Formal support for development of knowledge-based systems

Dieter Fensel, Frank van Harmelen, Wolfgang Reif, Annetie ten Teijet

Angaben zur Veröffentlichung / Publication details:

Fensel, Dieter, Frank van Harmelen, Wolfgang Reif, and Annetie ten Teijet. 1998. "Formal support for development of knowledge-based systems." *Failure and Lessons Learned in Information Technology Management* 2 (4): 173–82. https://doi.org/10.3727/108812898791918253.

Failure & Lessons Learned in Information Technology Management
Vol. 2, pp. 173–182, 1998

1088-128X/98 $10.00 + .00

Formal Support for Development of Knowledge-Based Systems

DIETER FENSEL,* FRANK VAN HARMELEN,† WOLFGANG REIF,‡
ANNETTE TEN TEIJE†

*Institute AIFB, University of Karlsruhe, 76128 Karlsruhe, Germany
†Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1018WB Amsterdam, The Netherlands
‡Faculty of Computer Science, University of Ulm, 89069 Ulm, Germany

This article provides an approach for developing reliable knowledge-based systems. Its main contributions are: Specification is done at an architectural level that abstracts from a specific implementation formalism. The *model of expertise* of CommonKADS distinguishes different types of knowledge and describes their interaction. Our architecture refines this model and adds an additional level of formalization. The formal specification and verification system KIV is used for specifying and verifying such architectures. We have chosen KIV for four reasons: (1) it provides the formal means required for specifying the dynamics of knowledge-based systems (i.e., dynamic logic), (2) it provides compositional specifications, (3) it provides an interactive theorem prover, and (4) last but not least it comes with a sophisticated tool environment developed in several realistic application projects.

Formal methods; Verification; Validation; Knowledge-based systems

INTRODUCTION

As computerized decision-making and control systems proliferate ever more widely, the results of hazards arising during their operation increase correspondingly. New decision-making technologies in medicine, manufacturing, industrial process control, and transportation are increasingly using knowledge-based technologies. The current techniques for developing Knowledge-based Systems (KBS) are unsatisfactory with respect to ensuring the safe operation of the systems they are meant to develop. For KBS development such safety-ensuring methods have in fact barely been developed.

As argued in the survey paper by Blair and others (Blair, Debenham, & Edwards, 1995), the major benefit that KBSs have over traditional Decision Support Systems or Information Systems is that they assist decision makers to gain *insight* into the difficult decision at hand rather than just solving the problem or producing a somehow "correct" model of the decision (Holtzman, 1989).

In order to ensure high-quality KBS, research in knowledge engineering developed methodologies, techniques, and tools for the construction of KBS. Although these methods were highly successful and widely adopted, almost all of them were often criticized from both within and outside the KBS community (e.g., from Software Engineering) for their informality and corresponding lack of precision. The conceptual models that are constructed in a modern Knowledge Engineering process are typically defined using a combination of natural language and graphical elements. As a result, these models are defined only informally or at best semiformally. As is well known, such documents in natural language have an ambiguous and imprecise semantics. There are as many meanings for a specification as there are readers, and it is a question of text interpretation as to whether a specification is as intended for a system. The ambiguity of such conceptual models in a Knowledge Engineering context was aptly illustrated by Aben's analysis of the use of a single inference step from the KADS framework in a number of papers in Bauer and Karbach (1992). Aben (1995, p. 36) states that out of nine papers that use the **abstract** inference step, only one author uses it in correspondence with Aben's interpretation of the original definition in Breuker et al. (1987, p. 37). Most of the other papers use different versions of the inference step, even including versions with different numbers of input arguments.

Although current KBS technology is well developed and actively used in practice, very little attention has been given in this technology to aspects like (i) functioning in an asynchronously changing environment, (ii) ensuring constraints on possible behaviors of the system, and (iii) dealing with uncertain observations and uncertain results of actions. These aspects are essential for safety-critical systems, and until such features are inte-

Correspondence and requests for reprints should be addressed to Dieter Fensel. E-mail: fensel@aifb.uni-karlsruhe.de

grated in existing KBS technology, this technology will not be usable for such safety-critical systems. French and Hamilton (1994) state: "Evidence from the computer industry suggests that without a methodology for comprehensive verification and validation, Expert Systems will not be considered safe and reliable for field use."

An example of a KBS application where safety considerations are paramount is the following. Oncology treatment consists of applying complicated protocols to cancer patients. These general protocols are formulated by specialists, and must be applied to individual patients. In such an individual application, many decisions must be made on the basis of current status and specific properties of the particular patient (drug doses, drug variety, treatment duration, treatment frequency). Doctors need support in applying and executing these protocols. Support can be given by knowledge-based DSSs (demonstrator applications have already been built) (Fox, Das, Elsdon, & Hammond, 1955; Roer, 1998). Assurance is needed that such a system does not exceed safety limits.

Besides the needs of safety-critical applications, there is a second important motivation for quality assurance on KBS. Because of the strong separation of inference and knowledge in KBS (see later sections of this article), there has always been a strong emphasis in KBS research on the development of reusable components, and on the construction of libraries of such components.

Quality assurance is particularly important in a library of reusable components, because:

- item components are used more than once, thereby increasing the cost of errors in a component (this repeated use can also serve to ammortize the increased costs that must be made to obtain the higher quality),
- components are used in a software context for which they were not originally developed, increasing the risk of unforeseen and therefore incorrect functioning of the component,
- components are used by programmers who did not develop the components, again increasing the risk of unforeseen usage of the component.

An additional complication with reusable components is that (through their use in different software contexts) they will have to be adapted to suit the specific details of each application. Because of this, the components must be adaptable, rather than fixed and immutable.

In traditional software engineering, the problems with informal specifications have lead to the development of formal approaches to software specification and design. Formal methods in knowledge engineering share the benefits of rigor and preciseness with formal methods in software engineering. Consequently, it may seem attractive to apply existing formal methods from software engineering to knowledge engineering.

In this article we outline our approach towards this goal. The next section discusses particular ways of describing the architectures of KBSs. The third section describes why these architectures (and the way they are used) are different for KBS than for arbitrary software. The fourth section describes how the formal specification and verification system KIV can be used to support the formal development of KBS that are based on our particular architecture. Finally, the last section concludes.

SOFTWARE ARCHITECTURES FOR KBS

In this section we describe a particular software architecture for KBSs. We do this by following the historical development of the field, starting from implementation-oriented descriptions in the late 1970s and 1980s, moving to implementation-independent descriptions in the early 1990s, and generalizing these architectures to facilitate reuse of components in recent work. This section ends by pointing out the consequences of all this for validation and verification of KBS.

The first era of KE technology stretched from the late 1970s to the mid-1980s. This period was characterized by the development of new programming techniques. New systems were described in terms of the representation techniques that they employed: rules, frames, Horn clauses, semantic networks, etc. KBS development environments gave support at the level of these representation techniques, and often aimed at integrating these different representations (e.g., ART, KEE, Knowledge Craft; see Jaurent, Ayel, Thome, & Zeibelin, 1986, for a comparison). Such programming techniques were often developed and widely used before a proper formal understanding of them was available.

The move away from this first period was already heralded as early as 1980 by Alan Newell in his "knowledge level" lecture (Newell, 1982), but should more properly be situated as late as 1985 when Clancey (1985) published his "heuristic diagnosis" paper. In this paper, Clancey analyzed a number of systems at a higher level of abstraction than simply their code. In particular, he identified a specific problem-solving method that underlay the behavior of a number of systems, even though they were all coded in different ways. Clancey called this method "heuristic classification" and described it in terms of its essential inference steps and the types of knowledge manipulated by these inference steps. Most importantly, this analysis was entirely independent of the particular way this method could be programmed in any particular representation language. This type of analysis triggered the development of a number of Knowledge Engineering methodologies. The late 1980s saw a number of such methodologies, which were all aimed at so-called "knowledge level" analysis of KBS tasks and domains. Generic Tasks (Chandrasekaran, 1986), CommonKADS (Schreiber, Wielinga, de Hoog, Akkermans, & Van de Velde, 1994; Wielinga, &

Breuker, 1986), Methods-to-Tasks (Eriksson, Shahar, tu, Puerta, & Musen, 1995), and Role-limiting methods (Marcus, 1988) are some of the prominent examples of such methods.

These approaches all differ in the structure that they propose for analyzing knowledge, the degree of task specificity, their link with executable code, and many other properties, but all of them are based on the idea of constructing a "conceptual model" of a system that describes the required knowledge and strategies at a sufficiently high level of abstraction, independent of any particular implementation formalism.

As an illustration, we describe the structure of one particular such model, namely the Expertise Model from the CommonKADS methodology. A CommonKADS expertise model consists of three categories of knowledge: domain, inference, and task knowledge (for the purposes of this article we ignore the strategic knowledge). The first category contains a description of the *domain knowledge* of a KBS application. This description should be as much as possible independent from the role this knowledge plays in the reasoning process. The *inference knowledge* of a CommonKADS expertise model describes the reasoning steps (or inference actions) that can be performed using the domain knowledge, as well as the way the domain knowledge is used in these inference steps. The inputs and outputs of such inference steps are called "knowledge roles." CommonKADS represents the relations between the inference steps through their shared input/output roles, with a dependency graph among inference steps and knowledge roles. Such a graph, called an inference structure, specifies only data dependencies among the inferences, not the order in which they should execute. This execution order among the inference steps is specified as *task knowledge*. For this purpose, CommonKADS uses a simple procedural language with procedural decomposition to execute inference steps and predicates to test the contents of knowledge roles. These procedures can be combined using sequences, conditionals, and iterations. The CommonKADS architecture is illustrated in Figure 1.

As shown in Figure 1, the CommonKADS architecture consists of separate layers, with explicit connections between these layers. Primitive procedures at the task layer are mapped to inference steps at the inference layer, and knowledge roles at the inference layer are mapped to knowledge at the domain layer. These separations were introduced to maximize the possibilities for reuse. For example, a combination of task and inference layers that model a reasoning strategy for design could be reused and applied to different sets of domain knowledge (e.g., designing a ship or designing an airplane). Vice versa, the same domain knowledge (e.g., knowledge about cars) could be used for different inference and task layers (e.g., to either design a ship or to perform fault-analysis on ships).

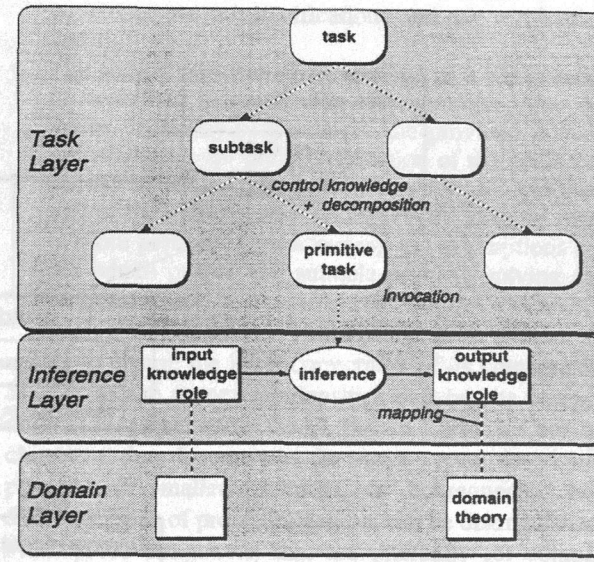

Figure 1. Structure of the KADS conceptual model.

In Fensel and Groenboom (1997), we presented an architecture for the specification of KBSs based on different reusable elements. This architecture is a refinement of the CommonKADS model of expertise, which has been widely used by the knowledge engineering community. Our framework for describing a KBS consists of three reusable elements: a task defines the problem that should be solved by the KBS, a problem-solving method (PSM) defines the reasoning process of a KBS, and a domain model describes the domain knowledge of the KBS (see Fig. 2). Each of these elements is described independently to enable the reuse of task descriptions in different domains and with different PSMs, the reuse of PSMs for different tasks and domains, and the reuse of domain knowledge for different tasks and PSMs. In terms of the standard CommonKADS model, the operational specification of the PSM_O corresponds to task and inference layer of Figure 1, whereas the domain knowledge (DK_K) corresponds to the domain layer of that figure. Our component-oriented architecture supplements each of these elements with interface definitions defining what a component requires and what it provides. A fourth element of a specification of a KBS is an *adapter* that is necessary to adjust the three other (reusable) parts to each other and to the specific application problem. It is used to introduce assumptions and to map the different terminologies.

A survey on the different elements and their relationships is provided in Figure 2. This architecture decomposes the entire system in smaller components of different types. In addition, it defines some structuring principles for the different components providing encapsulation of internal aspects. For example, a problem-solving method has two interfaces and an internal kernel.

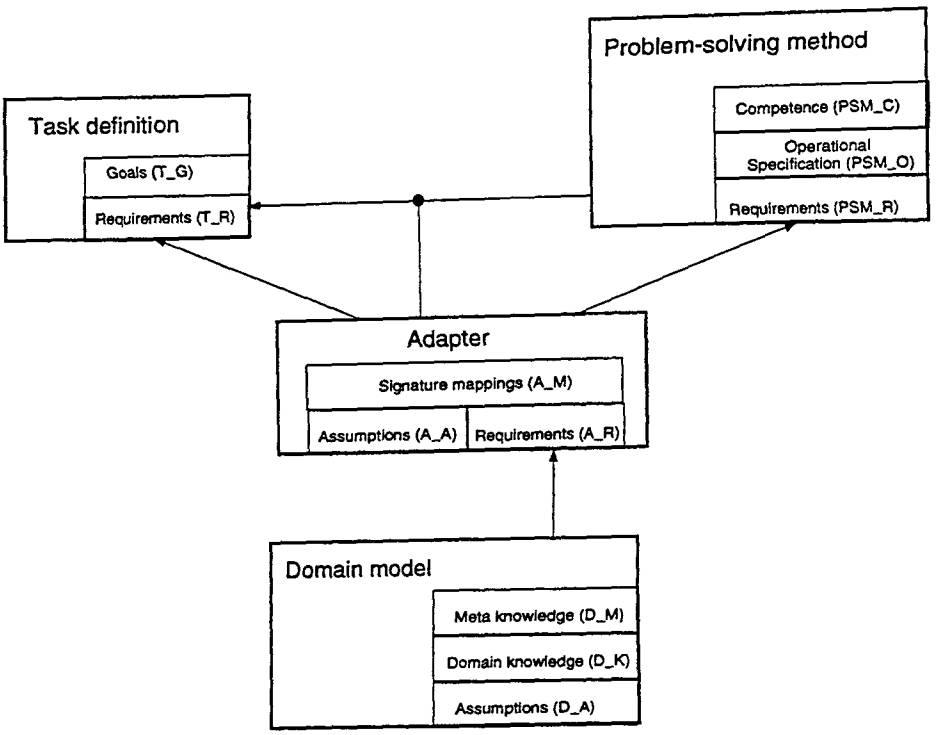

Figure 2. The four-component architecture for knowledge-based system.

First, it provides some functionality for solving a task as specified by its competence description. Second, it requires some functionality provided by domain knowledge as specified by its requirements interface. Finally, it internally defines a reasoning process that achieves the specified competence by using the domain knowledge as resource.

Our conceptual and formal model is a software architecture for a specific class of systems (i.e., KBSs). Software architectures have found increasing interest by the software engineering community in order to enhance the system development process and the level of software reuse (cf. Garlan & Perry, 1995; Shaw & Garlan, 1996). A software architecture decomposes a system into components and defines their relationships. This recent trend in software engineering works on establishing a more abstract level for describing software artifacts than was previously customary. The main concern of this new area is the description of generic architectures that describe the essence of large and complex software systems. Such architectures specify classes of application problems instead of focusing on the small and generic components from which a system is built up.

Much of the past research in validation and verification (V&V) of KBSs is done in terms of implementation languages, and mostly rule-based languages (Preece, Shinghal, & Batarekh, 1992). Such V&V systems check for such properties as loop-freeness of rule-bases, redundancy and reachability of rules, consistency, etc. However, it is well known from software engineering, and it is argued in more recent work on V&V of KBS (van Harmelen, 1998; Vermesan & Wergeland, 1994a, 1994b) that advantages can be gained by doing V&V in terms of a more abstract formal specification language, instead of an implementation language. If we are using an implementation-independent formal specification language, we can (i) do V&V in a much earlier stage of the life cycle: we can already verify properties of the system in terms of its specification, before the expensive effort of making an implementation, and (ii) we can separate the verification of functional, implementation-independent properties from the verification of implementation-specific properties. Finally, (iii) V&V is facilitated by the higher level of mathematical abstraction that can be employed in an implementation-independent specification language.

ITERATIVE REQUIREMENTS ENGINEERING

In KBS (and in AI in general), we are typically dealing with intractable problems: inference, planning, diagnosis, scheduling, design, and learning are all typical AI problems of which even the simple varieties are intractable. (A standard textbook even defines AI as the study of how to deal with intractable problems, Rich & Knight, 1991). As a result, no program exists that will compute correct solutions with an acceptable efficiency. Although such problems are not exclusive to AI, and are also known in traditional Software Engineering, they have received particular attention within AI.

In AI we exploit heuristics to solve such problems. The use of heuristics turns our programs into methods that approximate the ideal algorithm: our heuristic methods will sometimes fail to find a solution (incompleteness), they may erroneously claim to have found a solution (unsoundness), or they may find approximations of solutions when exact solutions are too expensive to compute. Besides the introduction of heuristics, KBS make such hard problems practically feasible by placing additional requirements on the domain knowledge and the possible inputs, or by removing some of the requirements on the desired outputs.

Such incompleteness and unsoundness clearly introduce a "gap" between what we want from our system (the *task-definition* from Fig. 2), and what the system manages to achieve (applying the *problem-solving methods* from Fig. 2 to the *domain-model*). The purpose of the *adapter* in Figure 2 is to characterize this gap, in terms of assumptions and requirements that are not met, although ideally they should have been. Therefore, making strong assumptions on available domain knowledge and problem restrictions is not a bug but a feature of KBSs that should enable efficient reasoning. For example, in Fensel and Benjamins (1998) we collected a large number of assumptions and related them to the different subtasks of diagnostic problem solving. This assumption list can be used to check given domain knowledge, to define goals for knowledge acquisition, to restrict the size of the problem that must be solved by the KBS, or to select a problem-solving method that fits to the given context as characterized by the established assumptions. However, this was a kind of postanalysis based on 20 years of developing system for model-based diagnosis. Therefore, later in the article we will sketch a more direct method (called *inverse verification*, cf. Fensel & Schönegge, 1998) that allows us to find and to construct such assumptions.

In practice, it is not realistic to capture this gap between ideal task-definition and actual competence of a system in one step. In Software Engineering, this process is known as evolutionary requirements capture: much time in verification effort is spent not in proving theorems that verify the behavior of the system, but in failing to prove nontheorems because of errors or missing assumptions in the system specification.

USING KIV TO DEVELOP KNOWLEDGE-BASED SYSTEMS

In the previous sections we identified two main requirements for a formalization and verification framework for KBSs:

- Formalization and verification must be possible at an architectural level (second section). We cannot base our framework on commitments to specific implementation formalisms like production rules. Instead, we have to make use of a conceptual model that structures our subspecifications and our verification tasks.
- The competence (i.e., functionality) of a KBS cannot be described independently of assumptions that describe necessary properties of the provided domain knowledge or the precise definition of the goals that should be achievable. In consequence, a significant part of the effort in developing a KBS must be spent in more precisely characterizing the assumptions under which proper and suitable problem solving can be guaranteed.

Establishing a proper composition of a system requires to ensure the local correctness of its components as well as the proper relationships between its components (i.e., global correctness). Figure 2 provides our architecture that decomposes the entire system into components of smaller grainsize. In consequence, two different types of proof obligations can be distinguished. First, proof obligations that are internally for components ensuring local correctness. These proofs can be reused together with the components. In the case of the domain model we have to ensure that the meta knowledge actually follows from the domain knowledge and the assumptions of the domain model. In the case of the task definition we have to ensure that the task is solvable given its requirements. In the case of the problem-solving methods we have to ensure a certain competence and the termination. Second, there are proof obligations that must be established during composing a system out of different components. These obligations ensure the global correctness of the composed system. The architecture helps to distinguish these different types of proof obligations and enables reuse as much as possible.

We discuss the specification and verification of the different elements of such an architecture and sketch the process of detecting hidden assumptions via inverse verification. We use the KIV system (Reif, 1995) for these activities. It is an advanced tool for the construction of provably correct software. KIV was originally developed for the verification of procedural programs but it serves well for verifying KBSs.

KIV

KIV supports the entire design process starting from formal specifications and ending with verified code. It offers functional algebraic as well as state-based modeling. Modular software and system designs can be specified, explored, validated, and verified with respect to formal safety and security models. KIV can be used as a pure specification environment with an integrated, user-friendly proof component. When it comes to implementation, specification components can be realized using applicative programs (in a Pascal-like syntax, and grouped together into modules). Proof obligations for implementation correctness are generated automatically, and can be discharged with KIV's proof component. KIV comes with a large library of reusable specifica-

tions, proved lemmas, and verified implementations that can be included into new developments. KIV supports a tight integration of proving and error detection and correction. This is important because most proof attempts are more likely to reveal errors in specifications, programs, and lemmas than to prove their absence.

KIV supports the entire design process starting from formal specifications (algebraic full first-order logic with loose semantics) and ending with verified code (Pascal-like procedures grouped into modules). It has been successfully applied in case studies up to a size of several thousand lines of code and specification (see, e.g., Fuchß, Reif, Schellhorn, & Stenzel, 1995). Its specification language is based on abstract data types for the functional specification of components and dynamic logic for the algorithmic specification. It provides an interactive theorem prover integrated into a sophisticated tool environment supporting aspects like the automatic generation of proof obligations, generation of counter examples, proof management, proof reuse, etc. Such a support is essential for making the verification of complex specifications feasible.

KIV provides mechanisms for combining elementary specifications to more complex ones (e.g., sum, enrichment, renaming, and actualization of parameterized specifications) that are common to most algebraic specification languages (cf. Wirsing, 1995).

In addition to (elementary) specifications, KIV provides modules to describe implementations in a Pascal-like style. A module consists of an export specification, an import specification, and an implementation that defines a collection of procedures implementing the operations of the export specification.

Finally, the KIV system offers well-developed proof engineering facilities: proof obligations are generated automatically. Proof trees are visualized and can be manipulated with the help of a graphical user interface. Even complicated proofs can be constructed with the interactive theorem prover. A high degree of automation can be achieved by a number of implemented heuristics. However, interaction is necessary for two reasons. In general, complex proofs cannot be completely automated, and proving usually means finding errors either in the specification or in the implementation. An elaborated correctness management keeps track of lemma dependencies (and their modifications) and the automatic reuse of proofs allows an incremental verification of corrected versions of programs and lemmas (see Reif & Stenzel, 1993). Both aspects are essential to make verification feasible given the fact that system development is a process of steady modification and revision.

Over the years, KIV was applied in a large number of substantial software case studies and pilot applications in academia and industry. Application areas were, for example, avionics, space systems, medicine, automotive electronics, system software, smartcards, and digital signatures. Given below are four examples.

Safe Command Transfer in a Space Craft.[1] In cooperation with the company Intecs Sistemi, Pisa, that developed the software, part of the guidance and navigation control (GNC) system of a space craft was treated formally, and reevaluated in KIV 3.0 at the University of Ulm. The given safety requirements have been verified, and a prototypical implementation has been proved correct. The major benefits of the formal verification were the detection of an error in the informal specification, and the explicit (and correct) specification of implicit assumptions.

Access Control. In this case study a generic access control model (based on Abadi, Burrows, Lampson, & Plotkin, 1991) is specified, implemented, and the implementation is proved correct. Furthermore, it was formalized and proved that it is not possible for a user to increase his rights without help from others. All specifications together contain about 1,100 lines of text, whereas the efficient implementation has a size of 1,200 lines of text. All in all 837 theorems and lemmas were proved. The overall time needed to complete the case study (including a vast number of modifications, error corrections, and reuse of proofs) was 14 weeks (see Fuchß et al., 1995).

Compiler Verification.[2] The case study is about correct compilation of PROLOG into code for the Warren Abstract Machine (WAM). In Börger and Rosenzweig (1994), the semantics of PROLOG is defined by a simple interpreter, which is refined in 11 steps to an interpreter of WAM machine code. The formal specification has more than 4,000 lines of text. The correctness of all the steps can be shown independently. In the course of verification several errors were revealed in the compiler assumptions as well as in the interpreters.

A Booking System.[3] A booking system for a national radio network was a formal redevelopment of the safety-critical kernel of an industrial project. The vast number of possible operations makes the specification (and implementation) large: the specification contains 4,000 lines and the implementation 7,100 lines of text. The verification effort amounts to ca. 2 person years.

Specifying the Architecture in KIV

The use of the KIV system for the verification of KBSs is quite attractive. KIV supports dynamic logic (cf. Harel, 1984), which has been proved useful in the specification of KBSs (cf. KARL) (Fensel, 1995), (ML)^2

[1]Sponsored by BSI, the German agency for IT security, and the European Space Agency, as a part of the VSE project (Hutter et al., 1995).
[2]Part of the DFG project "Integration of Automatic and Interactive Theorm Proving."
[3]Part of the VSE project (Hutter et al., 1995).

Architectural component		KIV structure	Ex.
Domain Knowledge	$\mapsto$	axiomatic theory in form of a structured algebraic specification	
Task	$\mapsto$	axiomatic specification	fig. 4
PSM specification	$\mapsto$	abstract program and axiomatic specification	fig. 5
Adapter	$\mapsto$	abstract program and axiomatic specification	

Figure 3. Mapping between architectural components and KIV structures.

(van Harmelen & Balder, 1992), MLPM and MCL (Fensel, Groenboom, & Renardel de Lavalette, 1998). Dynamic logic has two main advantages (especially if compared to first-order predicate logic). First, dynamic logic is quite expressive (e.g., we can formalize and prove termination or equivalence of programs or generatedness of data types). Second, in dynamic logic programs are explicitly represented as part of the formulas. Thus (especially if compared to the verification condition generator approach), formulas and proofs are more readable for people and provide more structural information that can be employed by proof heuristics.

KIV allows structuring of specifications and modularization of software systems. Therefore, the conceptual model of our specification can be realized by the modular structure of a specification in KIV. Figure 3 summarizes the way in which the components from Figure 2 are specified in KIV.

Formalizing the Components

In the following, we will sketch the specification of two components: a functional specification (we choose the definition of a task) and an operational specification that defines the control of the reasoning process of the problem-solving method.

The description of a task consists of two parts (cf. Fig. 4). First, it specifies a goal that should be achieved in order to solve a given problem. The second part of a task specification is the definition of requirements on domain knowledge necessary to define the goal in a given application domain. Both parts establish the definition of a problem that should be solved by the KBS. Contrary to most approaches in software engineering this problem definition is kept domain independent, which enables the reuse of generic problem definitions for different applications.

The concept of a problem-solving method can be found in most current knowledge engineering frameworks (e.g. Generic Tasks, Chandrasekaran, 198; CommonKADS, Schreiber et al., 1994; Method-to-task approach, Eriksson, 1995). A problem-solving method defines the control over a number of primitive inference actions (see Fig. 2). The inference actions (i.e., the elementary state transitions) are defined functionally. This combined specification style enables to specify explicitly as much control as required and hide further control aspects in the declarative specifications of the elementary steps. Providing an algorithmic solution for them has to be done during the design and implementation of the KBSs. During knowledge-level modeling we want to be able to abstract from it. The PSM we show in Figure 5 is set-minimizer (Fensel & Groenboom, 1997),

```
explanation = enrich abduction problem with
  constants goal: hypotheses;
  axioms
    complete(goal)
    parsimonious(goal)
end enrich

abduction problem = enrich hypotheses, data with
  constants observables: data;
  functions explain: hypotheses → data ;
  predicates
    complete: hypotheses,
    parsimonious: hypotheses;
  axioms
    complete(H) ↔ explain(H)=observables,
    parsimonious(H) ↔ ¬ ∃ H'. H'⊂H ∧ explain(H) ⊆ explain(H'),
    ∃ x . x ∈ observables
end enrich
```

Figure 4. Specification of the abductive task.

```
control = module
  export competence
  refinement
    representation of operations
      control implements output
    import inferences
    procedures hill-climbing(objects) : objects
    variables output, current, new: objects;
    implementation

      control(var output)
      begin
        hill-climbing(input,output)
      end

      hill-climbing(current, var output)
      begin
        var new = select(current, generate(current))
        if      new=current
          then output := current
          else  hill-climbing(new,output)
      end
```

Figure 5. The operational specification of the problem-solving method *set-minimizer*.

which minimizes sets by a one-step look-ahead search process.

Akkermans, Wielinga, and Schreiber (1993) define the competence of a problem-solving method independently from the specification of its operational reasoning behavior (i.e., a functional black-box specification). Proving that a problem-solving method has some competence has the clear advantage that the selection of a method for a given problem and the verification whether a problem-solving method fulfills its task can be performed independent from details of the internal reasoning behavior of the method. We will sketch some of these proof activities in the next subsection.

Verifying Components

When introducing a problem-solving method into a library we have to prove two aspects of the operational specification of the problem-solving method. We have to ensure the termination of the procedure and we have to establish some proven competence (i.e., functionality) of the component. When reusing the problem-solving method this proof need not be repeated and can (implicitly) be reused.

The according proof obligations are automatically generated by KIV as formulas in dynamic logic (cf. Harel, 1984). These proof obligations ensure that the problem-solving method terminates and that it terminates in a state that respects the axioms used to characterize the competence of the problem-solving method.

The next step is to actually prove these obligations using KIV. For constructing proofs KIV provides an integration of automated reasoning and interactive proof engineering. The user constructs proofs interactively, but has only to give the key steps of the proof (e.g., induction, case distinction) and all the numerous tedious steps (e.g., simplification) are performed by the machine. Automation is achieved by rewriting and by heuristics that can be chosen, combined, and tailored by the proof engineer. If the chosen set of heuristics get stuck in applying proof tactics the user has to select tactics on his own or activate a different set of heuristics in order to continue the partial proof constructed so far. Most of these user interactions can be performed by selecting alternatives provided by a menu.

To give an impression of how to work with KIV, Figure 6 is a screen dump of the KIV system. The current proof window on the right shows the partial proof tree currently under development. Each node represents a sequent (of a sequent calculus for dynamic logic); the root contains the theorem to prove. In the messages window the KIV system reports its ongoing activities. The KIV-Strategy window is the main window that shows the sequent of the current goal [i.e., an open premise (leaf) of the (partial) proof tree]. The user works either by selecting (clicking) one proof tactic (the list on the left) or by selecting a command from the menu bar above. Proof tactics reduce the current goal to subgoals and thereby make the proof tree grow. Commands include the selection of heuristics, backtracking, pruning the proof tree, saving the proof, etc.

Component Connection: Adapter Specification

The description of an adapter maps the different terminologies of the task definition, the problem-solving method, and the domain model. Usually an adapter introduces new requirements or assumptions, because, in general, most problems tackled with KBSs are inherently complex and intractable (cf. Fensel & Straatman, 1998). A problem-solving method can only solve such tasks with reasonable computational effort by introducing assumptions that restrict the complexity of the problem or by strengthening the requirements on domain knowledge.

Component Connection: Inverse Verification

The question may arise of how to provide such assumptions that close the gap between task definitions and problem-solving methods. In Fensel and Schönegge (1998), we present the idea of inverse verification. We start a proof with KIV that the competence of the problem-solving method implies the goal of the task. This proof usually cannot succeed but its gaps provide hints for assumptions that are necessary for it. That is, we use the technique of a mathematical proof to search for assumptions that are necessary to guarantee the relationship between the problem-solving method and the task. When applying the interactive theorem prover to an impossible proof it returns an open goal that cannot be proven but that would allow to finish the proof. Therefore, such an open goal defines a sufficient assumption. Further proof attempts have to be made to refine it to necessary assumptions (i.e., to minimize them). An assumption that is minimal in the logical sense (i.e., necessary) has the clear advantage that it maximizes the circumstances under which it holds true. It does not require anything more than what is precisely necessary to close the gap between the competence and the problem. In general, minimizing (i.e., weakening) assumptions can be achieved by analyzing their sufficiency proof with KIV and eliminating aspects that are not necessary for continuing the proof. However, besides logical minimality other aspects like cognitive minimality (effort in understanding an assumption) or computational minimality (effort in proving an assumption) may also influence the choice of assumptions.

CONCLUSIONS

In this article we have motivated the need for high-quality standards in Knowledge Engineering, given the re-

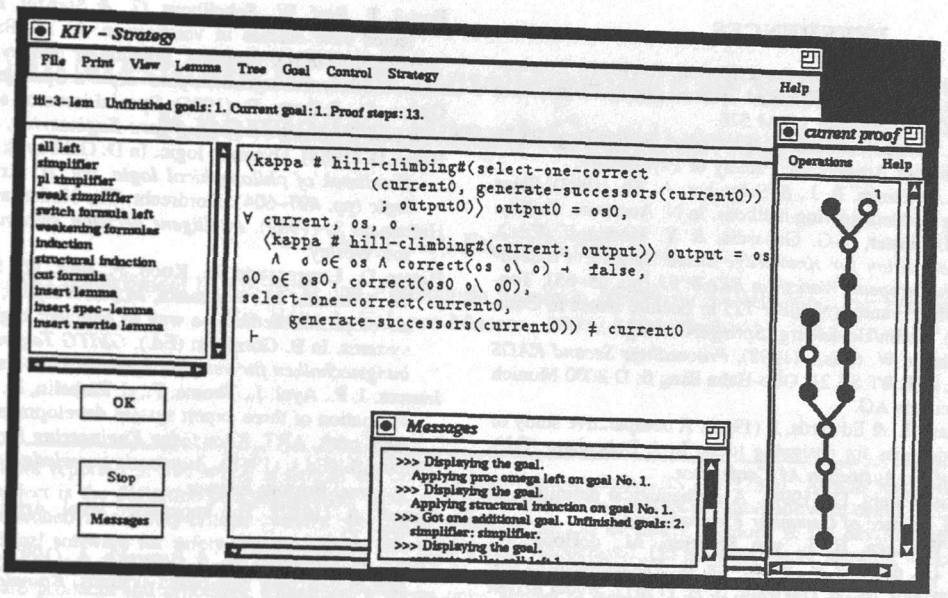

Figure 6. A screen dump of the KIV system.

quirements of safety-critical applications for KBSs. We have seen how the last decade of research in Knowledge Engineering has resulted in a number of KBS development methods that all aim at modelling knowledge and inference at a high level of abstraction, and do so using structured but informal representations such as diagrams or stylized natural language. Because of the informal nature of these high-level models, validation and verification efforts for KBS have mostly concentrated on implementation-oriented formalisms such as rules and frames.

More recent developments in Knowledge Engineering have made it possible to apply verification and validation techniques even at the high level of abstraction of the knowledge models. This is achieved by providing formal counterparts of these high-level models that are at the same time free from implementation details yet amenable to formal analysis by virtue of their mathematical structure.

The growing importance of libraries of reusable components in the construction of KBS has increased the need for high-quality software (besides the classical arguments concerning safety-critical applications). The need for adjusting predefined components in order to enable their reuse in varying environments has prompted refinements of the previously available development methods. Again, these refined methods can be given a mathematical basis.

In this article we have described the current state in Knowledge Engineering with respect to formal verifiability of software and software components. We have described one software architecture that is particularly suitable for reusable KBS components, and we have described how a formal verification tool from Software Engineering can be employed for the types of models used in Knowledge Engineering.

This use of formal tools to support quality assurance, reusability, and adaptability of KBSs and components dates only from recent years, and the work described is still in its laboratory stage. Many aspects of the use of KIV for formally specifying KBS can be improved, and we mention a few below.

First, as discussed in the third section, KBS often do not completely fulfill their idealized specifications because of the inherent complexity of their task. In ten Teije and van Harmelen (1998) we try to capture the notion of the *degree* to which a system satisfied its specification, as a replacement for the traditional notion of either meeting a specification in full or not at all. Second, KBS are typically interactive systems, and their interaction with the user (e.g., to obtain data or to provide explanations of the results) is as important for the success of a system as the final conclusion itself. However the dynamic logic in KIV is aimed at proving properties of the final output after termination of the program, and not so much at properties that must hold *during* execution of the program. Other logics, such as temporal logics, might well be better suited for such verification tasks, and research along these lines is already under way (Cornelissen, Jonker, & Treur, 1997). Finally, the use of KIV to specify KBS architectures has until now only been applied to small-scale examples. The IBROW project (see http://www.swi.psy.uva.nl/projects/IBROW3/home.html) is aimed at describing a full-scale library of reusably KBS components using the architecture described in the second section, and can provide the basis for an industrial strength verification of KBS components.

REFERENCES

Abadi, M., Burrows, M., Lampson, B., & Plotkin, G. (1991). A calculus for access control in distributed systems. In J. Feigenbaum (Ed.), *CRYPTO '91*. Springer LNAI 576.

Aben, M. (1995). *Formal methods in knowledge engineering*. Ph.D. thesis, University of Amsterdam, Faculty of Psychology.

Akkermans, J. M., Wielinga, B. J., & Schreiber, A. Th. (1993). Steps in constructing problem-solving methods. In N. Aussenac, G. Boy, B. Gaines, M. Linster, J.-G. Ganascia, & Y. Kodratoff (Eds.), *Knowledge Acquisition for Knowledge-Based Systems. Proceedings of the 7th European Workshop EKAW'93* (pp. 45–65), Toulouse and Caylus, France. (Number 723 in Lecture Notes in Computer Science). Berlin/Heidelberg: Springer-Verlag.

Bauer, C., & Karbach, W. (Eds.). (1992). *Proceedings Second KADS User Meeting*, ZFE BT SE 21, Otto-Hahn Ring 6, D-8000 Munich 83, 17–18. Siemens AG.

Blair, A., Debenham, J., & Edwards, J. (1995). A comparative study of formal methodologies for designing ideas. In J. Debenham (Ed.), *Proceedings of the Australian AI Conference*.

Börger, E., & Rosenzweig, D. (1994). A mathematical definition of full PROLOG. *Science of Computer Programming*.

Breuker, J. A., Wielinga, B. J., van Someren, M., deHoog, R., Schreiber, A. Th., deGreef, P., Bredeweg, B., Wielemaker, J., Billault, J. P., Davoodi, M., & Hayward, S. A. (1987). *Model driven knowledge acquisition: Interpretation models*. ESPRIT Project P1098 Deliverable D1 (task A1), University of Amsterdam and STL Ltd.

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

Clancey, W. J. (1985). Heuristic classification. *Artificial Intelligence, 27*, 289–350.

Cornelissen, F., Jonker, C., & Treur, J. (1997). Compositional verification of knowledge-based systems: A case study for diagnostic reasoning. In E. Plaza & R. Benjamins (Eds.), *10th European Knowledge Acquisition Workshop, EKAW-97* (pp. 65–80) (Number 1319 in Lecture Notes in Artificial Intelligence). Berlin: Springer-Verlag.

Eriksson, H., Shahar, Y., Tu, S. W., Puerta, A. R., & Musen, M. A. (1995). Task modeling with reusable problem-solving methods. *Artificial Intelligence, 79*(2), 293–326.

Fensel, D. (1995). *The knowledge-based acquisition and representation language KARL*. New York: Kluwer Academic Publisher.

Fensel, D., & Benjamins, R. (1998). The role of assumptions in knowledge engineering. *International Journal of Intelligent Systems*.

Fensel, D., & Groenboom, R. (1997). Specifying knowledge-based systems with reusable components. In *Proceedings of the 9th International Conference on Software Engineering & Knowledge Engineering (SEKE-97)*, Madrid, Spain.

Fensel, D., Groenboom, R., & Renardel deLavalette, G. R. (1998). Modal change logic (MCL): Specifying the reasoning of knowledge-based systems. *Data and Knowledge Engineering*.

Fensel, D., & Schönegge, A. (1998). Inverse verification of problem-solving methods. *International Journal on Human-Computer Studies*.

Fensel, D., & Straatman, R. (1998). The essense of problem-solving-methods: Making assumptions for gaining efficiency. *Journal of Human Computer Studies*.

Fox, J., Das, S., Elsdon, D., & Hammond, D. (1995). Decision making and planning by autonomous agents: A generic architecture for safety critical applications. In *Proceedings Expert Systems 95*. Cambridge: Cambridge University Press.

French, S., & Hamilton, D. (1994). A comprehensive framework for knowledge-base verification and validation. *International Journal of Intelligent Systems, 9*, 809–837.

Fuchß, T., Reif, W., Schellhorn, G., & Stenzel, K. (1995). Three selected case studies in verification. In M. Broy & S. Jähnichen (Eds.), *KORSO: Methods, languages, and tools for the construction of correct software—final report*. Springer LNCS 1009.

Garlan, D., & Perry, D. (1995). Special issue on software architecture. *IEEE Transactions on Software Engineering, 21*(4).

Harel, D. (1984). Dynamic logic. In D. Gabbay & F. Guenthner (Eds.), *Handbook of philosophical logic, vol. II: Extensions of classical logic* (pp. 497–604). Dordrecht, The Netherlands: Reidel.

Holtzman, S. (1989). *Intelligent decision systems*. New York: Addison-Wesley.

Hutter, D., Langenstein, B., Koob, F., Reif, W., Sengler, C., Stephan, W., Ullmann, M., Wittmann, M., & Wolpers, A. (1995). The VSE development method—a way to engineer high-assurance software systems. In B. Gotzheim (Ed.), *GI/ITG Tagung Formale Beschreibungstechniken für verteilte Systeme*. Universität Kaiserslautern.

Jaurent, J. P., Ayel, J., Thome, F., & Ziebelin, D. (1986). Comparative evaluation of three expert system development tools: KEE, knowledge craft, ART. *Knowledge Engineering Review, 1*(4), 19–29.

Marcus, S. (Ed.). (1988). *Automatic knowledge acquisition for expert systems*. Boston: Kluwer.

Newell, A. (1982). The knowledge level. *Artificial Intelligence, 18*, 87–127.

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

Reif, W. (1995). The KIV-approach to software verification. In M. Broy & S. Jähnichen (Eds.), *KORSO: Methods, languages, and tools for the construction of correct software—final report*. Springer LNCS 1009.

Reif, W., & Stenzel, K. (1993). Reuse of proofs in software verification. In R. Shyamasundar (Ed.), *Foundation of Software Technology and Theoretical Computer Science, Proceedings*, Bombay, India. Springer LNCS 761.

Rich, E., & Knight, K. (1991). *Introduction to artificial intelligence*. New York: McGraw-Hill.

Rogers, E. (1998). AI in health care. *IEEE Intelligent Systems and Their Applications, 13*(1). [Special Issue on AI in Medicine]

Schreiber, A. Th., Wielinga, B. J. deHoog, R., Akkermans, J. M., & Van de Velde, W. (1994). CommonKADS: A comprehensive methodology for KBS development. *IEEE Expert, 9*(6), 28–37.

Shaw, M., & Garlan, D. (1996). *Software architectures. Perspectives on an emerging discipline*. Englewood Cliffs, NJ: Prentice-Hall.

ten Teije, A., & van Harmelen, F. (1998). Characterising problem solving methods by gradual requirements. In *Proceedings of the Eleventh Workshop on Knowledge Acquisition for Knowledge-Based Systems (KAW'98)*, Banff, Alberta.

van Harmelen, F. (1998). Applying rule-base anomalies to KADS inference structures. *Decision Support Systems, 21*(4), 271–280.

van Harmelen, F., & Balder, J. R. (1993). $(ML)^2$: A formal language for KADS models of expertise. *Knowledge Acquisition, 4*(1). [Special issue: "The KADS approach to knowledge engineering," reprinted in A. Th. Schreiber et al. (Eds.), *KADS: A principled approach to knowledge-based system development*]

Vermesan, A., & Wergeland, T. (1994a). *Expert system verification and validation* (Working paper 92/1994). Centre for Research in Economics and Administration SNF, Bergen, Norway.

Vermesan, A., & Wergeland, T. (1994b). A formally-based methodology for deriving verifiable expert systems from specifications. In *Proceedings of the AAAI'94 Workshop on Validation and Verification of Knowledge-Based Systems*, Seattle.

Wielinga, B. J., & Breuker, J. A. (1986). Models of expertise. In *Proceedings ECAI-86* (pp. 306–318).

Wirsing, M. (1995). Algebraic specification languages: An overview. In *Proceedings of the 10th Workshop on Specification of Abstract Data Types*. Springer-Verlag LNCS 906.