In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering (ASEC-97)*, Incline Village, Nevada, November 3-5, 1997.

Using KIV to Specify and Verify Architectures of Knowledge-Based Systems

Dieter Fensel
University of Karlsruhe, Institute AIFB,
76128 Karlsruhe, Germany
dieter.fensel@aifb.uni-karsruhe.de

Arno Schönegge
University of Karlsruhe, Institute LKD,
76128 Karlsruhe, Germany
schoenegge@ira.uka.de

Abstract

Building knowledge-based systems from reusable elements is a key factor in developing them economically. However, one has to ensure that the assumptions and functionality of the reused building block fit together with each other and the specific circumstances of the actual problem and knowledge. We use the Karlsruhe Interactive Verifier (KIV) for this purpose. We show how the verification of conceptual and formal specifications of knowledge-based systems can be performed with it. KIV was originally developed for the verification of procedural programs but it serves well for verifying knowledge-based systems. 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. We provide some examples on how to specify and verify tasks, problem-solving methods, and their relationships.

1. Introduction

During the last few years, several conceptual [26] and formal specification techniques [8] for knowledge-based systems (KBSs) have been developed. These modelling or specification techniques enable the description of a KBS independent of its implementation. These approaches provide three essential contributions for enhancing the development process of KBSs (cf. [30]):

- First, the functionality and the knowledge of a KBS can be modelled and evaluated independent from implementation issues. Validation and verification of the functionality, the reasoning behavior, and the domain knowledge of a KBS is already possible in the early phases of the development process of the KBS.
- Second, identifying different modelling aspects and knowledge types involved in a KBS enables us to structure the development process as well as its intermediate and final results.
- Third, abstracting implementation- and application-specific circumstances enables the identification of reusable architectures, components, and knowledge bases that enhances quality and efficiency of the development process.

When put in a broader context such approaches correspond to recent efforts on software *architectures* [27] that provide patterns for breaking down an entire functionality into a number of components with predefined interactions. In [10], 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 [26] 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. Each of these elements is 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) [24] 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. [13]). The use of the KIV system for the verification of KBSs is quite attractive. KIV supports dynamic logic (cf. [14], [15]) which has been proved useful in the specification of KBSs (cf. KARL [7], $(ML)^2$ [31], and MLPM [9]). 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.[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 people 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. 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. The proof process is therefore a kind of search process for errors. Analysis of failed proof attempts and the automatic generation of counter examples support the iterative process of developing correct specifications and programs. 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 [25]). Both aspects are essential to make verification feasible given the fact that system development is a process of steady modification and revision.

In this 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. In 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 only discuss the verification of the PSM and the adapter and not the specification and verification of a domain model. In general, we would have to show that the domain knowledge is consistent and that it fulfils the requirements of PSM, task, and adapter.

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, the fully automatic construction of proofs is in general not feasible due to the enormous size of the search space.

2. The Architecture

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 [32]).

KIV provides mechanisms for combining elementary specifications to more complex ones (e.g. *sum*, *enrichment*, *renaming*, and *actualization* of parameterized specifications) which are common to most algebraic specification languages, cf. [33].

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, i.e. the dependencies between the (sub-)specifications and implementations in KIV. The single specifications (the rectangles in the graph) and modules (the rhomboid units in the graph) are discussed in the following sections. The conceptual units we identified above (i.e., task, PSM, domain model, and adapter) can be defined by hand as aggregation of elements of the development graph and current work is being done to include this directly in the tool environment.

3. Formalizing a Task

The description of a *task* consists of two parts (cf. [10]): 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. In the following, we define a simple diagnostic problem independent of the domain (e.g. medical diagnosis, mechanical diagnosis, diagnosis of electrical devices etc.).

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. [3]). 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, none of its subsets have the same or a greater explanatory power). Figure 1 provides the modular structure of the task definition of 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*[2] and introduces a requirement on domain knowledge. A function *explain* relates hypotheses with observations they explain. Further, 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.

2. *Data* and *hypothesis* specify finite sets of data and hypothesis, respectively. Their specification is omitted due to space limitation.

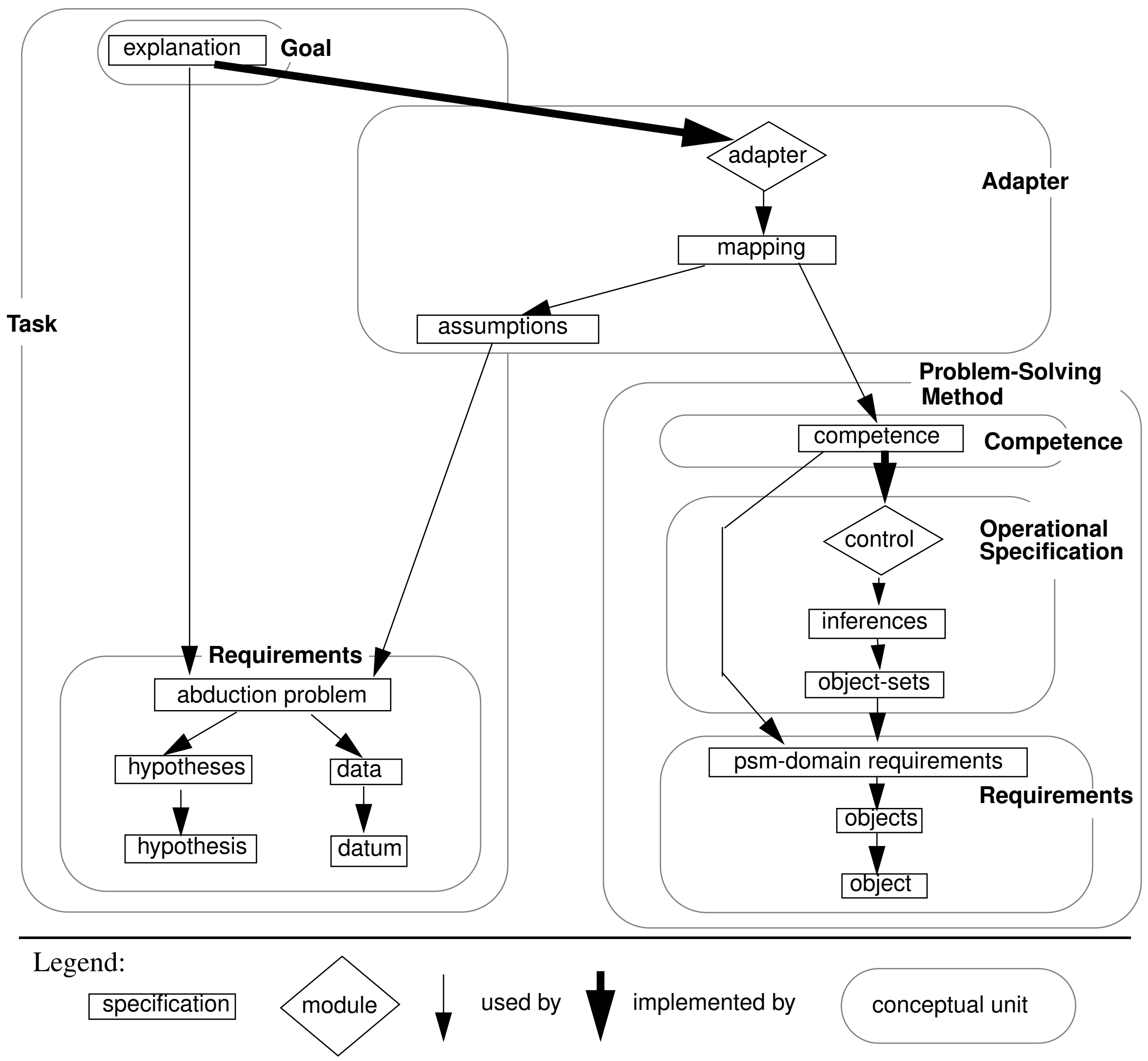

Fig. 1 The development graph in KIV.

4. Formalizing a Problem-Solving Method

The concept PSM is present in many current knowledge-engineering frameworks (e.g. Generic Tasks [4], CommonKADS [26], Method-to-task approach [6]). In general, PSMs are used to describe the reasoning process of a KBS. Aside from some differences between the approaches, there is a 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.

Extending, [1] defined the *competence* of a PSM (i.e., a functional black-box specification) 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.

Finally, a PSM has *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. Preconditions on valid inputs are extended to complex requirements on available domain knowledge.

Libraries of PSMs are described in [2], [4], and [23]. *Reusing* PSMs enhance the development process of KBSs. They can either be directly reused as reasoning component or in the case of a more complex task they can be used to decompose the task. In the latter case, each inference step of the PSM defines a new task. Each of these subtasks require again the selection of a PSM that may either directly solve it or recursively refine it to new subtasks. In the first case, a PSM is reused as a component (black-box reuse of PSM) whereas in the second case, the PSM defines an architecture (white-box reuse) for selecting further components.[3]

abduction problem = **enrich** *hypotheses*, *data* **with**
  functions *explain* : *hypotheses* $\rightarrow$ *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.

We use the very simple PSM *set-minimizer* of [10] 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 search strategy applied is one-step look ahead. The overall structure of the PSM-specification is provided in Figure 1 and the definition of some of its subspecifications and modules is given in Figure 3.

4.1. 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, finally, the method 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 are replaced by concrete parameters when the method is applied for a specific task and domain as we will see in Section 6.[4]

4.2. 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 [26]. The procedural control (in KADS located at the task body) is defined by the module *control*.

4.3. 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 less element is not a correct set.

---

3. An alternative way to the architectural point of view is to view PSMs that decompose complex tasks as a meta-object which obtains specifications as input and transformed specifications as output. Then, they correspond to design operators of KIDS [28].

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

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

object-sets = **generic specification**
parameter *psm-domain requirements* **target**
constants $\emptyset$: *object-sets*;
functions *os-insert* : *objects* x *object-sets* → *object-sets;*
predicates . ∈ . : *objects* x *object-sets;*
axioms
object-sets **generated by** $\emptyset_{os}$, *os-insert* ,
$\neg\ OS \in_{os} \emptyset_{os}$,
$o_1 \in$ *os-insert*$(o_2,O) \leftrightarrow o_1 = o_2 \vee o_1 \in_o O$,
$O_1 = O_2 \leftrightarrow (\forall\ o_1 .\ o_1 \in O_1 \leftrightarrow o_1 \in O_2)$
end generic specification

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))),
$\neg\exists O_1 .\ (O_1 \in OS \wedge correct(O_1))$
$\rightarrow$ *select-one-correct*$(O,OS) = O$
end enrich

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*))
if *new* = *current*
then *output* **:=** *current*
else *hill-climbing*(*new*,*output*)
end

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

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

5. Proving Total Correctness of the PSM

When introducing a PSM into a library we have to prove two aspects of the operational specification of the PSM. We have to ensure the termination of the procedure and the competence as specified. When reusing the PSM this proof need not to be repeated and can (implicitly) be reused.

The proof obligations are automatically generated by KIV as formulas in dynamic logic (cf. [14], [15]). In our example, it derives the following proof obligations (see [24], section 5.2 for more details on how the correctness of a module is translated into a set of proof obligations formulated in dynamic logic.):

(i) <*control*(*output*)> true, i.e. termination
(ii) <*control*(*output*)> *output* ⊆ *input*,
corresponds to axiom 1 of the competence
(iii) <*control*(*output*)> *correct*(*output*),
corresponds to axiom 2 of the competence
(iv) <*control*(*output*)> *o* ∈ *output*
→ ¬ <*control*(*output*)>*correct*(*output* \ *o*),
corresponds to axioms 3 of the competence.

These proof obligations ensure that the PSM terminates and that it terminates in a state that respects the axioms used to characterize the competence of the PSM (“<>“ is the diamond operator of dynamic logic).

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 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 partial proof constructed so far. Most of these user interactions can be performed by selecting alternatives provided by a menu.

For each of the proof obligations we formulate straightforward auxiliary lemmas i-lemma, ii-lemma, iii-lemma, and iv-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-lemma) $\langle$*hill-climbing*(*current*;*output*)$\rangle$ true

(ii-lemma) $current \subseteq O$
$\rightarrow \langle$*hill-climbing*(*current*;*output*)$\rangle$ $output \subseteq O$

(iii-lemma) *correct*(*current*)
$\rightarrow \langle$*hill-climbing*(*current*;*output*)$\rangle$ *correct*(*output*)

(iv-lemma)
$\langle$*hill-climbing*(*current*;*output*)$\rangle$ $output = O$
$\rightarrow \neg \exists o.\ o \in O \wedge correct(O \setminus o)$

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 proper (by click) is enough. KIV automatically unfolds the control procedure, finds the appropriate instantiation of the lemma, and carries out the first-order reasoning (necessary e.g. for (iii)). Thus the proofs of (i), (ii), (iii), (iv) respectively can be carried out with one user interaction.

It remains to prove the four lemmas. All of these proofs work by induction. And to construct 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 with a few and almost straightforward user interactions (in addition to 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-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
$select\text{-}one\text{-}correct(O, generate\text{-}successors(O)) \neq O \rightarrow$
$select\text{-}one\text{-}correct(O, generate\text{-}successors(O)) \subset O$.
This is equivalent to
$select\text{-}one\text{-}correct(O, generate\text{-}successors(O)) \subseteq O$
which can be proved as an instantiation of a stronger lemma which is used in the proof of ii-lemma.

ii-lemma. The proof is carried out by induction on the recursive calls of the *hill-climbing* procedure in a terminating run. The proof uses the property that
$O \subseteq O_0 \rightarrow$
$select\text{-}one\text{-}correct(O, generate\text{-}successors(O)) \subseteq O_0$.
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-lemma. The proof is carried out, as for lemma ii-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.

iv-lemma. The proof is carried out, as for ii-lemma and iii-lemma, by induction on the recursive calls of the *hill-climbing* procedure in a terminating run. The proof uses the property that
$\exists o.\ (o \in O \wedge correct(O \setminus o)) \rightarrow$
$select\text{-}one\text{-}correct(O, generate\text{-}successors(O)) \neq O$.
This property is proven as follows: First we show that from the condition
$\exists o.\ (o \in O \wedge correct(O \setminus o))$ it follows that
$O_1 := select\text{-}one\text{-}correct(O, generate\text{-}successors(O))$
$\in generate\text{-}successors(O)$

From the axiom for *generate successors* follows that an $o_1 \in O$ must exist 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 iv-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.

5. Remember that we deal with finite sets only.

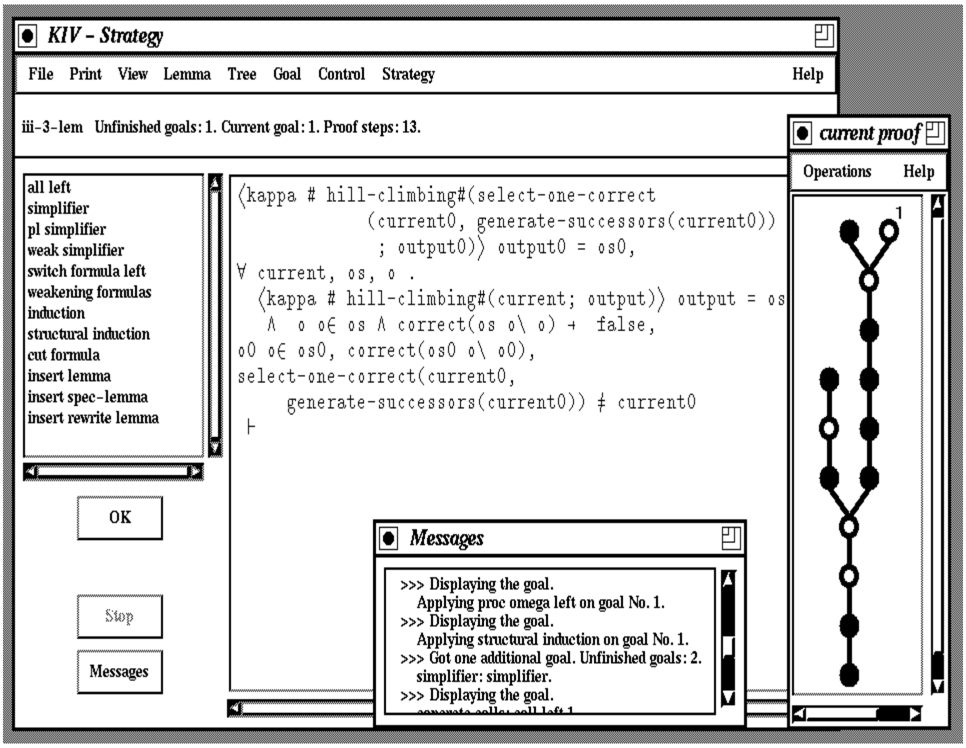

Fig. 4 Verifying the PSM with KIV.

6. Adapter: Connecting Task and PSM

The description of an *adapter* maps the different terminologies of the task definition, the PSM, and the 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. [11], [21]). 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 subspecification *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 easy 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 guarantee that it is parsimonious in general. Smaller subsets of it that are complete explanations may exist. 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 guarantee, 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. [12]) to prove that the (global) parsimoniousness of the result of the PSM follows from its local parsimoniousness. 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).

KIV automatically generates three proof obligations for the *adapter* module, again formulated in dynamic logic.

(i) <*explanation*(*res*)> true

(ii) <*explanation*(*res*)> *complete*(*res*)

(iii) <*explanation*(*res*)> *parsimonious*(*res*)

The proofs of i and ii are trivial and fully automatic. iii is the real proof obligation of this module. The informal proof sketch is as follows: We unfold the definition of parsimonious and it remains to prove that there is no proper subset H of *local-parsimonious-explanation* which explains (at least) all the data explained by the *local-parsimonious explanation*, i.e.,

explain(*local-parsimonious-explanation*) $\subseteq$ *explain*(H).

We choose some hypothesis $h \in$ *local-parsimonious-explanation*, such that

$H \subseteq$ *local-parsimonious-explanation* \ h.

Due to the monotony assumption we can derive

explain(H) $\subseteq$
explain(*local-parsimonious-explanation* \ h)

and transitivity of $\subseteq$ yields

explain(*local-parsimonious-explanation*) $\subseteq$
explain(*local-parsimonious-explanation* \ h).

Since *local-parsimonious-explanation* is complete (see ii) it holds

explain(*local-parsimonious-explanation*) = *all-data*

and thus

explain(*local-parsimonious-explanation* \ h) = *all-data*

that is *local-parsimonious-explanation* \ h is a complete set of hypotheses. This, however, contradicts the minimality axiom of the (mapped) competence. The proof in KIV requires 14 proof steps and 7 interactions. The seven interactions concern the application of axioms of the different specifications for the proof process.

The monotony assumption defines a natural subclass of abduction. For example [17] examine their role in model-based diagnosis. The assumption holds for applications, where no knowledge that constrains fault behaviour of devices is provided or where this knowledge respects the *limited-knowledge-of-abnormal behaviour assumption*. This is used by [16] as a *minimal diagnosis hypothesis* to reduce the average-case effort of finding all parsimonious and complete explanations with GDE.

The question of how to provide such assumptions that close the gap between task definitions and PSMs may arise. In [12], we presented the idea of *inverse verification*. We start a proof with KIV that the competence of the PSM 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 PSM and the task. When applying the *interactive* theorem prover to an impossible proof it returns a open goal that cannot be proven but which 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 (see [12] for more details).[6]

6. For example, the generation of counterexamples helps to find the essential aspects of an assumptions.

7. Related Work, Conclusions and Future Work

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 processing the different automatically generated proof obligations. The modular concept of proofs and proof reuse for partial modified specification make the verification effort feasible.

As a consequence of our modularized specification, we distinguish several proof obligations that arise in order to guarantee a consistent specification. Thus a separation of concerns is achieved that contributes to the feasibility of the verification. In addition, the proofs of the internal correctness of the components need not to be repeated when a component is reused. Only the proofs that are concerned with the proper combination of them have to be proceeded when developing a KBS.

The examples and proofs in our paper are kept simple. However, we have applied KIV also in more complex

```
assumptions = enrich abduction problem with
    axioms
        complete(all-hypotheses),
        H1 ⊆ H2 → explain(H1) ⊆ explain(H2)
end enrich

mapping =
actualize competence with assumptions by morphism
    correct → complete, input → all-hypotheses,
    local-minimal-set →
    local-parsimonious-explanation,
    objects → hypotheses,
    all-objects → all-hypotheses,
    ...
end actualize

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.

examples. E.g. we have verified with KIV theorems of [3] used to distinguish different complexity classes in abduction.[7] Notice that these proofs are in no way trivial: the informal proof for theorem 5.3 of [3] took one page. It states that for the class of ordered monotonic abduction problems using a specific preference criterion, there is a polynomial algorithm for finding a best explanation. This is proved by presenting an algorithm that in polynomial time returns such a best explanation. Formalizing the informal arguments of the correctness proof for this algorithm results in several hundred (machine checkable) proof steps.

Roughly, the interactive theorem proving system of KIV is comparable with systems as PVS [5] and Isabelle [22]. For our purpose, the KIV system is especially well suited due to its facilities for structuring specifications and software modules (including automatic generation of proof obligations), its proof engineering facilities (like an elaborated graphical user interface and reuse mechanisms), and the underlying dynamic logic.

[20] identifies two kinds of approaches in software reuse: Supporting the software development process with reusable components or making parts of the development process reusable via program transformation techniques. Our approach provide support by formally specified and verified building blocks i.e. components. The latter approach is taken by KIDS/SPECWARE [28], [29] which provides support in the derivation of efficient implementations from formal specifications. Here, problem-solving methods are not “first-order citizens“ that describe reusable components or architectures but second-order transformation rules working on specifications. As in our approach, system development is viewed as a semiautomatic activity. At the technical level, the main differences are the use of dynamic logic for the declarative specification of procedural constructs in KIV and the use of category theory and sheaf theory to express transformations of algebraic specifications in SPECWARE.

AMPHION ([18], [19]) is a knowledge-based software engineering system for the formal specification and automatic deductive synthesis of programs which consist of calls of subroutines from a library. It is specialized to application domains by means of a declarative domain theory and a library of subroutines. This specialization allows the automatic synthesis of programs from specifications. Our approach is more general-purpose (but, of course, less automatic): the programs developed are combinations and instantiations of (mostly domain-independent) problem-solving methods rather then simply a sequence of calls of subroutines from a library. Furthermore the (normal) user of the AMPHION system is not intended to create or modify the domain theory or the subroutine library. In our approach, the verification of user-defined problem-solving methods in a library with respect to their declarative specifications (competence) is done within the KIV system itself.

From a modelling point of view the main differences to the mentioned approaches stems from the fact that we specialise our approach to a specific type of systems (i.e., knowledge-based systems) which allow us to introduce strong assumptions on the architecture of the system under development. In consequence, we are able to provide stronger conceptual guidance for system development. Also, none of the approaches in software engineering make the distinction between a problem type (called a task in our framework) and a domain. In consequence, reusability is limited in their approaches.

From a technical point of view the main difference to the mentioned approaches stems from the fact that KIV uses dynamic logic, which enables the integrated specification and verification of declarative and procedural parts of a system. Part of this support is the automatic generation of proof obligations that guarantee the proper relationships between declarative and procedural parts as well as composed specifications in general.

Finally, we would like to mention some lines of our future work. The architecture 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 possible to specialize the generic concepts of KIV. This would allow us 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. Here we have to take a look on approaches like KIDS/Specware, especially for the process of refining a task via a problem-solving method into subtasks.

Acknowledgement. We would like to thank Rix Groenboom, John Penix, Annette ten Teije, Frank van Harmelen, Bob Wielinga, and the anonymous reviewers for very helpful comments and discussions and Jeff Butler for correcting the English.

7. Theorem 4.4 and (the more difficult) theorem 5.3. For both theorems only the total correctness of the algorithms and, of course, not their complexity bounds have been proven.

8. References

[1] 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.

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

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

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

[5] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas: A Tutorial Introduction to PVS. In *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques (WIFT '95)*, Boca Raton, Florida, April 1995.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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 1997.

[13] 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.

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

[15] 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.

[16] J. de Kleer and B. C. Williams: Diagnosing Multiple Faults, *Artificial Intelligence*, 32:97-130, 1987.

[17] J. de Kleer, K. Mackworth, and R. Reiter: Characterizing Diagnoses and Systems, *Artificial Intelligence*, 56, 1992.

[18] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood: Amphion: Automatic Programming for Scientific Subroutine Libraries. In *Proceedings of the International Symposium on Methodologies for Intelligent Systems (ISMIS-94)*, 1994.

[19] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood: A Formal Approach to Domain-Oriented Software Design Environments. In *Proceedings of the 9th Knowledge-Based Software Engineering Conference (KBSE-94)*, Monterey, CA, September 20-23, 1994.

[20] H. Mili, F. Mili, and A. Mili: Reusing Software: Issues and Research Directions, *IEEE Transactions on Software Engineering*, 21(6):528—562.

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

[22] L.C. Paulson and T. Nipkow: Isabelle: *Tutorial and Users Manual*, technical report, University of Cambridge, no 189, 1990.

[23] F. Puppe: *Systematic Introduction to Expert Systems: Knowledge Representation and Problem-Solving Methods*, Springer-Verlag, Berlin, 1993.

[24] 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.

[25] 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.

[26] 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.

[27] M. Shaw and D. Garlan: *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[28] D. R. Smith: Top-Down Synthesis of Divide-and-Conquer Algorithms, *Artificial Intelligence*, 27:43—96, 1985.

[29] D. R. Smith: Towards a Classification Approach to Design. In *Proceedings of th 5th International Conference on Algebraic Methodology and Software Technology (AMAST-96)*, Munich, Germany, July 1-5, 1996.

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

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

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

[33] M. Wirsing: Algebraic Specification Languages: an Overview. In *Proceedings of the 10th Workshop on Specification of Abstract Data Types*, Springer-Verlag, LNCS 906, 1995.