[FeG97] D. Fensel and R. Groenboom: Specifying Knowledge-Based Systems with Reusable Components. In *Proceedings of the 9th International Conference on Software Engineering & Knowledge Engineering (SEKE-97)*, Madrid, Spain, June 18-20, 1997.

[FeS96] D. Fensel und R. Straatman: The Essence of Problem-Solving Methods: Making Assumptions for Efficiency Reasons. In N. Shadbolt et al. (eds.), *Advances in Knowledge Acquisition, Lecture Notes in Artificial Intelligence (LNAI)*, no 1076, Springer-Verlag, Berlin, 1996.

[FeS97] D. Fensel and A. Schönegge: Assumption Hunting as Developing Method for Problem-Solving Methods, In *Proceedings of the Workshop on Problem-Solving Methods for Knowledge-Based Systems during the 15th International Joint Conference on AI (IJCAI-97)*, Nagoya, Japan, August 23-30, 1997.

[FRS+95] Th. Fuchß, W. Reif, G. Schellhorn and K. Stenzel: Three Selected Case Studies in Verification. In M. Broy and S. Jähnichen (eds.): *Methods, Languages, and Tools for the Construction of Correct Software*, Lecture Notes in Computer Science (LNCS), no 1009, Springer-Verlag, 1995.

[FvH94] D. Fensel and F. van Harmelen: A Comparison of Languages which Operationalize and Formalize KADS Models of Expertise, *The Knowledge Engineering Review*, 9(2), 1994.

[Gol82] R. Goldblatt: *Axiomatising the Logic of Computer Science*, LNCS 130, Springer-Verlag, Berlin, 1982.

[Har84] D. Harel: Dynamic Logic. In D. Gabby et al. (eds.), *Handook of Philosophical Logic, vol. II, Extensions of Classical Logic*, D. Reidel Publishing Company, Dordrecht (NL), 1984.

[Neb96] B. Nebel: Artificial Intelligence: A Computational Perspective. In G. Brewka (ed.), *Essentials in Knowledge Representation*, Springer Verlag, 1996.

[PGT96] C. Pierret-Golbreich and X. Talon: An Algebraic Specification of the Dynamic Behaviour of Knowledge-Based Systems, *The Knowledge Engineering Review*, 11(2), 1996.

[Rei92] W. Reif: The KIV-System: Systematic Construction of Verified Software, *Proceedings of the 11th International Conference on Automated Deduction*, CADE-92, Lecture Notes in Computer Science (LNCS), no 607, Springer-Verlag, 1992.

[Rei95] W. Reif: The KIV Approach to Software Engineering. In M. Broy and S. Jähnichen (eds.): *Methods, Languages, and Tools for the Construction of Correct Software*, LNCS 1009, Springer-Verlag, 1995.

[ReS93] W. Reif and K. Stenzel: Reuse of Proofs in Software Verification. In Shyamasundar (ed.), *Foundation of Software Technology and Theoretical Computer Science*, LNCS 761, Springer-Verlag, 1993.

[Sha89] A. Shaw: Reasoning About Time in Higher Level Language Software, *IEEE Transactions on Software Engineering*, 15(7):875—889, 1989.

[SpV94] J. W. Spee L. in 't Veld: The Semantics of $K_{BS}SF$: A Language For KBS Design, *Knowledge Acquisition*, vol 6, 1994.

[SWA+94] A. TH. Schreiber, B. Wielinga, J. M. Akkermans, W. Van De Velde, and R. de Hoog: CommonKADS. A Comprehensive Methodology for KBS Development, *IEEE Expert*, 9(6):28—37, 1994.

[vdV88] W. van de Velde: Inference Structure as a Basis for Problem Solving. In *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI-88)*, Munich, August 1-5, 1988.

[vHA96] F. van Harmelen and M. Aben: Structure-preserving Specification Languages for Knowledge-based Systems, *Journal of Human Computer Studies*, 44:187—212, 1996.

[vHB92] F. van Harmelen and J. Balder: $(ML)^2$: A Formal Language for KADS Conceptual Models, *Knowledge Acquisition*, 4(1), 1992.

[Wir90] M. Wirsing: Algebraic Specification. In J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier Science Publ., 1990.

to find improved characterizations for these assumptions. This can be achieved by a precise analysis of their role in the completed proof for retracing unnecessary properties of them. Again, we can use KIV as tool support for this process. The main difference to the usual use of KIV for verification purpose is that we do not aim for a succeeded proof. Instead, already from the beginning we expect the proof not to succeed and we use instead the open goals found during the proof process as result for further consideration.

7 Related Work, Conclusions and Future Work

[vHA96], [BeA] also propose the use of conceptual models for the specification and verification of KBSs. [BeA] provide some hand-made proofs. The idea of using algebraic specifications KBSs is also used by [SpV94] and [PGT96]. However, they do not use dynamic logic and no actual proofs have been reported. That should no be taken as a surprise as realistic formal proofs can only be done with mechanical support. We have shown in the paper how tasks and problem-solving methods can be specified and verified with KIV. KIV is well-suited for both as it combines algebraic specifications with imperative constructs that enable the specification of the reasoning behaviour. The interactive theorem prover provides excellent support in proceeding the different automatically generated proof obligations. The modular concept of proofs and proof reuse for partial modified specification make the verification effort feasible. In the paper, we have shown several proofs that are necessary to establish a correct specification.

We also realized several promising lines of future work. The conceptual model used to specify knowledge-based systems can be expressed in the generic module concept of KIV. However, this is connected with a loss of information because the KIV specification does not distinguish the different roles that specifications may have (goals, requirements, adapters etc.). Therefore, not all of the desired proof obligations could be generated automatically or at least not directly. Still, it seems to be possible to specialize the generic concepts of KIV. This would allow to provide the automatic generation of according proof obligations and of predefined modules and specification combinations to model the different aspects of a knowledge-based systems. Based on this, we plan to develop a methodological framework for the stepwise development of correct specifications of knowledge-based systems.

Acknowledgement. We would like to thank Frank van Harmelen for encouraging us to write the paper, Annette ten Teije and two anonymous reviewers for helpful comments.

8 References

[AWS93] J. M. Akkermans, B. Wielinga, and A. Th. Schreiber: Steps in Constructing Problem-Solving Methods. In N. Aussenac et al. (eds.): *Knowledge-Acquisition for Knowledge-Based Systems*, Lecture Notes in AI, no 723, Springer-Verlag, 1993.

[BAT+91] T. Bylander, D. Allemang, M. C. Tanner, and J. R. Josephson: The Computational Complexity of Abduction, *Artificial Intelligence*, 49, 1991.

[BeA] R. Benjamins and M. Aben: Structure-Preserving KBS Development through Reusable Libraries: a Case Study in Diagnosis. To appear in International Journal on Human-Computer Studies.

[BvV94] J. Breuker and W. Van de Velde (eds.): *The CommonKADS Library for Expertise Modelling*, IOS Press, Amsterdam, The Netherlands, 1994.

[Cha86] B. Chandrasekaran: Generic Tasks in Knowledge-based Reasoning: High-level Building Blocks for Expert System Design. *IEEE Expert*, 1(3): 23—30, 1986.

[EST+95] H. Eriksson, Y. Shahar, S. W. Tu, A. R. Puerta, and M. A. Musen: Task Modeling with Reusable Problem-Solving Methods, *Artificial Intelligence*, 79(2):293—326, 1995.

[Fen95b] D. Fensel: *The Knowledge Acquisition and Representation Language KARL*, Kluwer Academic Publ., Boston, 1995.

[Fen95c] D. Fensel: Formal Specification Languages in Knowledge and Software Engineering, *The Knowledge Engineering Review*, 10(4), 1995.

[FeG96] D. Fensel and R. Groenboom: MLPM: Defing a Semantics and Axiomatization for Specifying the Reasoning Process of Knowledge-based Systems. In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI-96)*, Budapest, August 12-16, 1996.

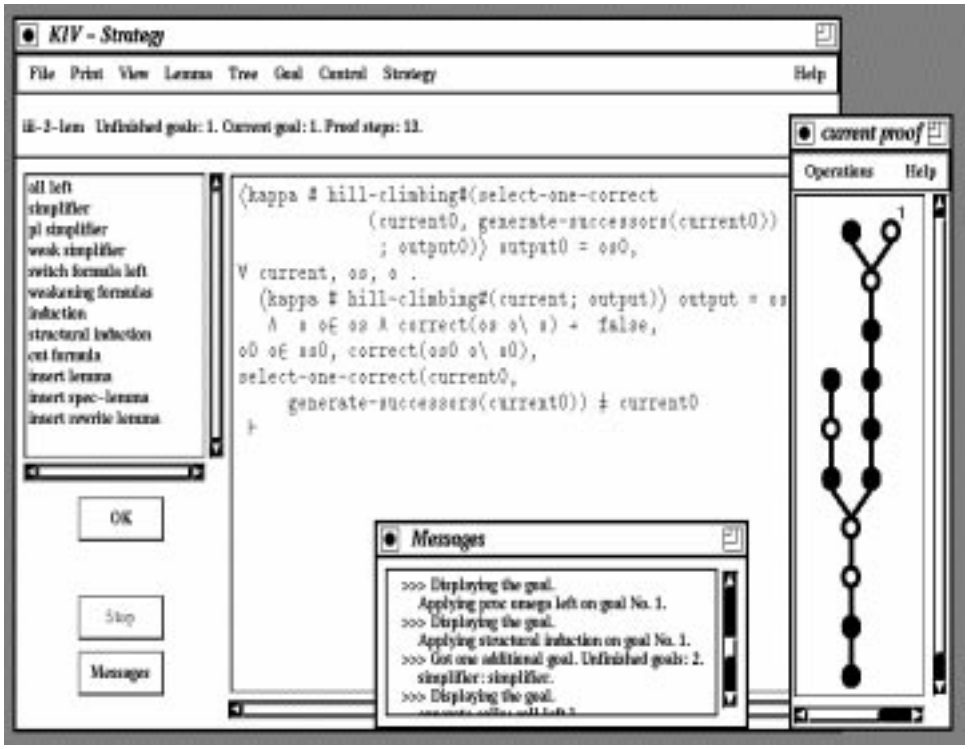

Fig. 4 Verifying the PSM with KIV.

that it is parsimonious in general. There may exist smaller subsets of it that are complete explanations. The adapter has to introduce a new requirement on domain knowledge or an assumption (in the case that it does not follow from the domain model) to guaranty, that the competence of the PSM is strong enough to achieve the goal of the task. The *monotony assumption* (cf. Figure 5) is sufficient (and necessary, cf. [FeS97]) to prove that the (global) parsimonious of the result of the PSM follows from its local parsimoniality. To ensure the automatic generation and management of this proof obligations by KIV we have to specify the adapter as a module that implements the task goal by importing the mappings and exporting the goal of the task (cf. Figure 5).

The question may arise how to provide such assumptions that close the gap between task definitions and PSM. In [FeS97], we present the idea of using mathematical proofs and analysis of their failure as a systematic means for forming assumptions. A mathematical proof that a PSM solves a given problem usually enforces the introduction of assumptions to close gaps in the line of reasoning of the proof. It can therefore be viewed as a search process for hidden assumptions. Gaps that can be found in a failed proof provide already first characterizations of these assumptions. Using an open goal of a proof directly as an assumption normally leads to very strong assumptions. That is, these assumptions are sufficient to guarantee the correctness of the proof, but they are often neither necessary for the proof nor realistic in the sense that application problems will fulfil them. Therefore, further work is necessary

assumptions = **enrich** *abduction problem* **with**
axioms
complete(*all-hypotheses*),
$H_1 \subseteq H_2 \rightarrow explain(H_1) \subseteq explain(H_2)$
end enrich

adapter = **module**
export *explanation*
refinement
representation of operations
explanation# **implements** *explanation*;
import *mapping*
variables *res* : *hypotheses*;
implementation
explanation#(**var** *res*)
begin
res := *local-parsimonious-explanation*
end

Fig. 5 Connecting PSM and Task.

This is equivalent to

$$\textit{select-one-correct}(O,\textit{generate-successors}(O)) \subseteq O$$

which can be proved as an instantiation of a stronger lemma which is used in the proof of iii-1-lemma.

iii-1-lemma. The proof is done by induction on the recursive calls of the *hill-climbing* procedure in a terminating run. The proof uses the property that

$$O \subseteq O_0 \rightarrow \textit{select-one-correct}(O,\textit{generate-successors}(O)) \subseteq O_0.$$

This property is proven by using the (three) axioms for the inferences *generate-successors* and *select-one-correct* and a suitable case-distinction (i.e., four interactions).

iii-2-lemma. The proof is done, as for lemma iii-1-lemma, by induction on the recursive calls of the *hill-climbing* procedure in a terminating run. For this it is enough to use the property of *select-one-correct* that it yields a correct object set, whenever it does not yield its first argument as result. This property follows immediately from the axioms.

iii-3-lemma. The proof is done, as for iii-1-lemma and iii-2-lemma, by induction on the recursive calls of the *hill-climbing* procedure in a terminating run. The proof uses the property that

$$\exists o.\ (o \in O \wedge \textit{correct}(O \setminus o)) \rightarrow \textit{select-one-correct}(O,\textit{generate-successors}(O)) \neq O.$$

This property is proven as follows: First we show that from the condition

$\exists o.\ (o \in O \wedge \textit{correct}(O \setminus o))$ follows that

$$O_1 := \textit{select-one-correct}(O,\textit{generate-successors}(O)) \in \textit{generate-successors}(O)$$

From the axiom for *generate successors* follows that there must exist an $o_1 \in O$ such that $O_1 = O \setminus o_1$, i.e. $O_1 \neq O$.

To give an impression of how to work with KIV, Figure 4 is a screen dump of the KIV system when proving iii-3-lemma. 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, which 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.

6 Adapter: Connecting Task and Problem-Solving Method

The description of an *adapter* maps the different terminologies of task definition, PSM, and domain model and introduces further requirements and assumptions that have to be made to relate the competence of a PSM with the functionality as it is introduced by the task definition. Because it relates the three other parts of a specification together and establishes their relationship in a way that meets the specific application problem they can be described independently and selected from libraries. Their consistent combination and their adaptation to the specific aspects of the given application must be provided by the adapter. Usually an adapter introduces new requirements or assumptions because in general, most problems tackled with KBSs are inherently complex and intractable (cf. [FeS96], [Neb96]). A PSM 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.

We have to introduce assumptions by the module *assumptions* (cf. Figure 5) to ensure that the competence of our method implies the goal of the task. First, we have to require that the input of the method is a complete explanation. Based on the mappings it is now simple to prove that the input requirement of the method is fulfilled (i.e., the input is correct). Second, based on the mappings we can prove that our method set-minimizer finds a local-minimal set that is parsimonious in the sense that each subset that contains one element less is not a complete explanation. However, we cannot guaranty

5 Proving Termination and Correctness of the PSM

We have to prove for the operational specification of the PSM termination can be guaranteed and the competence as specified. This proof obligations are automatically generated by KIV as formulas in dynamic logic (cf. [Gol82], [Har84]). In our example, it derives the following proof obligations:

(i-1) $\vdash \langle control\#(output)\rangle$ true, i.e. termination

(iii-1) $\vdash \langle control\#(output_0)\rangle\ output_0 \subseteq input$, corresponds to axiom 1 of the competence

(iii-2) $\vdash \langle control\#(output_0)\rangle\ correct(output_0)$, corresponds to axiom 2 of the competence

(iii-3) $\vdash \langle control\#(output_0)\rangle\ o \in output_0 \rightarrow \neg \langle control\#(output_0)\rangle correct(output_0 \setminus o)$, corresponds to axioms 3 of the competence.

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 done by the machine. Automation is achieved by rewriting and by heuristics which 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 so far constructed partial proof. Most of these user interactions can be done by selecting alternatives provided by a menu.

For each of the proof obligations we formulate straightforward auxiliary lemmas i-1-lemma, iii-1-lemma, iii-2-lemma, and iii-3-lemma; one for each of these proof obligations, respectively. These auxiliary lemmas express the corresponding property of the *hill-climbing* sub-procedure (cf. Figure 3):

(i-1-lemma) $\vdash \langle hill\text{-}climbing\#(current;output)\rangle$ true

(iii-1-lemma) $current \subseteq O \vdash \langle hill\text{-}climbing\#(current;output)\rangle\ output \subseteq O$

(iii-2-lemma) $correct(current) \vdash \langle hill\text{-}climbing\#(current;output)\rangle\ correct(output)$

(iii-3-lemma) $\langle hill\text{-}climbing\#(current;output)\rangle\ output = O \vdash \neg \exists o.\ o \in O \wedge correct(O \setminus o)$

Using these lemmas, each of the proof obligations can now directly be proven with the interactive proof environment of KIV. Activating the standard set of predefined heuristics (by click) and then selecting the auxiliary lemma to use (by click) is enough. KIV automatically does the unfolding of the control procedure, finds the appropriate instantiation of the lemma, and carries out the first-order reasoning (necessary e.g. for (iii-3)). Thus each of the proofs of (1-1), (iii-1), (iii-2), (iii-3) can be done with one user interaction, respectively.

It remains to prove the four lemmas. All of these proofs work by induction. And for constructing them with the help of KIV one has to tell KIV (again by clicking) which kind of induction should be used. KIV is then able to unfold (and symbolically execute) the procedure hill-climbing and find the correct instantiation of the induction hypothesis. While KIV tries to construct the proofs it comes up with subgoals reflecting certain properties of the inference actions. We then interact by formulating these properties as first-order lemmas in the specification of the inferences, and KIV is able to automatically find and use them to close the open subgoals. Thus, again with quite little, and almost straight-forward user interaction (besides the formulation of the lemmas) the original proof obligations are reduced to the task of proving some properties of the inferences stated in first-order logic. These in turn can be derived from the axioms. Here again some user interaction is required, mostly selecting the appropriate axioms (and also one quantifier instantiation). Besides this KIV does all the first-order reasoning. We now give a sketch of the proofs:

i-1-lemma. The termination of the PSM is proven by induction on the first parameter of *hill-climbing*, where the (well-founded[5]) order $\subset$ is used. In the induction step we use the fact that

$$select\text{-}one\text{-}correct(O,generate\text{-}successors(O)) \neq O \rightarrow select\text{-}one\text{-}correct(O,generate\text{-}successors(O) \subset O.$$

5. Remember that we deal with finite sets only.

Specifications

psm-domain requirements = **enrich** *objects* **with**
constants *input* : *objects*;
predicates *correct* : *objects*,
axioms
correct(*input*)
end enrich

inferences = **enrich** *object-sets* **with**
functions
generate-successors : *objects* → *object-sets*,
select-one-correct : *objects* x *object-sets* → *objects;*
axioms
$O_2 \in$ *generate-successors*$(O_1) \leftrightarrow$
$\exists o_1 . (o_1 \in O_1 \wedge O_2 = O_1 \setminus o_1)$,
$(\exists O_1 . (O_1 \in OS \wedge$ *correct*$(O_1)) \rightarrow$
select-one-correct$(O,OS) \in OS \wedge$
correct(*select-one-correct*(O,OS)),
$\neg\exists O_1 . (O_1 \in OS \wedge$ *correct*$(O_1))$
→ *select-one-correct*$(O,OS) = O$
end enrich

competence = **generic specification**
parameter *PSM-domain requirements* **target**
constants *local-minimal-set* : *objects*;
axioms
(1) *local-minimal-set* $\subseteq$ *input*,
(2) *correct*(*local-minimal-set*),
(3) $o_1 \in$ *local-minimal-set*
→ ¬*correct*(*local-minimal-set* $\setminus o_1$)
end generic specification

Modules

control = **module**
export *competence*
refinement
representation of operations
control# **implements** *local-minimal-set*
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-one-correct
(current,*generate-successors*(*current*)) **in**
if *new = current*
then *output* := *current*
else *hill-climbing#*(*new*,*output*)
end

Fig. 3 The sub-specifications and modules of set-minimizer.

provided in Figure 1 and the definition of some of its specifications and modules as given in Figure 3.

Domain Requirements. The main requirements on available knowledge and input that are introduced by the method are: the existence of a possible set of *objects* (a sort), the existence of a predicate *correct* holding true for some sets, and the method finally assumes that the *input* is a correct set. These requirements on knowledge and input data are specified as (formal) parameter of the specification of the method. They get replaced by concrete parameters when the method is applied for a specific task and domain as we will see in section 6.[4]

Operational Specification. The method works as follows: First, we take the input. Then we recursively generate the successors of the current set and select one of its correct successors. If there is no new correct successor we return the current set. The functions *generate-successors* and *select-one-correct* in the specification *inferences* correspond to elementary inference actions in CommonKADS [SWA+94]. The procedural control (in KADS located at the task body) is defined by the module *control*.

Competence. The *competence* in Figure 3 states that *set-minimizer* is able to find a *local* minimal subset of the given set of objects. The three axioms state that it (1) finds a subset that is correct (2), and minimal (3), i.e. that each set containing one element less is not a correct set.

4. This parameterization also allows to get different variants of the competence of a method by varying its knowledge requirement.

to include this directly in the KIV tool environment.

3 Formalizing a Task

The description of a *task* consists of two parts (cf. [FeG97]): 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. We use a simple task to illustrate the formalization of our approach. The task *abductive diagnosis* receives a set of observations as input and delivers a complete and parsimonious explanation (see e.g. [BAT+91]). An explanation is a set of hypotheses. A *complete explanation* must explain all input data (i.e., *observations*) and a *parsimonious* explanation must be minimal (that is, no subset of it has the same or a greater explanatory power). Figure 1 provides the modular structure of the task definition for our example. The internal definitions of some of the specifications are given in Figure 2. The specification *abduction problem* is an enrichment of *data* and *hypothesis*[3] and introduces a requirement on domain knowledge. A function *explain* must be provided to relate hypotheses with observations they explain. Further on, the two predicates *complete* and *parsimonious* are introduced that are required to define a solution of the task. Based on these definition, we can finally define what an *explanation* must fulfil. It must be complete and parsimonious.

4 Formalizing a Problem-Solving Method

The concept PSM is present in a large part of current knowledge-engineering frameworks (e.g. Generic Tasks [Cha86], CommonKADS [BvV94], [SWA+94], Method-to-task approach [EST+95]). In general, PSMs are used to describe the reasoning process of a KBS. Besides some differences between the approaches, there is strong consensus that a PSM decomposes the entire reasoning task into more elementary inferences; defines the types of knowledge that are needed by the inference steps to be done; and defines control and knowledge flow between the inferences. In addition, [vdV88] and [AWS93] define the *competence* of a PSM independent from the specification of its operational reasoning behaviour. Proving that a PSM has some competence has the clear advantage that the selection of a method for a given problem and the verification whether a PSM fulfils its task can be done independently from details of the internal reasoning behaviour of the method. The third element of a PSM are *requirements* on domain knowledge. Each inference step and therefore the competence description of a PSM requires specific types of domain knowledge. These complex requirements on domain knowledge distinguish a PSM from usual software products. Pre-conditions on valid inputs are extended to complex requirements on available domain knowledge.

We use the very simple PSM *set-minimizer* of [FeG97] for our example. It receives a set of objects as input and tries to find a minimized version of the set that still fulfils a correctness requirement. The applied search strategy is one-step look ahead. The overall structure of the PSM-specification is

dabduction problem = **enrich** *hypotheses*, *data* **with**
functions *explain* : *hypotheses* → *data;*
predicates
complete : *hypotheses*,
parsimonious : *hypotheses;*
axioms
$complete(H) \leftrightarrow explain(H) = all\text{-}data$,
$parsimonious(H) \leftrightarrow$
$\neg \exists H_1 . H_1 \subset H \wedge explain(H) \subseteq explain(H_1)$
end enrich

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

Fig. 2 The specification of the abductive task.

3. *Data* and *hypothesis* specify finite sets of data and hypothesis, respectively. Their specification is skipped for space limitation.

2 The Structure of the Entire Specification

In KIV, the entire specification of a system can be split into smaller and more tractable pieces. Each *elementary specification* introduces a signature and a set of axioms. The semantics of such a specification is the class of all algebras that satisfy the first-order axioms (i.e., *loose* semantics is applied [Wir90]). KIV provides several mechanisms to combine elementary specifications to more complex specifications: *sum*, *enrichment*, *renaming*, and *actualization* of *parameterized* specifications (cf. [Rei95]). 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.

Figure 1 provides the structure of the entire specification of our example. The *development graph* provides an overview of the overall structure, i.e., the dependencies between the (sub-)specifications and implementations. The single specifications (the rectangles in the graph) and modules (the rhomboid units in the graph) are discussed during the following sections. The development graph of KIV does not directly reflect the conceptual units we identified above (i.e., task, PSM, domain model, and adapter). However they can be defined by hand as aggregation of elements of the graph and current work is done

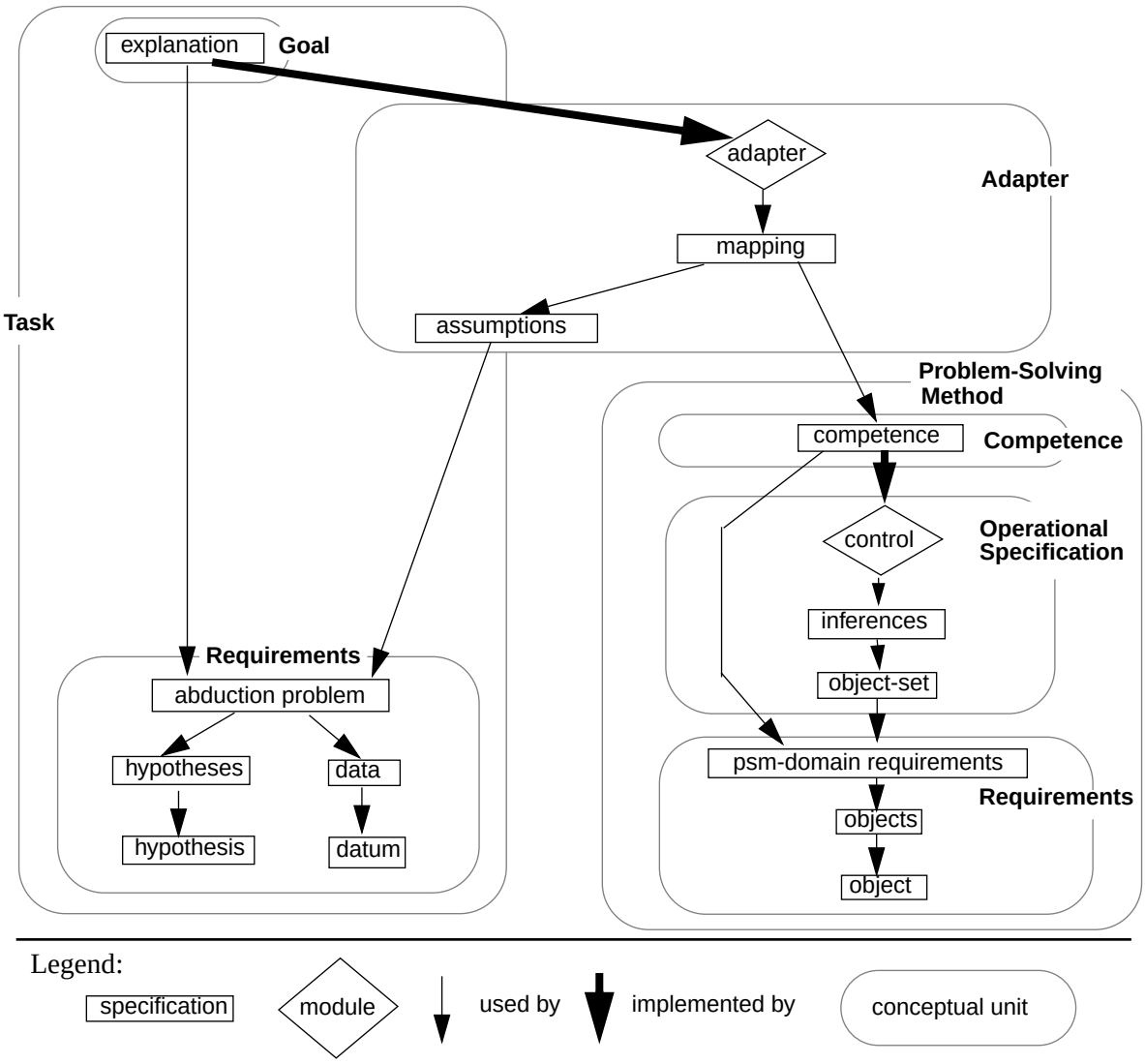

Fig. 1 The development graph in KIV.

2. An extended version of the paper and containing all proofs is available via http://www.aifb.uni-karlsruhe.de/WBS/dfe/eurovav.html.

framework for describing a KBS consists of four elements: a *task* that defines the problem that should be solved by the KBS, a *problem-solving method* (PSM) that defines the reasoning process of a KBS; a *domain model* that describes the domain knowledge of the KBS. Each of these three elements are described independently to enable the reuse of task descriptions in different domains, the reuse of PSMs for different tasks and domains, and the reuse of domain knowledge for different tasks and PSMs. 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.

In this paper, we discuss the specification and verification of the different elements and their relationships. We use the KIV system (Karlsruhe Interactive Verifier) (see [Rei95]) for both activities. It is an advanced tool for the construction of provably correct software. 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. [FRS+95]). The use of the KIV system for the verification of KBSs is quite attractive. KIV supports dynamic logic (cf. [Gol82], [Har84]) which has been proved useful in specification of KBSs (cf. KARL [Fen95b], $(ML)^2$ [vHB92], and MLPM [FeG96]). Dynamic logic has two main advantages (especially if compared to first-order predicate logic). First, dynamic logic is quite expressive, e.g. one can formalize and prove termination or equivalence of programs or generatedness of data types.[1] 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 humans and provide more structural information which can be employed by proof heuristics.

KIV allows structuring of specifications and modularisation of software systems. Therefore, the conceptual model of our specification can be realized by the modular structure of a specification in KIV. 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. With the interactive theorem prover even complicated proofs can be constructed. A high degree of automation can be achieved by a number of implemented heuristics. However, interaction is necessary because for two reasons: In general, complex proofs cannot be completely automated and proving usually means to find error either in the specification or in the implementation. The proof process is therefore a kind of search process for errors. Analysis of failed proof attempts and the automatic generation of counterexamples support the iterative process of developing correct specifications and programs. Finally, an elaborated correctness management keeps track of lemma dependencies (and their modifications) and automatic reuse of proofs allows an incremental verification of corrected versions of programs and lemmas (see [ReS93]). Both aspects are essential to make verification feasible given the fact that system development is a process of steady modification and revision.

During the paper we illustrate some of the specification elements and proof processes necessary to establish the correctness of the different elements of a complete specification. In each section, we use different aspects of a running example for illustrating these processes. In section 2, we introduce the structure of our specification in KIV. In section 3, we illustrate the specification of a task. In section 4, we present the specification of a problem-solving method. Its termination and correctness proofs are provided in section 5. During section 6, we illustrate how the appropriate relationship between task and problem-solving method becomes established. Section 7 summarizes the paper and defines objectives for future research. For the sake of space limitation we discuss only the verification of the PSM. We cannot present the proofs of the adapter and the specification and verification of a domain model. In general, we would have to prove that the domain knowledge is consistent and that it fulfils the requirements of PSM, task, and adapter.[2]

---

1. Of course, due to its expressive power any effective calculus for dynamic logic has to be incomplete. Fortunately, this does not limit the practical applications (because the incompleteness stems from self reference). However, as in first-order logic, fully automatic construction of proofs is in general not feasible due to the enormous size of the search space.

Specifying and Verifying Knowledge-Based Systems with KIV

In *Proceedings of the European*
Symposium on the Validation and Verification of Knowledge Based Systems EUROVAV-97, Leuven
Belgium, June 26-28, 1997.

Dieter Fensel[1] and Arno Schönegge[2]

[1] University of Karlsruhe, Institute AIFB, 76128 Karlsruhe, Germany. E-mail: dieter.fensel@aifb.uni-karlsruhe.de
[2] University of Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, 76128 Karlsruhe, Germany.
E-mail: schoenegge@ira.uka.de

Abstract. We discuss the use of the Karlsruhe Interactive Verifier (KIV) for the verification of conceptual and formal specifications of knowledge-based systems. KIV was originally developed for the verification of procedural programs but it fits well for verifying knowledge-based systems. Its specification language is based on algebraic specification means 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. Only through this support, verification of complex specifications becomes possible. We provide some examples on how to specify and verify tasks, problem-solving methods, and their relationships.

Keywords. Knowledge-based systems, knowledge-level modelling, formal specification, verification, interactive theorem proving

1 INTRODUCTION

During the last years, several conceptual and formal specification techniques for knowledge-based systems (KBSs) have been developed (see [FvH94], [Fen95c] for surveys). The advantage of these modelling or specification techniques is that they enable the description of a KBS independent of its implementation. This has several advantages. First, such a specification can be used as golden standard for the validation and verification of the implementation of the KBS. It defines the requirements the implementation must fulfil. Second, validation and verification of the functionality, the reasoning behavior, and the domain knowledge of a KBS is already possible during the early phases of the development process of the KBS. A model of the KBS can be investigated independently of aspects that are only related to its implementation. Especially if a KBS is built up from reusable components it becomes an essential task to verify whether the assumptions of such a reusable building block fit to each other and the specific circumstances of the actual problem and knowledge. Third, integrating the formal specification into a conceptual model supports understandability of specification as well as verification (cf. [vHA96], [BeA]).

In [FeG97], we presented a conceptual and formal framework for the specification of KBSs based on different reusable elements. The conceptual framework is developed in accordance to the CommonKADS model of expertise (see [SWA+94]) which has become widely used by the knowledge engineering community. As a consequence of our modularized specification, we identify several proof obligations that arise in order to guarantee a consistent specification. The overall verification of a KBS is broken down into different types of proof obligations that ensure that the different elements of a specification together define a consistent system. Thus a separation of concerns is achieved that contributes to the feasibility of the verification. The conceptual model applied to describe KBSs is used to brake the general proof obligations into smaller pieces and makes parts of them reusable. Our