Mid-America Conference on Intelligent Systems
October 27 & 28, 1994 Overland Park, Kansas

PRACTICAL ISSUES OF AI

John M. Switlik
Commercial Airplane Support Group - Wichita
Boeing Computer Services
Wichita, KS 67207

ABSTRACT

This paper covers several issues that are of practical importance to computer application projects, and it looks at systems with an artificial intelligence (AI) heritage, namely knowledge based systems (KBS) and knowledge based engineering (KBE). The intent of the paper is to discuss the problems that are inherent to advanced computing systems through briefly reviewing related topics; these topics are still active research areas, however practical computing has to successfully address the issues. The most prevalent means for doing this revolves around applying human expertise and providing a sound foundation for development. The paper will look at the changes in the software industry brought on by computing progression into complex systems, will briefly review issues of computability and complexity, and then will look at the necessity for verification, validation, and test (VVT) of KBS/KBE.

1 OVERVIEW: BOXES - BLACK AND WHITE

Let's look at a box view of computer systems. Boxes can be black or white (or opaque or clear). As well, one can think of various levels of gray boxes. For the purposes of this paper, consider that these boxes are similar except for the type of justification that may be connected with them.

Take the black box, please. Generally, there is some intended behavior that the developer of the box wants to achieve, there is a set of knobs for controlling the behavior, and there will be input/output and state descriptions related to the box. There is an expected behavior that the user of the box will have which might be described in terms of specific functionality, knob setting, or state transitions. The user will establish a judgement about whether the black box meets the expectation through various means usually based upon usage. Successful judgements can be associated with justification. Continued successful use, for example, will raise the user's confidence in, and knowledge of, the box. The confidence with which the user views the box's creator and verifier will be a factor, as well. In regard to having an intent, a behavior, and a justification, the black box shares a commonality with the white box.

However, the white box (or varying lighter gray boxes) has one additional characteristic. With a white box, the user may, if he or she so wishes, look into the box and its logical structure as a bases for the justification and do a more thorough review of the box, whereas with the black box, this is not an option. That is one method of viewing the blackness (hidden information). A white box can be unlocked, from the user's view, so that its contents can be made visible.

The black box has to be evaluated from its external view, or behavior, only [8]. One can attempt to deduce the internal view from this external view, and, with luck, get close to an accurate understanding of the workings. Without a look at the contents, though, there is no means to check this inference. One might argue that a black box allows only the extensional view of its particular domain, while the white box supports the intensional view, as well. In this regard, the white box will allow non-enumerative, or analytical, reasoning about a domain, while, unless there is a good inference as to the black box's intensional aspects, such a non-enumerative look would be hit-and-miss, at best.

Figure 1 shows three boxes. The idea is that the black box supports only external or behavioral views. The gray box allows some analysis, but there may be missing or hidden information. The white box is fully open but not necessarily simple or complete. There is another set of issues concerned with evolutionary methods, e.g. neural nets. This paper only notes that the black box idea applies to these as well as it does to the traditional methods.

These definitions are not meant to be necessarily complete but are intended to illustrate a problem. For the sake of the paper's content, just assume that the reader doesn't have the key that unlocks the black box, whereas, for the white box, all keys are readily available and useable.

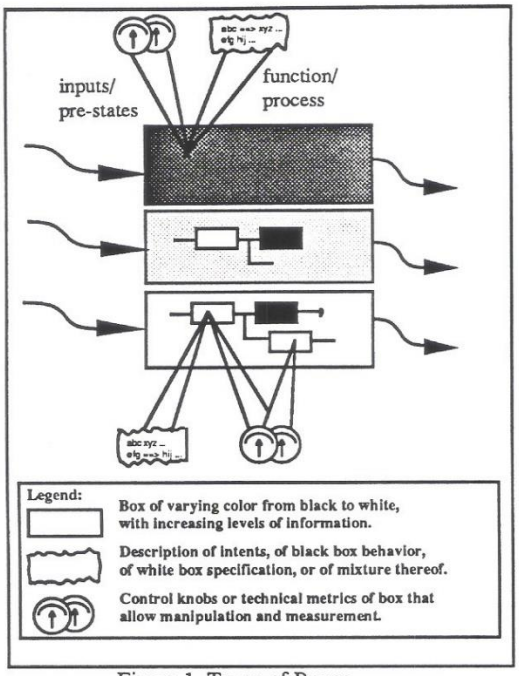

Figure 1. Types of Boxes

But, perfect visibility still raises issues. Though the user can obtain access to a white box, this does not mean that the bases for the justification of the box can be easily (or even necessarily) verified. That is, the major weakness of a computer-based endeavor resolves around this issue: *the ability to verify a white box's contents will be directly related to the analytical basis of the particular domain and to the ability of this basis to be computed.* As computing progresses, we face the increasingly complex tasks of verification and validation. Support will partially come from advances in the mathematical and logical support for computer science [5][6][15], will partially come as different disciplines revolutionize their domains [10], and from applied human expertise [2]. This paper will address these problems and suggest one workable approach.

2 SOFTWARE TRENDS

The development (and use) of computer system software has seen, over the years, a transition from the very low-level black to the high-level black box being collected into white boxes. One example of the former is the machine focus of early programming, some of which included setting hard instructions and parameters by wire. Other examples of the former would also be assembly-level coding and many types of procedural programming. Now, even with the nature of these low-level boxes, the problems of providing reliable computing systems was not a given; that is, issues of computability have been there from the beginning as the experiences of the industry indicate.

An example of the high-level black box might be a machine learning system (e.g. neural network) or a system of agents [7]. However, other examples of the latter are the domain tools (such as ICAD - Intelligent CAD), 4GL environments, spreadsheets, DBMS, simulation languages (such as AUTOMOD), and the general AI tool (such as ADS), and many other examples. A growing set of black boxes that provide white box functionality, also, are the CASE tools [7]. This paper does not address these types of systems in particular, however strong arguments can be made for the importance of this class of systems for the future of computing [10].

This white/black box issue is also evident in the emphasis of certain methodologies on information hiding as found in the object-oriented programming paradigm. As the software industry has progressed, the availability of black boxes of all sorts has increased considerably. Development, in many cases, has taken on the characteristic of white-box manipulation of black-box effects. This is especially true within the use of the general AI tools and domain tools. Along with availability, the size and complexity of the black boxes has grown as well. Several issues become more predominant: the predictability of runtimes declines, black boxes become known more by their behavior as the potential for obtaining a white-box view becomes more remote, verification becomes more difficult [8].

The box example illustrates some of the problems. On a closer look, there will be specific issues related to languages-grammars, their associated algebras-logics-operators, interpretation functions, and a myriad other technical details. Advanced work in computing logic can focus on the language and what it can convey or on the concept and how to express it [7]. A lot of the discussion revolves around 'knowledge' and what it means in different contexts.

Research papers reporting results appear continually (some are mentioned in References). From a practical viewpoint, though, the major idea is to support a development approach that assumes that the more complex environments need tools, an empirical basis, and an experimental mindset. Tools that facilitate analysis, measurement, and continued improvement of these white-black box collections, that we call code (e.g. rules, concepts, functions, objects), will be the focus [14].

Figure 2 provides a view of the software process and shows the relationship between the above-the-floor (*atf*) human view and the below-the-floor (*btf*) computer view. The emergence of the object paradigm, which attempts to model the world from a common framework, has paralleled the advance in knowledge systems. The *atf* effort addresses how to determine the problem representation sufficiently that *btf* model can be built. Semantic objects from the *atf* world will relate to specific objects of the *btf* world [12]. The main problem of software is to provide workable morphisms between the *atf* view and the *btf* constraints. This can be phrased in the problem versus solution space differences. The growing interest in rapid development partly results from a recognition that handling these morphisms is best supported by tools that match well with the human intuitive view of a problem [10]. This subject will be addressed further in the next section.

As an example of a tool, take the general AI tool that provides the basis for developing a KBS application. ADS (AION Development System) offers an interactive development environment with editors for defining and modifying objects that make up the functionality of the KBS, e.g. classes, rules, states, vocabularies, slots, and other objects. In addition, ADS offers an interpreted execution environment with analysis and debugging support. The features of ADS allow domain knowledge representation to be developed and demonstrated quickly. The focus of ADS is general and does not limit the types of application domains in which it is used. There are add-ons to ADS, such as the VGE (Virtual Graphics Environment) product of Stone and Webster, that provide a specific domain focus, in this case, CATIA (product of Dassault Systemes) CATGEO programming. The ADS/VGE combination provides a rich environment for developing intelligent systems with the design/manufacturing domain.

As another example, ICAD is a domain (intelligent design) tool where the application domain is the design of parts; ICAD accomplishes its tasks through the provisions of a language, called IDL, and its supporting environment that allows the definition of parts, their attributes (including features), and relationships between parts. Parts can be related several ways using IDL, including user-defined relationships, and the language allows the definition of geometric constraints. The IDL language statements are converted into compilable code automatically, and compilation can be accomplished incrementally. The ICAD environment provides a browser that graphically supports perusal of parts, their attributes, their relationships, and their

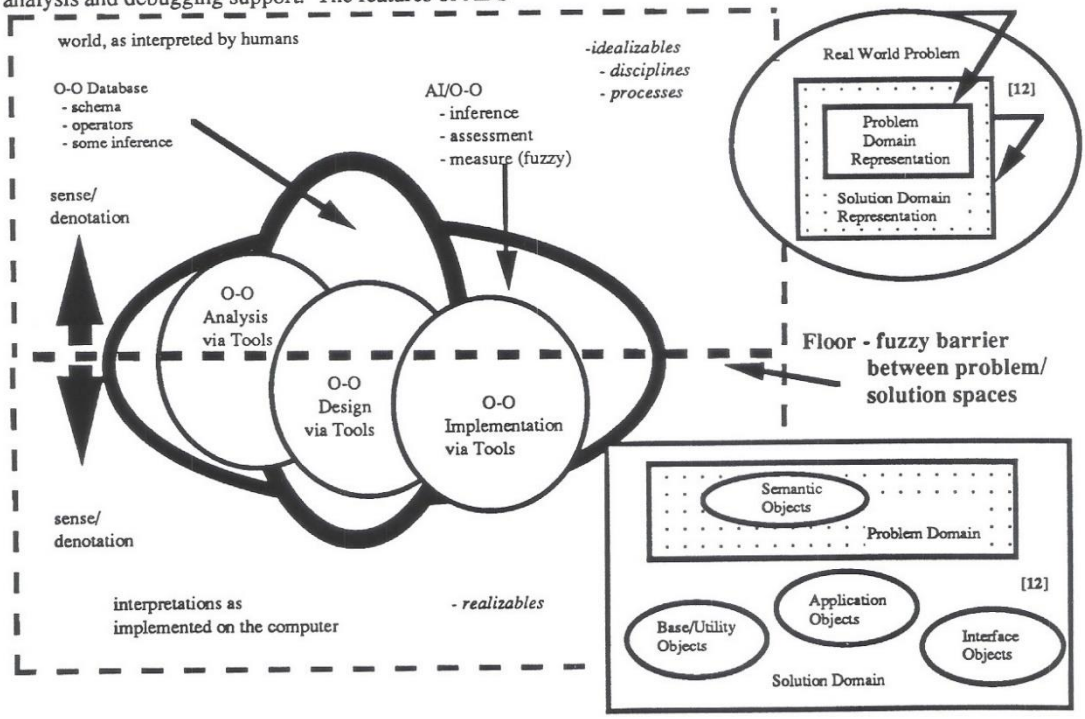

Figure 2. Software Trends and Issues of Rapid Development

geometric characteristics. Also, ICAD provides linkages to several CAD systems, including CATIA.

The chief advantage of these domain tools is that the morphisms between the problem and the solution spaces are specific and thereby constrained in the particular manner required by the domain [12]. As well, objects and their operators will have already been defined such that they incorporate some domain knowledge. The emphasis on both analysis and design with a domain tool will be at a higher level than with a general class of tool, allowing earlier focus toward implementation via prototyping.

Given the availability of a domain tool, a KBS/KBE project can take an approach that makes use of incremental cycles consisting of design, construction, and measurement of operable code. The measurement focuses on early and continual added-value to the customer.

This approach can be applied in many application areas. It is especially useful in disciplines with large and complex domains where analytic views are difficult to attain, though portions of the domain may be analytically verifiable. The approach emphasizes identification of goals that can be incrementally attained; the approach must be accompanied with rigorous testing and validation - VVT; and the customer will recognize the validity of the approach through value added by the computer.

One area that requires focus during prototyping is that of computability. Another issue is that of scale-up. A solution in the small does not necessarily continue to work as the problem size is increased. Also, the characteristics of the problem space that might assist in specifying bounds for the solution space need to be explored. The justification for prototyping are many; one might look at this approach as performing concurrent engineering of software.

3 COMPUTABILITY, COMPLEXITY, AND LOGIC

First the paper looked at how the computing paradigm can be discussed in terms of boxes of varying degrees of color and complexity; then the paper reviewed how the software development aspect of computing is undergoing change. The differences in the development approach that are required by KBS/KBE and advanced computing have been presented. Now, the paper looks at why these issues come into play in order to prepare the stage for a discussion of VVT. In general, the computability of a problem will be largely influenced by the domains involved, and a lot of computing success has resulted from good domain understanding. However, many assumptions about the knowledge of a domain do not hold up under the rigorous demands of computing. There are many reasons for the mismatch if one views the issue from the *atf/btf* framework. We need to build an experimental/empirical viewpoint, in which computing failure, in itself, is not necessarily bad; some domains are poorly understood and need clarifying analysis. Any computing experience, good or bad, if viewed properly will help expand what is known about the domain. A potential fractured/fractal (e.g. unconnected or multi-connected regions within the problem and solution spaces) nature in some domains will best be explored via automated means supported by good human intuition and processes.

Let's look at computability. Its definition will include words relating to whether an object has a certain domain property as expressed by a predicate and how to decide that this is so. The decidability of the predicate can be reduced to finding a function that provides an answer. To show that such a function is computable, it is "sufficient to give an algorithm that computes it." However, without a precise definition of the algorithm, "all such demonstrations are open to question until they are executed [13]." It is even more uncertain to show that a function is not computable. Hence, this is an important subject to KBS and KBE, in particular, and all software, in general. We can work toward precision in specifying algorithms, however, given the inadequacy of the algorithmic approach, heuristics (rules-of-thumb) are often more successful. The question of computability can be seen to rest with functions and computing functions.

Some discussions about computability involve the definition of an abstract machine, e.g. Turing, that can be defined for a function. However, there is one major problem with an abstract machine. There is no guarantee that the machine will stop computing and provide an answer. Hence, a lot of effort is put into increasing the chance that a computation will succeed. One can think that there is a Turing (or similarly defined machine) for each function. A thesis has been proposed (by Church) that suggests that there is an associated abstract machine for a function, if the function can be solved intuitively [1]. However, there can be some discussion about aspects of mathematics, to date not computed or partly computed, yet handled by human expertise; this paper will ignore such philosophical matters.

To be solvable, the algorithm for the function must be expressed in the precise manner suggested above. By no means have all possible functions been reviewed. Yet, several unsolvable functions have been identified. One can think of two major classes of unsolvability. The first is directly related to the

limits of the Turing machine. The other is related to complexity which will be discussed below.

Hence it has been commonly accepted that not all functions are solvable or computable (though they can be approximated). Some fundamental questions might be: are all unsolvable problems equivalent, is there a reasonable approximation (e.g. heuristic) to make a problem solvable, how good is the approximation, is its execution time acceptable, and how do we judge approximations.

Figure 3 illustrates how these concepts relate. There is a large class of functions, some of which are solvable intuitively. Some of the latter may or may not be computed in an algorithmic manner (they might be computed heuristically, for instance). The belief is that an algorithm can be computable (complexity, though, is a definite factor.). The work in this area uses mathematical induction quite frequently. There are issues that arise from the technique and whole classes of computing are outside of the discussion, yet these have made great stride. One definite factor in the argument is that human expertise can bring to bear talents affecting computing success. We will look at this issue a little later.

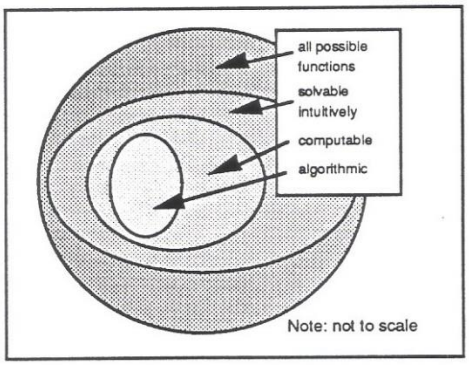

Figure 3. Functions and Computability.

The success of computer science rests with identifying computable functions, decidable predicates, and solvable problems [13]. The analysis development task can assist in reformulating problems on the *atf* side such that they might be computable on the *btf* side. The design development task can assist in providing a means to determine computable definitions. However, prototyping is the prime means to provide a proof of computability and a measure of effectiveness of the solution. As well, prototyping can help to experimentally explore unknown aspects of a domain.

A second class of unsolvability deals with complexity. Given the intricacies of knowledge in many application areas, a solution will likely have characteristics that become computationally complex. This means that one must resort to an approximation [3], however any approach to computing must include applied methods that are incremental, empirical, experimental, and measurable. The complexity of an algorithm can be stated in terms of the space required by, and the time utilized in, execution. The latter is considered the more important and can be stated in terms of a lower and upper bound. Heuristics can be handled similarly, though it will be with less precision.

Several problems have been identified in terms of their complexity as being unsolvable except through approximation. The solution might be either algorithmic or heuristic. Unfortunately, the types of problems faced by AI/KBS/KBE, like those of operations research, have generally been of this very complex nature [8].

A class defined for these problems is called nondeterministic polynomial (NP) [3]. Essentially, this means that an equation specifying the time required to determine a solution on a nondeterministic abstract machine will not be a simple polynomial. Some of the known hard problems are: 1) formula satisfiability - given a collection of boolean formulas, is it satisfiable? 2) clique problem - given an undirected graph, what is a complete subgraph of any arbitrary size? 3) vertex-cover problem - given an undirected graph, what is a vertex cover of any arbitrary size?. Other problems include: hamiltonian-cycle, subset-sum, traveling-salesman, subgraph isomorphism, integer-programming, set-partition, graph-coloring, bin-packing. Yet, despite this large list, working approximations have been proposed for many of these problems, and these have been applied successfully.

Now, considering logic, the basis for computer systems was established after Boole's 1847 papers. Since then, there has been much work in mathematical and computational logics. The past twenty years has seen a lot of activity in establishing higher-order logics, theorem provers, and universal algebras. In general, even simple logics, such as first order, are undecidable. One can relate this characteristic to the notion that termination of an algorithm in the Turing sense can not be known to be true before the occurrence.

As well, at the lowest level of execution, satisfiability of a collection of boolean formulas is in the NP class, however there are fragments of program logic that are decidable and thereby computable. Successful solutions consist of approximating

algorithms that consist of various mixtures of these fragments. The issue will be, then, to map known solutions and test the veracity of the approach, which implies using an experimental approach. Consistency becomes an operational issue: attempt to preserve it during execution.

Of the several approaches that have been proposed and researched for handling these problems of logic, an increasingly common approach has been to focus on developing an object basis for the system. This approach allows the issues of logic to be subsumed into a modeling technique that matches well the intuitive human grasp of entities, their characteristics, and relationships [6][12]. This approach does not remove the problems, it just puts them into a different context that makes them more tolerable. The focus can be on a constructive build of capability and reuse of known computable functions, though, there will always be the issue of proving that the solution fits the problem and that consistency issues are resolved.

Other efforts in logic have been directed toward defining nonmonotonic logics [4]. These approaches have evolved from the real requirements within complex domains when dealing with temporal, causal, and modal issues. The qualification problem recognizes that the number of preconditions for an action can be immense. The issue is how to reasonably constrain the scope, else the amount of computation before a decision will grow too large, thereby compounding the decision action. The frame problem recognizes that in any situation there will be those things that do not change. For any decision, a need to specify in their entirety all the constants, would amount to a tremendous overhead of bookkeeping. The ramification problem recognizes the unreasonableness of explicitly recording all the possible consequences of actions.

Some of these issues of logic come into play with ADS and ICAD. In the case of ADS, there is the requirement to explicitly state defaults, as needed, or to extend the logic to handle a closed-world approach. The value of "known as false" is not the same as "unknown." unless one makes the assumption that one is dealing with a closed-world. This allows one to decide that the absence of an object with a particular attribute denotes that the attribute is not true for any object. The establishment of defaults will reduce the occurrences of the unknown state, however these defaults must be consistently maintained during the system life cycle. On the other hand, if one handles this problem by extending the logic, then there is the added complexity of these statements and the possible effects on the computation. As can be expected, either approach has implications related to VVT.

In the case of ICAD, part of the power of ICAD is that it implements a lazy-evaluation scheme for determining the values of non-bound variables. This approach essentially reduces computational complexity to a manageable portion in that the whole is not needed in order to derive a part. Or in terms of a breadth versus depth expansion, ICAD is mixed and will allow the minimal path to be followed. However, the downside is that a demand for a value will cause dependency-driven backtracking in an attempt to determine a value. The amount of computation involved with such a demand is not known deterministically which can cause unpredictable runtimes. In many cases, search in computation will loop. Termination constraints need to be imposed externally.

The reality of these issues is an integral part of the current state-of-the-art in computing. Knowledge systems solve problems similar to many of the above, hence the issues of computability, complexity, and logic are very important to these systems.

4 REQUIREMENT: VERIFICATION, VALIDATION, AND TEST (VVT)

The future always has uncertainties, however there is no question about the following: 1) The demands upon computer systems will continue to increase through more types of computer applications being attempted. One example is the growing use of the computer for support of design, planning, and manufacturing of complex parts. 2) The complexity of computer systems will continue to grow due to the nature of the application domains that will be need to be understood. An example is the hard problem of representing and making decisions within a design and feature space. Increasing, too, will be the efforts required to certify a computer system through VVT. Though a wide-range of techniques will come into play, testing will continue to be of upmost importance for KBS/KBE.

In general, the development of a computer system can have well-defined points at which VVT activity might occur (See Figure 4). The user expresses the intent for a computer system in the form of requirements. The interpretation of these, in the traditional sense, will be in the form of a specification which, then, drives the program development. The program, after testing, gets released to the user for evaluation. In the traditional sense, the stages were more clearly marked than one finds with KBS/KBE. Now, a program can be verified by checking whether it does what it is specified to do. This is mostly a computer-related problem. Validation, on the other hand, concerns itself with whether a program does what it should do in the larger scope that includes the intent and the expectation, as well.

Let's look a little more at verification. It can be approached from several angles, usually via walkthroughs and code analysis. There can be automated approaches followed for algorithms, though only the most simple algorithms will be successfully verified, in practice. Heuristics and knowledge are not as easily verified as are algorithms [8]. Research into verification covers several areas, such as formal (mathematical) specifications, theorem proving techniques, and code generation from specification [9][14].

Formal approaches to verification are plagued by the problems of computability. Program logic without bindings is decidable assuming a closed domain. The inclusion of bindings, such as quantifiers and procedure definitions, make the problem undecidable. Then too, any thoroughly proven specification, given the state of the art, does not encompass some important factors, such as influences from the runtime environment. In short, the work and resources for formal methods (one program == many pages of proof) are excessive. Now, as tools for code generation improve, the effort at verification will be more directed toward evaluating (simulating) the specification itself [10].

This brings up validation which is the harder and wider issue of whether a program does what it is supposed to do. This issue involves underlying questions of knowledge, epistemology, ontology, the representation of knowledge, and the particulars of a domain [2]. Among the criteria for evaluation will be validity, usability, reliability, and effectiveness. Validity will need to consider consistency and completeness. Given the added complexities and uncertainties, knowledge processing projects are required to base their activities upon incremental, cyclic, empirical, evolutionary approaches that favor early and continual value-added results and measurements.

One good VVT approach utilizes the mathematical techniques of experimental design, augmented with domain knowledge. An example would be a modification of other techniques, e.g. Taguchi's signal/noise, Mill's Cleanroom Engineering [11], and structured testing. Such an approach would include multivariate sampling, measurements via fuzzy evaluations, and black box functional views. One major characteristic would be that testing would be extended into the full life cycle of the system through continual/periodic auditing of a system's veracity [8].

There are two issues that come into play with KBS/BKE that complicate VVT. The inference engine brings in additional factors (e.g. undecidability, nondeterminism) which thwart methods that look at code statically. And, an attempt to bring in the domain raises the issues of complexity. Some attempts at solving these problems range from taking a diagnostic viewpoint [9] to focusing on an operational viewpoint [16][17]. The pitfalls lie with the assumptions about and current understanding of the domains. Additionally, there is a potential pitfall

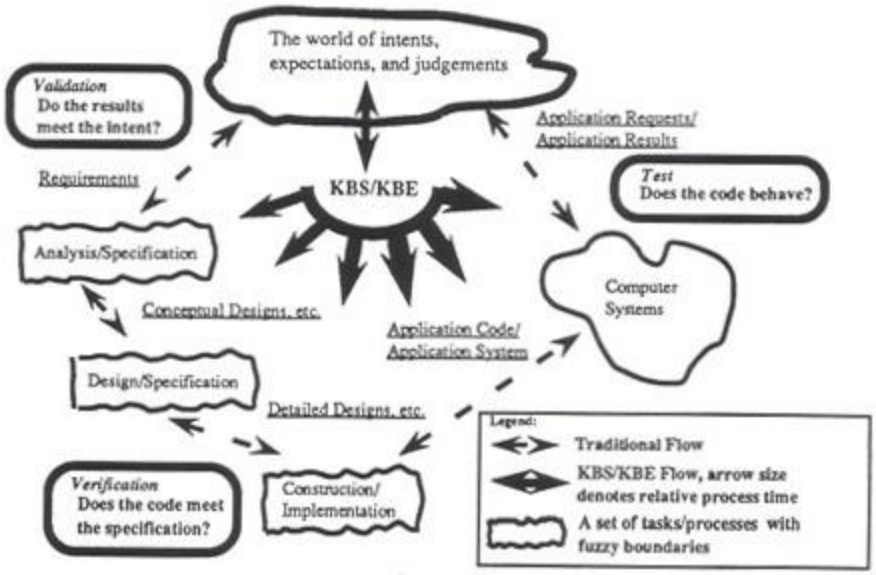

Figure 4. Verification, Validation, and Test

always present in the development of KBS and KBE systems: the verification and validation efforts themselves are as prone to succumb to the issues of computability and complexity as are the systems undergoing testing.

5 CONCLUSION

Computability and complexity are real issues in solving problems by computer. The chasm of unsolvability is bridged through approximations that are comprised of collections of computable patterns. This applies to testing as well as to other aspects of computing. Combinatorics preclude the use of "exhaustive" techniques, except for small cases. Verification and validation are important pursuits. Testing can be accomplished through computationally supportable approaches.

Despite all the problems of complex computing, people with their talents and expertise have brought a lot to the equation that is important to any practical computing success. This will continue to be so; one focus needs to be toward increasing the amount, and capability, of tools for supporting good people in evaluating a system in the context of verification and validation and toward having good processes in place for handling requirements and development.

The continued success of KBS/KBE will be founded upon a reasonable and creative approach to overcoming the limitations of computing within the constraints of possibility, cost, and validity.

6 REFERENCES

[1] Boolos, G.S. and R.C. Jeffrey. 1989. *Computability and Logic*. Cambridge, UK: Cambridge University Press.

[2] Collins, Harry M. 1994. The Nature of Scientific Knowledge - Some Implications for Aritficial Intelligence. *Phi Kappa Phi Journal*. 2:28-31.

[3] Cormen, T.H. et al. 1990. *Introduction to Algorithms*. Cambridge, MA: The MIT Press.

[4] Ginsberg, M.L. 1987. *Readings in Nonmontonic Reasoning*. Los Altos, CA: Morgan Kaufmann Publishers, Inc.

[5] Girard, J-Y and Y. Lafant, P. Taylor. 1989. *Proofs and Types*. UK: Cambridge University Press.

[6] Gougen, J.A. and R.M. Burstall. 1992. Institutions: Abstract Model Theory for Specification and Programming. *Journal of the ACM* 39:95-146.

[7] Guha, R. V. and Douglas B. Lenat. 1994. Enabling Agents to Work Together. *Communications of the ACM*. 7:127-142.

[8] Guida, G. and G. Mauri. 1993. Evaluating Performance and Quality of Knowledge - Based Systems: Foundation and Methodology. *IEEE Transactions on Knowledge and Data Engineering*. 5:2:205-225.

[9] Loiseau, L. 1993. A Diagnosis Approach to Formulate Validation of Knowledge Based Systems. In *IJCAI-93 Workshop on Validation, Verification and Test of KBSs*. 51-58.

[10] Lowry, M.R. 1992. Software Engineering in the Twenty-First Century. *AI Magazine* 13:71-87.

[11] Mills, Harlan D. 1988. Stepwise Refinement and Verification in Box-Structured Systems. *IEEE Computer* 6:23-36.

[12] Monarchi, D.E. & G.I. Puhr. 1993. Research Typology for Object-Oriented Analysis. *Communications of the ACM*. 9:35-47.

[13] Ralston, A. and E.D. Reilly (Eds). 1983. *Encyclopedia of Computer Science and Engineering*. New York: Van Nostrand Reinhold Company, Inc.

[14] Rousset, M-C and P. Hors. 1993. A formal framework for structures & deductive knowledge consistency checking. In *IJCAI-93 Workshop on Validation, Verification and Test of KBSs*. 59-66.

[15] Vickers, S. 1989. *Topology via Logic*. Cambridge, UK: Cambridge University Press

[16] Williamson, K. and M. Dahl 1993. Knowledge Base Reduction for Verifying Rule Bases containing Equations. In *AAAI-93 Verification and Validation Workshop*.

[17] Zlatareva, N. 1993. Distributed Verification and Automated Generation of Test Cases. In *IJCAI-93 Workshop on Validation, Verification and Test of KBSs*. 67-77.

7 ACKNOWLEDGEMENTS

Author wishes to acknowledge discussions with Mike Wallace, Roger Stumps, John Huffman, Mark Dahl and other co-workers concerning topics in the paper.

8 BIOGRAPHICAL SKETCH

JOHN M. SWITLIK is an Advanced Computing Technologist with BCS (IDS/Sim Group - TIS/CAS-W). His areas of interest are: intelligent/evolutionary systems applied to CAD/CAE and the mathematic/logic basis of computing theory/practice. Current projects include intelligent design, KBE, and concurrent SE. He can be contacted at: jms5326@ks.boeing.com.

Mid-America Conference on Intelligent Systems (MACIS 1994)

"Applications in Manufacturing and Service Industries"

Oct. 27 & 28, 1994
Marriott Hotel, Overland Park Kansas

Proceedings

Edited by:

Farhad Azadivar
Director
Advanced Mfg. Institute
Kansas State University

David Ben-Arieh
Associate Professor
Industrial Engineering
Kansas State University

Shing Chang
Assistant Professor
Industrial Engineering
Kansas State University

Hani Melhem
Assistant Professor
Civil Engineering
Kansas State University

Sponsored by:
The Advanced Manufacturing Institute (AMI)
Kansas State University, Manhattan, KS

Co-Sponsor:
Center for Excellence in Computer Aided Systems Engineering,
University of Kansas, Lawrence, Kansas

Participating Organizations:

University of Nebraska-Lincoln

University of Missouri-Rolla

Institute of Industrial Engineers - Greater Kansas City Chapter

Silicon Prairie Technology Association

AlliedSignal, Inc.

Boeing Commercial Airplane

Mid-America Manufacturing Technology Center (Kansas)

American Association for Artificial Intelligence

Black & Veatch

Institute for Electrical and Electronics Engineers- Kansas City Computer Society

Data Discovery, Inc.

In Cooperation with:

Oklahoma Center for Artificial Intelligence and the Oklahoma Symposium on Artificial Intelligence
Oklahoma State University, University of Oklahoma and University of Tulsa

Conference Committee:

Farhad Azadivar	*Conference Chair* , Advanced Manufacturing Institute, Kansas State University
Eddie Fowler	*Program Chair*, Kansas State University/Electrical & Computer Engineering
Susan Jagerson	*Conference Coordinator*, Advanced Manufacturing Institute, KSU

Organizing Committee:

John Atkinson	Data Discovery, Inc.
Susan Catts	Silicon Prairie Technology Association
John Cheung	University of Oklahoma
Fred Choobineh	University of Nebraska-Lincoln
Cihan Dagli	University of Missouri-Rolla
Dave Douglass	AlliedSignal, Inc.
John Switlik	Boeing Commercial Airplane
Costas Tsatsoulis	Center for Excellence in Computer Aided Systems Engineering
John Voeller	Black & Veatch
Caroline Zumbrunnen	Mid-America Manufacturing Technology Center

Technical Committee:

Farhad Azadivar	Kansas State University/Industrial Engineering
Eddie Fowler	Kansas State University/Electrical & Computer Engineering
David Ben-Arieh	Kansas State University/Industrial Engineering
Shing Chang	Kansas State University/Industrial Engineering
Prakash Krishnaswami	Kansas State University/Mechanical Engineering
Hani Melhem	Kansas State University/Civil Engineering

Table of Contents