NASA Contractor Report 181821

Use of Metaknowledge in the Verification of Knowledge-Based Systems

(NASA-CR-181821) USE OF METAKNOWLEDGE IN THE VERIFICATION OF KNOWLEDGE-BASED SYSTEMS Final Report (College of William and Mary) 17 p CSCL 09B

N89-25627

Unclas

G3/62 0217225

Larry J. Morell

College of William and Mary
Williamsburg, VA 23185

Grant NGT 47-003-029
April 1989

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

1. Introduction

A *knowledge-based* system (KB system) is a computer program in which knowledge from a narrowly defined application domain is separately encoded and processed by a distinct general problem-solving method. The separately encoded knowledge is called a *knowledge base* and the problem solving method is called an *inference engine.* The inference engine proceeds by making deductions from the knowledge base.

The similarity to deductive systems from formal logic is strong. A deductive logic system consists of axioms along rules of inference by which deductions are made from the axioms. The knowledge base of a KB system may be thought of as a set of axioms while the inference engine may be thought of as the rules of inference. Even the notion of an application domain is given explicit treatment in deductive logic systems under the rubric of interpretation mappings, i.e., mappings of the symbols of the logic to the real-world entities.

This paper models knowledge-based systems as deductive systems for the purpose of clarifying the issues involved in verifying such systems. The model indicates that the two primary areas of concern in verification of KB systems are demonstrating their consistency and completeness. A system is *inconsistent* if it asserts something that is not true of the modeled domain. A system is *incomplete* if it lacks deductive capability. Two forms of consistency are discussed along with appropriate verification methods. Three forms of incompleteness are discussed. The use of *metaknowledge,* knowledge about knowledge, is explored in connection to each form of incompleteness.

It is clear that no one verification technique will demonstrate a system to be both complete and consistent. Specific techniques are applicable for some inconsistencies but not for others. Data flow analysis [1,2], for instance, may be useful for demonstrating the absence of loops in the knowledge base, but not for demonstrating the absence of timing defects. A second observation is that correctness cannot be deduced automatically for an arbitrary knowledge-based system. The emphasis in this paper is to discuss techniques applicable to a broad class of knowledge-based systems. This is the motivation for the abstract model of knowledge-based systems and its analysis given below. The next section defines the terminology used in the paper. Section 3 discusses some of the problems faced in verifying a KB system. The model of KB systems is introduced in section 4. Section 5 explores the notions of consistency and completeness in the light of the model. Section 6 presents summaries and conclusions.

2. Terminology

Following Waterman [3] a *knowledge-based* system (KB system) is a computer program in which knowledge from a narrowly defined application domain is separately encoded and processed by a distinct general problem-solving method. The separately encoded knowledge is called a *knowledge base* and consists of a collection of related *knowledge items.* Each knowledge item represents a unit of knowledge from the application domain. The interrelationships among the knowledge items enable deductions to be drawn from the knowledge base as a whole. The deductions are solutions to problems from the application domain. The implementation of this deductive problem-solving method is called an *inference mechanism* or an *inference engine.* The inference engine thus applies knowledge found in the knowledge base to solve problems in the application domain.

A simple computerized help system illustrates this knowledge-based approach. The application domain in this case involves providing information about commands available on the computer system. The knowledge base consists of a directory of help files, one for each command in the system. Each of these files is a knowledge item. A user who needs information about a particular command interacts with the help processor, the inference engine in this case. The help processor supplies the desired information from the knowledge base. Thus the help system contains all the elements to qualify it as a knowledge-based system. It has a separate encoded knowledge base (the help file directory) for a narrowly defined application area (command help), which is processed by a distinct deductive mechanism (the help processor).

To describe a system as being knowledge-based is to describe its manner of implementation, not the class of problems it is trying to solve. Some problems can be solved both by knowledge-based techniques and conventional programming. For example the help processor could be implemented as a CASE statement to select the information to be displayed by PRINT statements. Violation of any part of the definition makes a system not knowledge-based. Data-driven programming, for instance, produces programs with distinct processing and data components, but the data component might not represent a narrowly defined application domain and the processing component may not be a general problem-solving method, independent of the application domain. An interpreter for a programming language along with a single program from that language cannot be collectively considered as as knowledge-based system. If this were the case, *every* computer executing a program would be a knowledge-based system.

Frequently cited advantages of knowledge-based systems include

(1) more rapid development and early prototypes,

(2) more easily verified systems, and

(3) more easily modified systems.

The help system illustrates each of these. Construction of the inference mechanism can proceed in parallel with the construction of the documentation file knowledge base. A rapid prototype can be constructed using non-uniform documentation files; uniformity can be enforced at a later date. In fact, tools can be developed for ensuring uniformity.

Verification of a knowledge-based system is facilitated by the separation of the knowledge and the processing components. The inference engine may be "off the shelf", not requiring any verification. Use of a uniform representation of knowledge in the knowledge base enables the construction of tools that verify desired properties of the knowledge base. For example, tools can be constructed to ensure that all the files in the help system satisfy a given format.

Separation of a uniformly encoded knowledge base also improves modifiability. Utilities can be written to find knowledge items (encoded knowledge) which satisfy specified patterns. Replacing a knowledge item is easily accomplished. Traces of system execution can be used to explain system results. Contrast this with a conventional system in which the knowledge is intermingled with control and distributed across many lines complicating the identification and replacement of knowledge.

Ricks and Abbott [4] describe an experiment in which flight information is computed both by a knowledge-based program and by a more conventional program that did not use knowledge-based techniques. Their conclusions were

> The results show that rule-based programming techniques have the potential for improving the productivity of the programmer or designer who develops a system. In this study, modification of the rule-based program was easier, more efficient, and less error-prone than the traditional program's. The rule-based program's separate, homogeneous rule base and inference engine could aid in the simplification and test-tool development needed during the verification process. It was also easier to implement an explanation capability in the rule-based program.

A knowledge-based system is not synonymous with an expert system. Though there are several defining characteristics of an expert system, a prominent feature appears to be the possession of expertise in a given domain. The essential point to recognize is that the term "expert system" emphasizes the behavior of a program while the term "knowledge-based system" emphasizes the implementation technique of a system, i.e., the isolation of its knowledge into a knowledge base. What confuses the issue is that many expert systems are implemented as knowledge-based systems. This does not mean, however, that every expert system is a knowledge-based system or every knowledge-based system is an expert system! For example, the help system mentioned above would not be considered an expert system by most, since its only expertise is that of mapping and displaying. However a help system which attempted to understand the contents of its files and create useful links for readers approaches an expert system. Also, of the two expert systems constructed by Ricks and Abbott only one was knowledge-based.

Two other terms must be distinguished: verification and validation. *Verification* is the process of demonstrating that software possesses features specified by its documentation. *Validation* is the process of demonstrating that software possesses features desired by its end-user. Without documentation, verification cannot be done, but validation can be. The standard "waterfall" model of software development emphasizes verification as its primary method for moving toward validation. Requirements lead to specifications, which in turn are translated into designs that are refined until code is produced. Verification can be performed at each of these stages. Most effort in verification has concentrated on showing that the code has desired features. Many techniques have been developed for verifying code as a whole [5] and at the unit level [6].

Validation is more comparable to a field test, i.e., placing the software in an operational environment and observing its behavior. Frequently validation must be conducted in a simulated operational environment because the operation environment is not available. Software for a lunar landing is an example of this; it is clearly impossible to test it in its operational environment before deploying it!

3. Problems faced in verifying a knowledge-based system

Many have commented upon problems faced in verifying and validating knowledge-based systems. The arguments may be summarized by the following questions[7]:

(1) What do you verify?

(2) Against what do you verify?

(3) With what do you verify?

Each of these questions is addressed briefly below.

What do you verify? The two principal components in a knowledge-based system, the knowledge base and the inference engine, are both candidates for verification. Frequently an off-the-shelf inference mechanism is used and thus requires no verification. Custom built inference mechanisms do, since any contained faults may have indeterminable effects on all processing, even if the knowledge base is perfect.

The knowledge base is the crucial component for verification in a knowledge-based system. Aspects of the knowledge to be verified appear later in this paper, but a few are named here to illustrate the complexity of the task. The correctness, accuracy, and timeliness of each individual knowledge item must be verified. The consistency of the knowledge must be shown; separate knowledge items should not be mutually contradictory. The knowledge base should be complete: obvious "holes" should be filled in. Consistency and completeness are discussed in depth later in this paper. What further compounds the problem is that individual knowledge items may be correct or incorrect according to the relationships they bear to other items. These relationships might not be explicitly stated in the knowledge base; some may be induced by the inference engine. It is also necessary that human factor considerations be verified.

Against what do you verify a knowledge-base? Verification presumes a specification. All too often in knowledge-based systems no such specification exists. The argument is made that it is more work to write a specification than it is to write the knowledge base directly. In some cases this argument holds, but in general there are many desirable properties that need to be specified that can be used as incomplete or semi-specifications. This knowledge about the knowledge base is called *metaknowledge* and is a vital necessity for verification. An example would be a listing of reliable sources of medical information. If each knowledge item in a medical database indicates its source, then the exclusive use of reliable sources can be verified. An important aspect of this paper is to argue for the judicious use of metaknowledge as an aid to verification.

What tools and techniques can be used to verify features of the knowledge base? Traditional software verification techniques do not appear immediately applicable to verification of knowledge-based systems. One reason is that most verification techniques require a full specification of a problem. Furthermore, many verification techniques are aimed at executable code as opposed to static data bases. Certain techniques such as program coverage, data flow analysis, and safety analysis appear to have their counterpart in knowledge-based systems, but will require adaptation. Many tools developed so far that are applicable to knowledge bases come from the expert system community and tend to be oriented toward particular languages outside mainstream Algol-like languages.

Many other questions arise related to the nature of verifying knowledge bases. Additional information can be found in [7] and 8].

4. A Model of Knowledge-based Systems

This section describes a model of knowledge-based systems which facilitates discussion of features which impact verification.

A knowledge-based system can be modeled as a symbolic deduction system in which a problem from a particular problem space is encoded and manipulated according to a knowledge base in order to produce a solution (see Figure 1). In this figure H represents the encoding of the problem, R represents the derived solution and the turnstyle represents the deductive process applied by the inference engine operating on the knowledge base. The solid arrow indicates the method of solution that would be used if the operation were performed manually. The dotted line represents the interpretation mapping, I, and its inverse relation, I^{-1}. This mapping (and its inverse) assigns meaning to the represented problem, the knowledge base and the solution. Verification presumes this mapping because any assertion of inadequacy must be grounded in the world in which the actual problem is to be solved. Put another way, a computer solves a problem only if it accurately encodes the problem and produces an acceptable solution.

A knowledge-based system can be deficient in three ways: it can be inconsistent, incomplete, or both. Brief definitions of these terms are given here and expanded upon later. Inconsistency is a characteristic of the interpretation I. If in applying I at any point the system is asserting something patently false about the domain it models, the system is inconsistent. If, on the other hand, the system lacks deductive capability it should have, it is incomplete. Weakness in either area can yield an incorrect program. Verification therefore focuses on demonstrating that various forms of inconsistency and incompleteness have not occurred.

As a simple example, consider a knowledge base containing assertions about various people. The knowledge base on football players may include statements such as:

(Joe (sex male))
(Joe (pregnant true))

The interpretation mapping maps these symbolic expressions to assertions about the application domain. If I(Joe) = "Josephine Doolittle", a woman, then the knowledge base is inconsistent with the application domain. If I(Joe) = "Joe Montana", a man, then the knowledge base is inconsistent for an obviously different reason. Note however that neither of these inconsistencies can be determined simply by looking at the knowledge base; they each require mapping the knowledge base to some domain in which the resulting statements are contradictory. One situation in which the knowledge base is considered incomplete would be if the the purpose of the program is to determine great football players, but most of Joe's statistics are missing or are represented in such a way that the inference engine cannot deduce they all refer to the same person.

Note that under these definitions of consistency and completeness neither implies the other. In a knowledge-based system the knowledge is separated from the inference mechanism, and each can be independently wrong. It is most likely, however, that inconsistency or incompleteness will imply

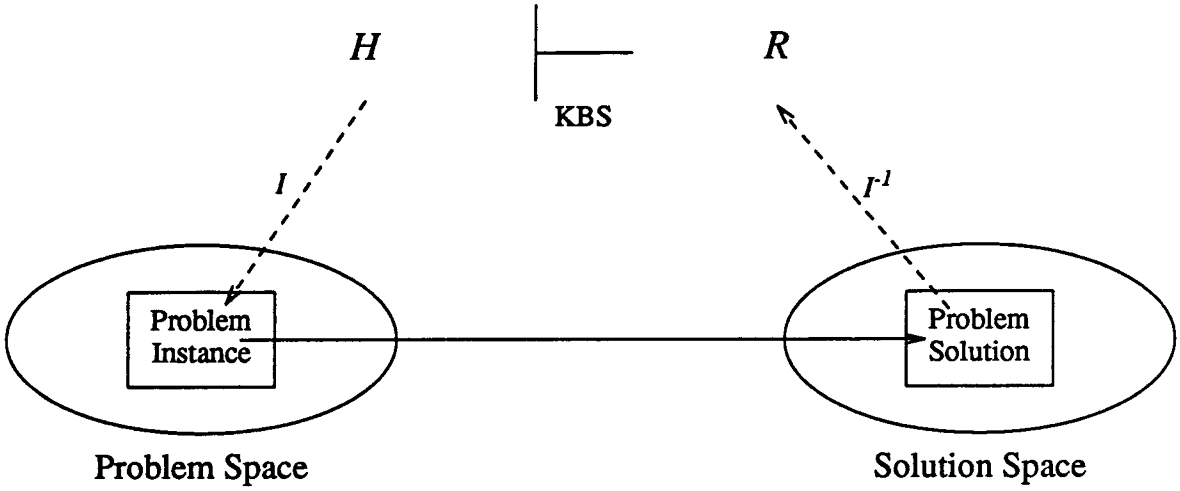

Figure 1

incorrectness. It is therefore important to understand ways in which a system can be inconsistent or incomplete, and determine methods for demonstrating that such is not the case for a given system.

One immediate observation is that no one verification technique will demonstrate a system to be both complete and consistent. Specific techniques are applicable for some inconsistencies but not for others. Data flow analysis [1,2], for instance, may be useful for demonstrating the absence of loops in the knowledge base, but not for demonstrating the absence of timing defects. A second observation is that correctness cannot be deduced automatically for an arbitrary knowledge-based system. The emphasis in this paper is to discuss techniques applicable to a broad class of knowledge-based systems. This is the motivation for the highly abstract model of knowledge-based systems given here.

5. Verification Techniques

The model developed in the preceding section is now used as a basis for exploring the notions of consistency and completeness of a knowledge-based system. Two forms of consistency are discussed along with appropriate verification methods. Three forms of incompleteness are discussed. The use of *metaknowledge,* knowledge about knowledge, is explored in connection to each form of incompleteness.

5.1. Consistency

A knowledge-based system is *inconsistent* if and only if applying the interpretation function *I* to a state of the knowledge base produces a state that is inconsistent with the modeled world. This definition does not restrict when such a mapping occurs - it merely asserts that, when applied, a consistent state must be produced. Since consistency is thus a property of both the initial knowledge base and the intermediate states accessed during the execution of the knowledge-based system, two types of consistency may be identified. A knowledge-based system is said to be *statically consistent* if its initial knowledge base state is consistent with the modeled world (as determined by mapping *I*). A knowledge-based system is *dynamically consistent* if any intermediate state of its knowledge base is consistent with the modeled world. Thus, static consistency deals with what the knowledge base directly asserts, and dynamic consistency deals with what the knowledge base potentially asserts. Each of these forms of consistency are now discussed in depth.

5.1.1. Static Inconsistency

One approach to verifying static consistency is to

(1) collect a number of assertions about the modeled world

(2) using *I*, map the knowledge base to a state description about the modeled world, and

(3) verify the state produced in (2) satisfies the assertions collected in (1).

Such a scheme succeeds if the set of assertions completely characterizes the modeled world, for then the knowledge base contains no inconsistencies with the modeled world. Provided the correctness of the inference engine has been proved, the knowledge-based system is dynamically consistent as well.

If such a complete characterization is available and if a suitable inference engine can be found, the characterization could be used as the knowledge base itself, eliminating the need for building the knowledge-based system in the first place! Of course a separate system may be necessary for improved efficiency. In such cases it might be possible to compile the characterization. This applies when the system is specified in an executable specification language.

In lieu of a complete characterization, a specification of an incomplete set of properties is all that is possible. Given such a set of properties, S, the goal is then to ensure that the knowledge base satisfies at least these properties. The approach mentioned above involves verifying that the knowledge base under *I* satisfies S. There are several difficulties with this approach.

(1) It requires developing a second description of the domain.

(2) It is difficult to automate since this would entail implementing both *I* and a procedure for checking to see if S(*I*(KB)) = True.

An alternate approach is to map *S* into *S'* by I^{-1}, so that I^{-1}(S) is a set of assertions about the knowledge base, i.e., assertions the knowledge base must satisfy. This method has served well in practice, but the application of I^{-1} to P may not be straightforward. To map domain assertions to knowledge base assertions requires the ability to represent arbitrary assertions about the application domain. The formalism for such metaknowledge may not be readily representable in the knowledge-based system.

This second means of checking static consistency may be characterized as using type metaknowledge. Information about the format of the knowledge base, what constitutes incompatible information, ranges of legal values, etc. can be used to detect whether violations have occurred in the knowledge base. This is called type metaknowledge since it closely corresponds to the data type information used by compilers to determine inconsistent use of variables.

Extensive use of type metaknowledge is used in many expert system shells and support tools. Early work in this area included the knowledge base enhancement system TEIRESIAS [9] and the knowledge base debugging system used in ONCOCIN [10]. More recent work includes CHECK [11] and EVA [12], both of which are oriented toward verification of rule-based systems. EVA is briefly described here since it encompasses most of the functionality of CHECK.

EVA [12] checks three aspects of a rule base:

- structural consistency
- logical consistency
- semantic consistency

A rule base is structurally consistent if every rule is usable. Redundant rules, rules involved in cycles, and rules whose left-hand side cannot be satisfied are examples of useless rules. A rule base is logically consistent if no left-hand side implies both A and ˜A, and no rule contains a redundant clause. Lastly, a rule base is semantically consistent if no user-defined qualifications are violated. These qualifications function in the same role as a typing mechanism in a strongly typed language, ensuring that variables and constants are used correctly in a given context. Examples include specifying the bounds and types of variables and constants, and indicating their proper usage in defined relations.

5.1.2. Dynamic Inconsistency

Dynamic inconsistency arises when a knowledge-based system has the potential of producing a state that is inconsistent under the interpretation mapping *I*. Dynamic inconsistency therefore encompasses the semantics of how the inference engine maps one state into another state. Using this information it is possible to determine the relationships among various knowledge items. Expressing these relationships enables two important analyses to be performed on the knowledge-based system. *Safety analysis* verifies that a system does not violate prescribed safety conditions. It may be possible to tolerate an occasional failure of a program, but not if that failure is catastrophic. *Sensitivity analysis* determines the system response to slight modifications in the knowledge base or the input. Extreme sensitivity does not necessarily imply incorrectness, but does indicate areas where additional verification techniques should be directed.

Several standard verification techniques appear adaptable to safety and sensitivity analysis. Possibilities include mutation analysis, symbolic execution, proof-of-correctness, data flow analysis, and symbolic testing. The adaptation of these techniques to safety and sensitivity analysis of knowledge-based systems is discussed below.

5.1.2.1. Safety Analysis

Safety analysis [13, 14] begins with assertions describing safe behavior of a system. Deductions are then made as to the degree to which this behavior is attained.

We first introduce some notation, and then proceed to discuss methods that are potentially applicable to safety analysis of knowledge-based systems. It is necessary to distinguish between a program, its behavior and the function it computes.

Definition If P is a program, then [P] denotes the *program function* computed by P defined as

$$[P] = \{(x,y) \mid \text{Program P on input x outputs y}\}$$

and <P> denotes the *behavior function* computed by P defined as

$$<P> = \{(x,y) \mid \text{Program P on input } x \text{ halts, having behaved in a manner described by } y\}$$

The definition of <P> is imprecise in that it does not specify the particular behavior of interest; the intent is to capture those aspects of the program's execution not inferable from its output. The definition of <P> does not prevent <P>(x) from containing [P](x); in fact, unless otherwise specified it will be assumed that [P] can be deduced from <P>, i.e., <P>(x) will always contain enough information from which to deduce [P](x). Obvious additional candidates for inclusion in <P>(x) are the program's execution time and space consumption, since these two characteristics are sometimes vitally important to safety. The exact information to be included in the behavior specification will always be application dependent since only at that level can it be decided what information is needed to determine, for instance, if the program is safe. For example, a program that computes the time at which the landing gear should be lowered on an aircraft, but completes the computation only after the plane is on the ground, has unacceptable behavior. In contexts of national security, something as obscure as the radio frequencies emitted by the computer while the program executes might be considered part of the program's behavior. The amount of information included in the behavior will vary according to the needs of the project.

Safety and correctness are related concepts and can now be defined.

Definition

A program P is *safe with* respect to an assertion pair <A , B > if and only if for all x that satisfies A , <P>[x] satisfies B . Otherwise it is said to be *not safe* (or *unsafe*) with respect to <A , B >. <A , B > is called a *safety specification.*

Definition

A program P is *correct with* respect to an assertion pair [A , B] if and only if for all x that satisfies A , [P][x] satisfies B . Otherwise it is said to be *not correct* (or incorrect) with respect to [A , B]. [A , B] is called a *a correctness specification.*

Both of these specifications must also be unambiguous and decidable.

Safety and correctness differ in their intent and scope. Safety has a broader scope since it considers the entire behavior of the program, while correctness considers only input-output pairs. The intent of safety specifications is more narrow than correctness specifications, though. Safety focuses on the impact a program may have on its environment; correctness provides no such focus.

Several safety specifications can apply to a program simultaneously. *Safety analysis* then is the process of determining which safety specifications are satisfied and to what extent. The most straightforward method of doing this is to capture the behavior of the system for all inputs which satisfy the input assertion, and to compare this behavior to that specified by the output assertion. This "black box" analysis is frequently impossible, because the input space defined by the input assertion is too large. It is thus necessary to analyze classes of computations at one time. In traditional programming a data flow graph is constructed to aid in this processing [1]. For knowledge-based programs a complementary graph is necessary. The construction of such a graph is described below.

Recall that a knowledge base is a collection of knowledge items from which the inference engine performs its deductions. The deduction process proceeds as follows: one state of the knowledge base yields the next, which yields the next, and so on until the inference mechanism halts. During any of these transitions the inference engine determines the next state from a subset of the current knowledge items. The next state is a modification or enhancement of the previous set of knowledge items. For the purposes of discussion here it will be assumed that all existing or potential knowledge items are known. Each transition connects a set of knowledge items referenced in the current state with the set of knowledge items modified or produced in the succeeding state. This may be represented by a graph, in which the nodes denote knowledge items and the arcs represent connections induced by potential transitions. Such a graph is called here a *computation flow graph,* an analogue of data flow graphs associated with conventional programs (see [1]). Knowledge items which may be present when the inference engine begins execution are specially marked as initial nodes. Knowledge items which may be present when the inference engine halts are specially marked as terminal nodes.

An example of a computation flow graph is shown in figure 2. Here the inference engine uses x_1, x_2, and x_3 during some transition to produce or modify knowledge item y . It may also be the case that x_2 is related to some other knowledge item z , and hence there would be an arc connecting x_2 to z in the complete graph.

Safety analysis can be performed on a computation flow graph. This analysis may proceed in two directions: forward or backward. In the backwards mode, sets of terminal knowledge items which do not satisfy the safety assertion are successively traced to initial knowledge items that generate them. Each subset of initial items so identified which satisfies the input assertion yields a violation of a safety assumption. The simplest instance of this process occurs when the output assertion involves only single terminal nodes. In this case no back tracing is necessary since the presence of a terminal node that does not satisfy the output assertion is indicative of a violation. An example would be a control system in

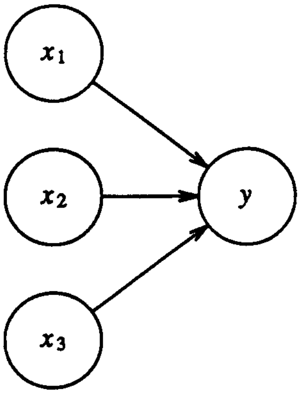

Figure 2

which a particular output, X, would be considered dangerous. This safety specification might be represented by (TRUE, SETTING ≠ X); i.e., for all inputs the output setting is never X. Inspecting the computation flow for the presence of X as a terminal item requires no backward tracing.

In some cases, to determine if a violation could happen it is necessary to propagate the output safety assertion backwards through the system, deducing the set of states which are safe. A *state* in this sense is a collection of knowledge items which could simultaneously exist during execution of the system. Hence each state *S*that satisfies the output assertion determines a set of predecessor states, namely that set of states from which *S*is a logical successor. Each of these predecessor states determines a set of predecessor states and so on until the set of initial states are determined which will ultimately lead to satisfaction of the output assertion. If this set includes all states which satisfy the input assertion then the safety specification is satisfied.

Safety analysis can use forward propagation as well. Given states that satisfy the input assertion, forward propagation successively generates states in the same manner as the original graph is defined. If all terminal states satisfy the output safety assertion, then the safety specification is satisfied.

A knowledge-based system of sufficient complexity could induce a computation flow graph that is unmanageably large. One way of handling such situations is to produce an abstract graph. Several abstractions of computation flow graphs can be imagined. One abstraction which has already been used is that of treating the graph as a collection of states, rather than of individual knowledge items. Another abstraction results from overlaying nodes which share characteristics considered important. The resulting nodes represent collections of knowledge items that may be processed uniformly by the inference engine.

An alternative to abstracting the graph is to explore multiple paths through the graph simultaneously. A symbolic execution system [15, 16] does precisely this. In symbolic execution of a conventional program, a path is selected through the program, and the computation along that path is determined by executing the path with a symbolic input. The output produced is then expressed in terms of this symbolic input. The parallel in knowledge-based systems is to use symbolic input to represent a collection of inputs satisfying an input assertion. The system is then executed, traversing all paths in the computation flow graph determined by inputs represented by the symbolic input.

If the backward or forward propagation must stop at an earlier point (perhaps due to insufficient information or combinatoric explosion) at least a set of safe intermediate states has been identified. Such information can then be supplied to the inference engine as metaknowledge to help it decide whether it is in a safe state. Deductions from potentially unsafe states may be viewed more suspiciously than those derived from safe states.

5.1.2.2. Sensitivity Analysis

Sensitivity analysis determines the system response to slight modifications in the knowledge base or the input. Sensitivity analysis is particularly appropriate to knowledge-based system because the knowledge base sometimes contains artificially precise rules or values. It is not uncommon, for instance, for an expert system to include estimates of the reliability of certain facts as metaknowledge to the inference engine. The impact of such estimates on the operation of the system is an important phenomenon to investigate. If changing an estimate by 1% radically affects the functioning of the program, and the estimate is only considered accurate to 10%, further investigation is warranted. Similarly, demonstrating that an estimate can be changed considerably without impacting the program may imply that as long as the estimate is within the ball park, it can be trusted.

Mutation testing [17, 18, 19], symbolic execution [15, 16], and symbolic testing [20, 21] appear applicable to sensitivity analysis.

Mutation testing is a technique of judging the quality of test data. Test data distinguishes one version of a knowledge base from another version by demonstrating that the output of the two systems differ on the test data. A mutation operator applied to the knowledge base produces a slightly different knowledge base, called a mutant. Test data is considered adequate if it distinguishes all non-equivalent simple mutants from the original program. A simple mutant is a slight variant on the original program, e.g. changing 1.890 to 1.892. Mutation testing thus explores sensitivity boundaries by requiring test data to distinguish slight changes. The knowledge base is mutated slightly and test data is developed that distinguishes this mutant from the original knowledge base, if possible. This process is repeated for every possible application of a mutation operation. The resulting test data then reflects the sensitivity boundaries of the knowledge base.

Symbolic execution can be used in a way to perform sensitivity analysis in a manner called *symbolic testing* [20, 21]. In symbolic testing, a knowledge item in the knowledge base is replaced by a symbolic term. The system is then symbolicly executed, as described earlier. In this case, however, the symbolic output is expressed in terms of both the symbolic input and the symbolic term. This process therefore captures the impact of the original knowledge item. Sensitivity can therefore be deduced directly from this expression.

5.2. Completeness

The preceding section discusses ways of analyzing a knowledge-based system for inconsistency. The model discussed in section 3 indicates an additional way in which a knowledge-based system can be insufficient: it can be incomplete. Whereas consistency deals with the degree to which the knowledge base faithfully represents the application domain, completeness addresses the expressibility of the system and the limits of its deductive mechanism. Consistency assesses what *is*; completeness addresses what *should be*. To see the difference, consider the following example. A knowledge base with one knowledge item may be fully consistent, in that the information encoded in that item accurately portrays a true assertion in the modeled domain. It may even be that all deductions from this single item are consistent. But it is likely that the system is woefully incomplete, since there could be much relevant knowledge from the application domain which is not represented and therefore unavailable to enter into the deductive process.

Definition

A deductive system D is *complete* for a problem space P and an interpretation function I if and only if for every $\mathrm{P} \in P$

P is expressible in D as some H under I^{-1},

$H \rightarrow R$, and

R maps under I to S, a solution of P.

where a problem domain is the set of problems to be solved.

Incompleteness can arise then from several sources:

- inadequate expressiveness of the model
- inadequate knowledge base
- inadequate deductive power

Each of these are discussed below.

5.2.1. Inadequate expressiveness of the model

A knowledge-based system provides one or more ways of encoding or expressing knowledge. The encoded knowledge is, of course, a knowledge item or a collection of knowledge items. Popular encoding methods include rules, frames, and semantic nets. The expressibility of a knowledge-based system is the degree to which arbitrary units of knowledge from the application domain can be expressed via the facilities provided by the system. The greater the expressibility, the greater the flexibility of the system. Clearly, if a problem cannot be presented to the system, it cannot be solved by the system. Equally clear is that the system must be able to express the solution. Deficiency in either regard indicates an incomplete system.

What is addressed here is the ability to represent the problem to be solved using the mechanisms supplied by the model. Adequate expressiveness in this regard is sometimes assumed *a priori*, but this is not necessarily appropriate. To verify adequate expressibility would require a formal description of the problem domain. This can be very complex in some cases, e.g., how does one characterize all the potential faults in an aircraft engine for use in a diagnostic system?

The knowledge base verification system EVA incorporates a primitive form of specification that enables the description of constraints on legal combinations of values. Much work remains to be done in this area. A potential source of applicable methods may come from the area of formal semantics[22], since it has expressibility as a primary concern.

Another related issue is how "user-friendly" is the manner of expression for stating the problem. Though full expressibility may be present, the syntax or order may be so convoluted that it is very difficult to use. It has been suggested that this issue be checked as a separate phase in verification [8] so as not to entangle the assessment of the capability of the system with its usability.

5.2.2. Inadequate knowledge base

What kinds of metaknowledge enable the detection of an inadequate knowledge base? The answer to this question reveals the strengths of a knowledge-based approach to programming. Since the knowledge items must be represented with some degree of uniformity, tools can be developed to search, manipulate, and generalize the information found therein. Furthermore, the metaknowledge must equally be represented by some uniform mechanism, and the metaknowledge about the metaknowledge, *ad infinitum.*

The following list contains categories of metaknowledge that could prove useful in verifying knowledge-based systems. The list is not intended to be exhaustive; rather it is representative of the kind of information that needs to be collected in order to maintain intellectual control over the development of a knowledge-based system.

Accuracy
How precise is the knowledge?

Applicability
What are the limits of its applicability? What are the conditions under which this knowledge item could prove useful? What aspect of the problem or system does it address?

Assessment
How do you assess the value of this knowledge? Can run-time statistics aid in certifying its usefulness? For example, how frequently has the knowledge item been employed in producing a correct solution?

Consistency
What knowledge would prove inconsistent with this knowledge?

Completeness
What additional knowledge is necessary to complement this knowledge?

Disambiguation
When two pieces of knowledge are both applicable, how should one choose between them?

Justification
Why is this knowledge believed important enough to include in the knowledge base?

Life Span
How long should this knowledge remain in the system? Truth maintenance in a dynamic environment can be very complicated.

Purpose
What circumstances (goals) motivated the inclusion of this knowledge?

Reliability
What is the probability that this knowledge will be correct for a given situation?

Source
What is the source of the knowledge (expert, book, experiment, etc.)?

It should be emphasized that these metaknowledge categories may be applicable to more than one type of knowledge. It is possible to have metaknowledge about all aspects of the system, including, but not limited to, the representation method, the system tools, the application domain, and the system execution history. For example it is perfectly meaningful to discuss the reliability of a fact, a rule, a heuristic, and even a reliability metafact. Likewise completeness may refer to the logical completeness of a rule (according the format of legal rules) or the completeness of having covered all the values from the application domain for some particular assertion. Clearly if metaknowledge is used in the system to improve the performance of the system, it is then possible to have meta-metaknowledge, and so on. Thus, these categories of metaknowledge span the spectrum from the minutia of the knowledge items to the overall goals of the system.

Metaknowledge has been used to some degree in MYCIN and ONCOCIN, but not to the degree suggested above. Completeness of knowledge bases will be much improved if the metaknowledge categories suggested above are incorporated routinely into every knowledge-based system.

5.2.3. Inadequate inference mechanism

Another way the system could be incomplete is through limited inference mechanism. Though this is unlikely in a standard system, it could easily occur in a custom-made inference engine. An example would be a too-rigid disambiguation method of conflict resolution in a rule-based system. The following kinds of metaknowledge might prove useful in these circumstances:

Knowledge about when a given rule or knowledge item is enabled; i.e. context dependencies that make certain knowledge items applicable. An example would be a control program which must behave differently in different phases of operation. Certain knowledge items might become active, while others become subdued.

Knowledge about goal ordering and scheduling.

Knowledge about the representation used, for debugging and explanation.

Knowledge about how long certain computations should take.

Causal knowledge about the system which can be used to judge the adequacy of the deductions made in the system. CASNET [23] had such a "first principles" model available to it to judge its own behavior.

6. Summary and conclusions

A knowledge-based system has been modeled as a deductive system. The model indicates that the two primary areas of concern in verification are demonstrating consistency and completeness. A system is *inconsistent* if it asserts something that is not true of the modeled domain. A system is *incomplete* if it lacks deductive capability. Two forms of consistency were discussed, static and dynamic. Particular emphasis was placed on safety and sensitivity analysis. Three forms of incompleteness were discussed. The use of *metaknowledge,* knowledge about knowledge, was explored in connection to each form of incompleteness.

The following is suggested by earlier discussions:

(1) It is imperative that metaknowledge be explicitly incorporated into knowledge-based systems. Research needs to be done to determine what categories of metaknowledge are most useful, and how those categories are best represented.

(2) Conventional verification techniques appear adaptable to knowledge-based systems. To establish this it will be necessary to apply some of the ideas presented here to a "real world" knowledge-based system. Safety and sensitivity analysis techniques described here appear to be appropriate candidates for such an experiment.

References

[1] L. D. Fosdick and L. J. Osterweil, Data Flow Analysis in Software Reliability, *ACM Computing Surveys 8,* 3, pp. 305-330 (Sept. 1976).

[2] J. W. Laski and B. Korel, A Data Flow Oriented Program Testing Strategy, *IEEE Transactions on Software Engineering SE-9,* 3, pp. 347-354 (May 1983).

[3] D. A. Waterman, *A Guide to Expert Systems,* Addison-Wesley Publishing Company(1986).

[4] W. R. Ricks and K. H. Abbott, Traditional Versus Rule-Based Programming Techniques: Applications to the Control of Optional Flight Information, NASA Technical Memorandum 89161, NASA Langley Research Center, Hampton, VA.(July 1987).

[5] J. S. Collofello, Introduction to Software Verification and Validation, Curriculum Module SEI-CM-13-1.0, Software Engineering Institute, Carnegie Mellon University(October 1987).

[6] L. J. Morell, Unit Testing and Analysis, Curriculum Module SEI-CM-9-1.1, Software

Engineering Institute, Carnegie Mellon University (December 1988).

[7] R. M. OKeefe, O. Balci, and E. P. Smith, Validating Expert System Performance, *IEEE Expert 2,* 4, pp. 81-90 (Fall 1987).

[8] F. Hayes_Roth, D. A. Waterman, and D. B. Lenat, eds., *Building Expert Systems,* Addison-Wesley (1983).

[9] R. Davis, Interactive Transfer of Expertise, in *Rule-Based Expert Systems,* ed. E. H. Shortliffe, eds.,Addison-Wesley(1984).

[10] M. Suwa, A. C. Scott, and E. H. Shortliffe, Completeness and Consistency in a Rule-Based System, in *Rule-Based Expert Systems,* ed. E. H. Shortliffe, eds.,Addison-Wesley(1984).

[11] T. A. Nguyen, W. A. Perkins, T. J. Laffey, and D. Pecora, Knowledge Base Verification, *AI Magazine 2,* 4, pp. 65-79 (Summer 1987).

[12] R. A. Stachowitz and J. B. Combs, Validation of Expert Systems, *Proceedings, Hawaii International Conference on System Sciences,* (January 1987).

[13] N. G. Leveson and P. R. Harvey, Analyzing Software Safety, *IEEE Transactions on Software Engineering SE-9,* 5, pp. 569-579 (September 1983).

[14] N. G. Leveson and J. L. Stolzy, Safety Analysis Using Petri Nets, *IEEE Transactions on Software Engineering SE-13,* 3, (March 1987).

[15] L. Clarke, A System to Generate Test Data and Symbolically Execute Programs, *IEEE Transactions on Software Engineering SE-2,* p. 215 222 (Sept. 1977).

[16] W. E. Howden, Symbolic Testing and the DISSECT Symbolic Evaluation System , *IEEE Transactions on Software Engineering SE-3,* pp. 266-278 (1977).

[17] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, Theoretical and Empirical Studies on Using Program Mutation to test the Function Correctness of Programs, *Principles of Programming Languages,* pp. 220-233 (1980).

[18] R. DeMillo, R. J. Lipton, and F. G. Sawyer, Hints on Test Data Selection: Help for the Practicing Programmer, *Computer 11,* pp. 34-41 (April 1978).

[19] W. E. Howden, Weak Mutation Testing and Completeness of Test Sets, *IEEE Transactions on Software Engineering SE-8,* pp. 371-379 (July 1982).

[20] L. J. Morell, A Theory of Error-Based Testing, University of Maryland TR-1395, Department of Computer Science(August, 1984). PhD Thesis.

[21] L. J. Morell, A Model for Code-based Testing Schemes, *Fifth Annual Pacific Northwest Software Quality Conference,* pp. 309-326 (October 1987).

[22] F. G. Pagan, *Program Flow Analysis, Theory and Applications,* Prentice-Hall, Inc., Englewood Cliffs, N.J.(1981).

[23] S. M. Weiss and C. A. Kulikowski, *A Practical Guide to Designing Expert Systems*, Rowman & Allanheld Publishers, Totawa, New Jersey(1984).

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.
NASA CR-181821

2. Government Accession No.

3. Recipient's Catalog No.

4. Title and Subtitle
Use of Metaknowledge in the Verification of Knowledge-Based Systems

5. Report Date
April 1989

6. Performing Organization Code

7. Author(s)
Larry J. Morell

8. Performing Organization Report No.

10. Work Unit No.
505-66-21-01

9. Performing Organization Name and Address
College of William and Mary
Department of Computer Science
Williamsburg, VA 23185

11. Contract or Grant No.
NGT 47-003-029

13. Type of Report and Period Covered
Contractor Report

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes
Langley Technical Monitor: Sally C. Johnson

Final report

16. Abstract

Knowledge-based systems are modeled as deductive systems. The model indicates that the two primary areas of concern in verification are demonstrating consistency and completeness. A system is inconsistent if it asserts something that is not true of the modeled domain. A system is incomplete if it lacks deductive capability. Two forms of consistency are discussed along with appropriate verification methods. Three forms of incompleteness are discussed. The use of metaknowledge, knowledge about knowledge, is explored in connection to each form of incompleteness.

17. Key Words (Suggested by Author(s))
Knowledge-Based Systems
Expert Systems
Software Validation
Metaknowledge

18. Distribution Statement
Unclassified - Unlimited

Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	16	A03