupp, P** 'An evaluation of Prolog as a prototyping system' in **Buddle, R (ed)** *Approaches to prototyping* Springer-Verlag (1983) pp 424–433

52 **Hayes, I J** 'Specification case studies' *Technical monograph PRG-46* Oxford University Computing Laboratory, Oxford, UK (1985)

53 **Plant, R T** 'A case study in expert system development' in *Proc. Second Int. Symp. Methodologies for Intelligent Systems* Charlotte, NC, USA (October 1987)

54 **Jacob, R J K** 'Using formal specifications in the design of a human-computer interface' *Commun. ACM* Vol 26 No 4 (April 1983)

55 **Sufrin, B A and He, J** 'Specification, analysis and refinement of interactive processes' in **Harrison, M and Thimbleby, H (eds)** *Formal methods in human-computer interaction* Cambridge University Press (1990)

56 **Gold, D and Plant, R T** 'Towards the specification of expert systems' *Working paper CIS/90/2* University of Miami, FL, USA (1990)

57 **Plant, R T** 'Utilizing formal specifications in the development of knowledge-based systems' in **Partridge, D (ed)** *Artificial intelligence and software engineering* Ablex Press (1990)

BIBLIOGRAPHY

Darlington, J 'Program transformation' in **Darlington, J, Henderson, P and Turner, D A (eds)** *Functional programming and its applications* (1982)

Howden, W E 'Weak mutation testing and completeness of test sets *IEEE Trans. Soft. Eng.* Vol 8 No 4 (April 1982) pp 371–379

Plant, R T 'A methodology for knowledge acquisition in the development of knowledge-based systems' *Doctoral thesis* University of Liverpool, UK (1987)

Shooman, M L *Software engineering, design, reliability, management* McGraw-Hill (1984)

Spivey, J M *The Z notation: a reference manual* Prentice Hall (1989)

Weitzel, J R 'A system development methodology for knowledge-based systems' *IEEE Trans. Syst. Man Cyber.* Vol 19 No 3 (May/June 1989)