Specification and Verification of Knowledge-Based Systems

Dieter Fensel[1], Arno Schönegge[2], Rix Groenboom[3], Bob Wielinga[1]

[1] University of Amsterdam, Department SWI, Roetersstraat 15, 1018 WB Amsterdam, the Netherlands, E-mail: {fensel | wielinga}@swi.psy.uva.nl
[2] University of Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme, 76128 Karlsruhe, Germany. E-mail: schoeneg@ira.uka.de
[3] University of Groningen, Department of Computing Science, P.O. Box 800, 9700 AV Groningen, the Netherlands. E-mail: rix@cs.rug.nl

Abstract. The paper introduces a formal approach for the specification and verification of knowledge-based systems. We identify different elements of such a specification: a task definition, a problem-solving method, a domain model, an adapter, and assumptions that relate these elements. We present abstract data types and a variant of dynamic logic as formal means to specify these different elements. Based on our framework we can distinguish several verification tasks. In the paper, we discuss the application of the Karlsruhe Interactive Verifier (KIV) for this purpose. KIV was originally developed for the verification of procedural programs but it fits well for our approach. We illustrate the verification process with KIV and show how KIV can be used as an exploration tool that helps to detect assumptions necessary to close the gap between the task definition and the competence of a problem-solving method.

1 Introduction

During the last years, several conceptual and formal specification techniques for knowledge-based systems (KBS) have been developed (see [18], [15] for surveys). The main advantage of these modelling or specification techniques is that they enable the description of a KBS independent of its implementation. This has two main implications. First, 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 (based on libraries of problem-solving methods [33], [5], [2] and domain ontologies [49]) it becomes an essential task to verify whether the assumptions of such a reusable building block fit to the actual provided task and knowledge. Second, such a specification can be used as golden standard for the validation and verification[1] of the implementation of the KBS. It defines the requirements the implementation must fulfil.

The work that is presented in this paper provides three contributions to the field.

First, we will develop a conceptual and formal framework for the specification of KBSs. The conceptual framework is developed in accordance to the CommonKADS model of expertise [42] because this model has become widely used by the knowledge engineering community. The formal means applied are based on combining variants of algebraic specification techniques [4] and dynamic logic [24]. We could not completely rely on existing specification languages for KBS since most of them cover only a subset of a complete specification of a KBS or lack from a appropriate semantics and axiomatization.

Second, we identify several proof obligations that arise in order to guarantee a consistent specification. The overall verification of a KBS is broken down into five different types of proof obligations that ensure that the different elements of a specification together define a consistent system with appropriate functionality.

Third, we show how the Karlsruhe Interactive Verifier (KIV) [37], developed in the area of program verification, provides support in proceeding these proofs. Our view of verification is not restricted to the task of decorating software with a *verified* stamp after its construction. We believe that verification techniques can be valuable already in the development process (cf. [28]). Especially in the context of the development process for KBS as proposed here we think of detecting (hidden) assumptions of PSMs by analysing failures of proof attempts.

The paper is organised as follows. In section 2, we discuss the different elements of a formal specification of a KBS and which kinds of proof obligation arise in their context. In section 3 we introduce an example of a task definition. We discuss the task of selecting a best explanation in abductive diagnosis. In section 4, we provide the definition of the weak PSM *hill-climbing* that finds a *local* optimum. We will then discuss in section 5 how ontology mappings and additional assumptions relate the task description with the competence of *hill-climbing*. In section 6 is shown, how the Karlsruhe Interactive Verifier (KIV) can be used for these verification tasks. We demonstrate termination and equivalence proofs and show how KIV can be used as an exploration tool that helps to find the appropriate assumptions necessary to link a competence of a PSM with a task definition. Section 7 summarizes the paper and defines objectives for future research.

2 A Formal Framework for the Specification of

1. There is no clear consensus in the literature over the precise definition of the terms validation and verification. In our context, a formal proof that a specification or a program fulfils some properties postulated by another specification will be called verification. A formal specification can be used to validate the behavior of a program in the case it is operational. The implementation should then produce output that corresponds to the output produced by the operational specification. We use the term *correspondence* instead of *equivalence* to cover non-deterministic specifications and partial evaluation techniques. During the paper, we focus on the internal verification of a specification and its different parts.

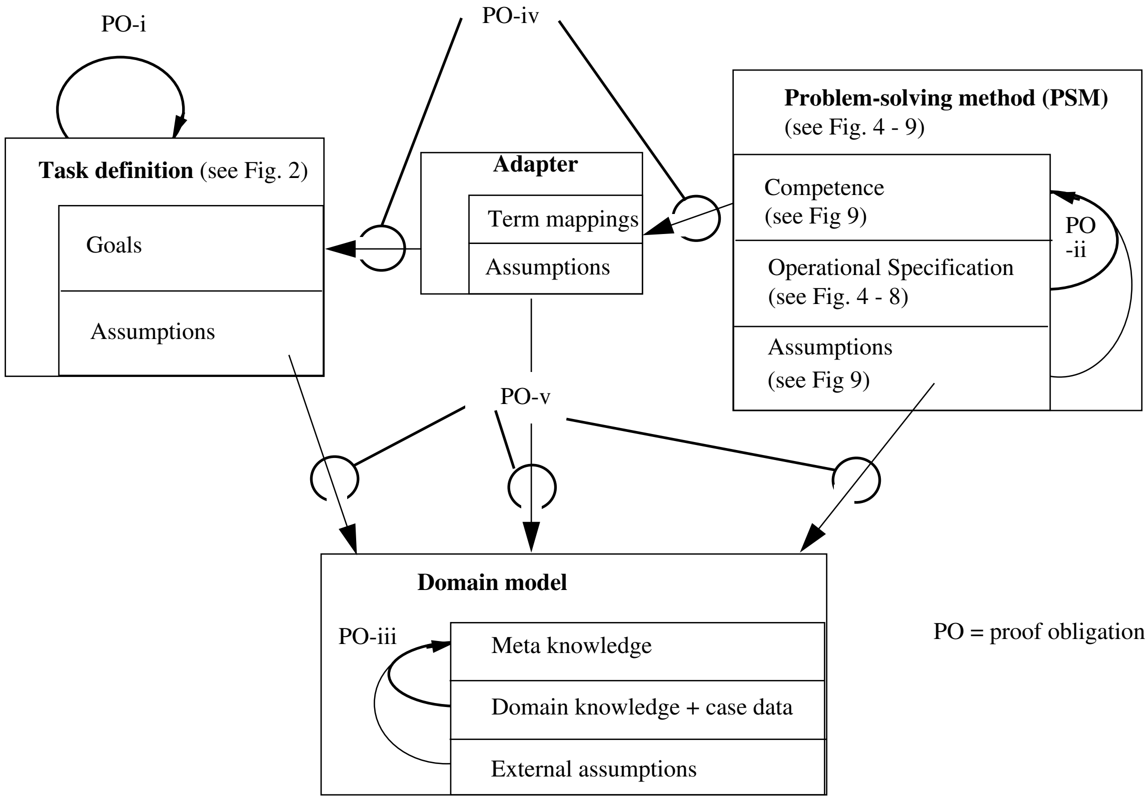

Fig. 1 The four elements of a specification of a KBS.

Knowledge-Based Systems

During the following, we first introduce the different elements of a specification. Then we discuss, how they are related and which proof obligations arise from these relationships.

2.1. The Main Elements of a Specification

A framework for describing a knowledge-based system consists of four elements (see Figure 1): a *task* definition that defines the problem that should be solved by the KBS; a *problem-solving method* (PSM) that defines the reasoning process of a KBS; and a *domain model* that describes the domain knowledge of the KBS. Each of these elements are described independently to enable the reuse of task descriptions in different domains [5], the reuse of PSMs for different tasks and domains (cf. [33], [5], [2]), and the reuse of domain knowledge for different tasks and PSMs (cf. [44], [49]). A fourth element of a specification of a KBS is an *adapter* that is necessary to relate the functionality of a PSM and the provided domain knowledge with the desired functionality as it is defined by the task definition. Additional assumptions have to be introduced and the different terminologies have to be mapped. The necessity of adapters arise from the reusability of the other building blocks of a specification. They must be adjusted to each other and to the specific requirements of a given application problem.

The description of a *task* specifies some goals that should be achieved in order to solve a given problem. This functionality is specified as a relation between input and output of a KBS. A second part of a task specification is the definition of *assumptions* over domain knowledge. For example, a task that defines the selection of the maximal element of a given set of elements requires a preference relation as domain knowledge. Assumptions are used to define the requirements on such a relation (e.g. transitivity, connexitivity, etc.). A natural candidate for the formal task definition are *algebraic specifications*. They have been developed in software engineering to define the functionality of a software artefact (cf. [4], [11], [50]) and have already been applied by [43] and [31] for KBS. In a nutshell, algebraic specifications provide a signature consisting of types, constants, functions and predicates and a set of axioms that define properties of these syntactical elements.

The description of the *reasoning process* of the KBS by a PSM consists of three elements. First, the definition of the functionality of the PSM. Such a functional specification defines the *competence* of a PSM independent from its realization (cf. [46], [1], [48]). Again algebraic specifications can be used for this purpose. Second, an *operational description* defines the dynamic reasoning process of a PSM. Such an operational description explains how the desired competence can be achieved. It defines the main reasoning steps (called *inference actions*) and their dynamic interaction (i.e., the knowledge and control flow) in order to achieve the functionality of the PSM. Dynamic logic [24] or temporal logic [45] are required to specify the dynamic interaction of these inferences. The definition of such an inference step could recursively introduce a new (sub-)task definition. This process of stepwise refinement stops when the realization of such an inference is regarded as an implementation issue that is neglected during the specification process of the KBS. The third element of a PSM are *assumptions* over domain knowledge. Each inference step requires a specific type of domain knowledge. These complex requirements on the input of a PSM distinguish it from usual software products. Pre-conditions on valid inputs are extended to complex requirements on available domain knowledge.[2]

The description of the *domain model* introduces the domain knowledge as it is required by the PSM and the task definition. Ontologies (i.e., meta-theories of domain knowledge) are proposed in knowledge engineering as a means to explicitly represent the commitments of a domain knowledge (cf. [44], [49]). For our purpose, we require three elements for defining a *domain model*: First, a description of properties of the domain knowledge at a meta-level. The *meta knowledge* characterizes properties of the domain knowledge. It is the counter part of the assumptions on domain knowledge of the other parts of a specification. They reflect the assumptions of task definitions and PSM on domain knowledge. Second, the *domain knowledge* and *case data* necessary to define the task in the given application domain and necessary to proceed the inference steps of the chosen PSM. Third, *external assumptions* that relate the domain knowledge with the domain (i.e., the system model with the actual system). These assumptions link the domain knowledge with the actual domain. These assumptions can be viewed as the missing pieces in the proof that the domain knowledge fulfils its meta-level characterizations. Some of these properties may be directly inferred from the domain knowledge whereas others can only be derived by introducing assumptions on the environment of the system.

The description of an *adapter* maps the different terminologies of task definition, PSM, and domain model and introduces assumptions that have to be made to relate the competence of a PSM with the functionality as it is introduced by the task definition (cf. [13], [3]). It relates the three other parts of a specification together and establishes their relationship in a way that meets the specific application problem. Each of the three other elements can be described independently and selected from libraries of reusable task definitions, PSM[3], and domain models. Their consistent combination and their adaption to the specific aspects of the given application (because they should be reusable they need to abstract from specific aspects of application problems) must be provided by the adapter. Its assumptions are necessary as in general, most problems tackled with knowledge-based systems are inherently complex and intractable, i.e., their time complexity is NP-hard ([7], [8], and [30]). A PSM can only solve such tasks with reasonable computational effort by introducing assumptions that restrict the complexity of the problem or strengthen the assumptions over domain knowledge. In the following, we illustrate the adaption of a PSM in accordance to a formal description of a task by introducing assumptions. These assumptions either weaken the task definition or introduce additional requirements on the domain knowledge that is expected by the PSM. In the first case, the task is weakened to meet the competence of the PSM and in the second case, the competence of the method is strengthened by the assumed domain knowledge.

2. In terms of [47], a task definition is an *extensional* specification and a PSM combines *extensional* with *intensional* specification elements. The entire competence of the PSM and their elementary reasoning steps are specified extensionally. The interaction of the elementary reasoning steps in order to achieve the competence is specified intensionally.

3. As a consequence, PSM can be described independent from domains and tasks as proposed in [27].

2.2. The Main Proof Obligations

Following the development process for task and domain specific PSMs as proposed in this paper the overall verification of a KBS is broken down into five kinds of proof obligations (see Figure 1):

(i) the consistency of the task definition ensures that a model of the task definition exist[4]

$$\textit{Task definition} \not\vdash \bot;$$

(ii) the operational description of the PSM exhibits the functionality described in the competence theory, i.e.,

$$PSM_{assumptions} \vdash (\langle PSM_{operational}\rangle \text{ true} \wedge [PSM_{operational}]\, PSM_{competence})$$

this proof obligation recursively returns for each inference action of a PSM;

(iii) the internal consistency of the domain model

$$\textit{Domain model}_{assumptions} \wedge \textit{Domain model}_{domain\ knowledge} \vdash \textit{Domain model}_{meta\ knowledge};$$

(iv) under certain assumptions, the functionality described in the competence theory is sufficient for solving the task, i.e.,

$$Adapter_{assumptions} \vdash (PSM_{assumptions} \wedge PSM_{competence} \rightarrow \textit{Task definition});$$

(v) the domain model fulfils these assumptions, i.e.,

$$\textit{Domain model}_{meta\ knowledge} \vdash Task_{assumptions}$$
$$\textit{Domain model}_{meta\ knowledge} \vdash PSM_{assumptions}$$
$$\textit{Domain model}_{meta\ knowledge} \vdash Adapter_{assumptions}.$$

Notice that PO-i deals with the task definition internally, PO-ii deals with the task definition internally, and PO-iii deals with the domain model internally, whereas PO-iv and PO-v deal with the external relationships between tasks, PSM, assumptions, and domain knowledge. Thus a separation of concerns is achieved which contributes to the feasibility of the verification (cf. [25]). The conceptual model applied to describe knowledge-based system is used to brake the general proof obligations into smaller pieces and makes parts of them reusable. As PSMs can be reused, the proofs of PO-ii does not have to be repeated for every application. Only when a new PSM is introduced into the library (cf. [33], [5]), these proofs have to be done. Similar proof economy can be achieved for PO-i and PO-iii by reusable task definitions and domain models. Application specific proof obligations are PO-iv and PO-v. The first links the competence of the PSM to the functionality of the task introducing some assumptions. PO-v ensures that the assumptions over domain knowledge are fulfilled.

3 The Task "Select A Best Explanation"

The task *abductive diagnosis* requires a set of observations as input and tries to deliver a correct, complete and most plausible diagnosis as output (see e.g. [8]). Roughly, one can identify three reasoning tasks of abductive diagnosis:

- generating possible diagnoses;
- testing whether these possible diagnoses are correct and complete;
- selecting the diagnosis with the highest plausibility.

In the following we will focus on the third subtask selecting the explanation with the highest plausibility. The task assumes a set of correct and complete hypotheses for a given abductive problem as

4. The proof is usually done by constructing a model via an (inefficient) generate & test like implementation.

task *Select Best Diagnosis*

sorts

H_{all}, *plausibilities*

functions

$pl: H_{all} \rightarrow plausibilities$

predicates

$hypothesis : H_{all}$

$diagnosis : H_{all}$

$pref : plausibilities \times plausibilities$[1]

variables $x,y,z : H_{all}$

axioms

goal

(1) Each *diagnosis* is a *hypothesis*

$\forall x\ (diagnosis(x) \rightarrow hypothesis(x))$

(2) The *diagnosis* is a global optimum

$\forall x\ (diagnosis(x) \rightarrow \forall y\ (hypothesis(y) \rightarrow pref(pl(y),pl(x))))$

input requirement

(3) Non-emptyness of *hypothesis*

$\exists x\ hypothesis(x)$

knowledge requirement

(4) reflexivity of *pref*:

$\forall x\ pref(pl(x),pl(x))$

(5) transitivity of *pref*:

$\forall x,y,z\ (pref(pl(x),pl(y)) \wedge pref(pl(y),pl(z)) \rightarrow pref(pl(x),pl(z)))$

(6) antisymmetry of *pref*:

$\forall x,y\ (pref(pl(x),pl(y)) \wedge pref(pl(y),pl(x)) \rightarrow pl(x) = pl(y))$

(7) connexivity of *pref*:

$\forall x,y\ (pref(pl(x),pl(y)) \vee pref(pl(y),pl(x)))$

endtask

1. Read $pref(x,y)$ as y is preferred over x.

Figure 2. The task definition *select best diagnosis*.

input. As output, it provides a diagnosis having the maximal plausibility.

Figure 2 defines the signature of the task under concern. The sort H_{all} defines the set of all possible hypotheses that can explain possible faults in the domain. The sort *plausibilities* is used to express the plausibility of a hypothesis. H_{all} and *plausibilities* must be data or knowledge types in the given domain and introduce ontological commitments. The function *pl* must be provided as domain knowledge. It relates hypotheses to their property that gets optimized.

The predicate *hypothesis* defines the set of actual hypotheses from which we search for the optimal element. The interpretation of the predicate *hypothesis* must be provided as data input to the task. The assumption over input is that it is not empty (see (3)).

A further predicate *pref* is required to distinguish between different elements of *hypothesis* by imposing an ordering in them. More precisely spoken, it is used to distinguish their values of the function *pl*. It defines an additional ontological commitment as this preference (i.e., the interpretation of the predicate *pref*) must be provided as domain knowledge. This semantical requirement for a preference relation must be further characterized by its properties. The precise definition of the preference predicate by axioms (4) - (7) defines requirements on the domain relationship that is used to interpret it. These axioms ensure that pref defines a *linear ordering* (sometimes also called total ordering).

We still have to define the functionality that should be provided by the task. That is, we have to define the predicate *diagnosis* with the predicate symbols *hypotheses*, *pref* and the function symbol *pl*. With axioms (1) + (2) we ask for a *maximal* element of input as goal of the task.

4 The Weak Problem-Solving Method Hill-climbing

There exist several techniques (see e.g. [40]) and PSMs (see [33], [5]) that are available to solve a given task. Three main decisions have to taken for deciding which problem-solving technique should be applied:

- Is the problem-solving technique or method suitable for the given type of task and how large is the effort to map the terminology of the task onto the terminology of the method ([12], [16])?
- What types of domain knowledge are required by it? For example, a problem-solving method like propose & revise can only be applied to a parametric design problem if knowledge is available that can be used to propose initial parameter values and knowledge that can be used to repair constraints violations. Different variants of propose & revise can be identified by making these requirements on domain knowledge more precise [13].
- How important and problematic is the efficiency aspect? Blind

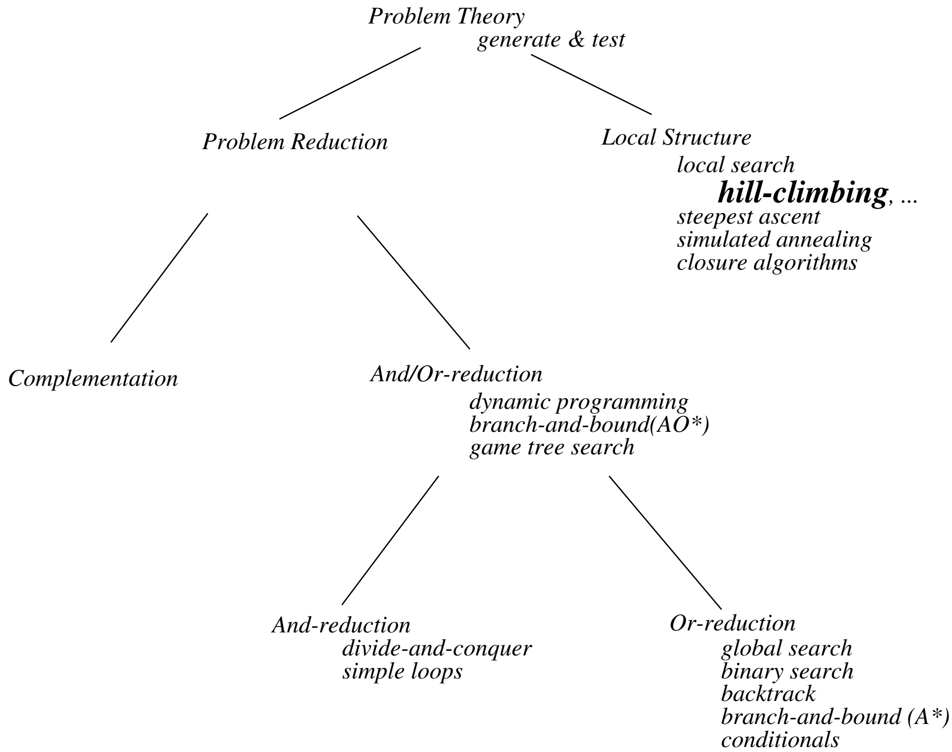

Figure 3. Refinement hierarchy of algorithm theories (see [40]).

generate & test like methods have low efficiency but are easy to understand and implement. More sophisticated methods that compile test knowledge into the generate step provide higher efficiency but provide less understandability and require more effort in implementing and maintaining them ([10], [19]).

The main distinction between the algorithmic techniques of [40] or so-called weak problem-solving methods (actually heuristic and non-heuristic search techniques, [6]) at the one hand and problem-solving method at the other hand is that the latter make stronger commitments on the terminological structure of the tasks they can be applied to and the type of knowledge they require. For example, the task-specific board-game method is gained from the weak search strategy chronological backtracking by posing an internal terminological structure on the states of the search process and using this structure to define state transitions ([12], [16]) that define stronger requirements on

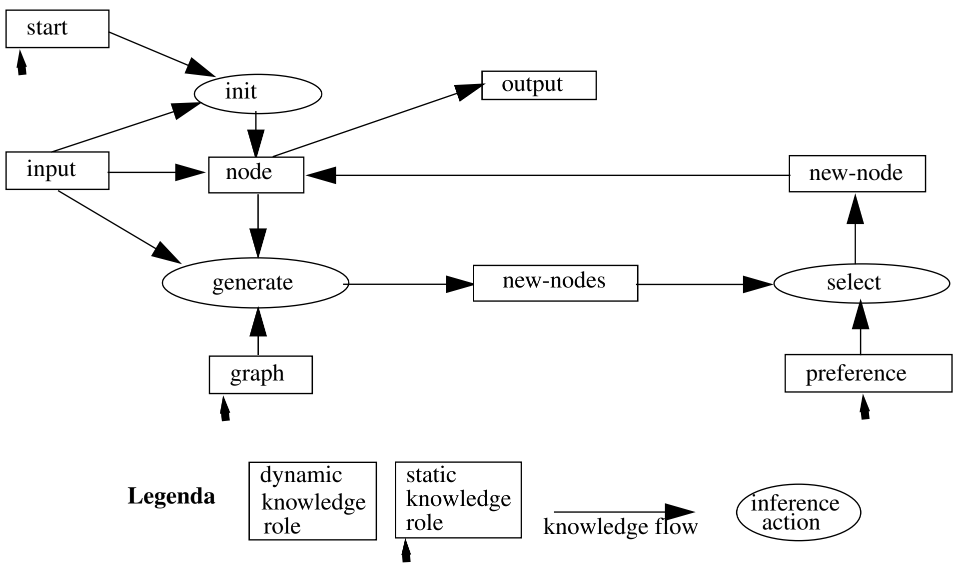

Fig. 4 Knowledge flow diagram of *hill-climbing*.

inf init
sorts *nodes*
static predicates *input*, *start* : *nodes*
dynamic predicates *node* : *nodes*
variables *x,y* : *nodes*
axioms
$[init]\ \forall x\ (node(x) \rightarrow (input(x) \wedge start(x)))$
$[init]\ \forall x,y\ (node(x) \wedge node(y) \rightarrow x = y)$
implementation
$init =_{\text{def}} node :\leftrightarrow \varepsilon x.(input(x) \wedge start(x))$
endinf

inf generate
sorts *nodes*
static predicates *input* : *nodes, node* : *nodes, graph* : *nodes* × *nodes*
dynamic predicates *new nodes* : *nodes*
variables *x,y* : *nodes*
axioms
$[generate]\forall y\ (new\text{-}nodes(y) \rightarrow input(y) \wedge \exists x\ (node(x) \wedge graph(x,y)))$
implementation
$generate =_{\text{def}} new\text{-}nodes :\leftrightarrow \lambda y.(input(y) \wedge \exists x\ (node(x) \wedge graph(x,y)))$
endinf

inf select
sorts *nodes, pref-values*
static functions *h* : *nodes* → *pref-values*
static predicates *new nodes* : *nodes, pref* : *nodes* × *nodes*
dynamic predicates *new node* : *nodes*
variables *x,y* : *nodes*
axioms
$[select]\ \forall x\ (new\text{-}node(x) \rightarrow new\text{-}nodes(x) \wedge \forall y\ (new\text{-}nodes(y) \rightarrow pref(h(y),h(x))))$
$[select]\ \forall x,y\ (new\text{-}node(x) \wedge new\text{-}node(y) \rightarrow x = y)$
$\exists x\ new\text{-}nodes(x) \rightarrow [select]\ \exists x\ new\text{-}node(x)$
implementation
$select =_{\text{def}} new\text{-}node :\leftrightarrow \varepsilon x.(new\text{-}nodes(x) \wedge \forall y\ (new\text{-}nodes(y) \wedge pref(h(y),h(x))))$
endinf

Figure 5. The inferences *init*, *generate* and *select*.

available domain knowledge.

Because for reasons of simplicity we have chosen a simple task definition, we can choose a weak problem-solving method for solving it. The task is a simple selection task where the elements are characterized by one property that should be optimized.[5] We decided to choose the local search method *hill-climbing* for our example (cf. Figure 3). *Hill-climbing* is a local search algorithm that stops when it has found a local optimum.

The main requirement on *domain knowledge* that is introduced by *hill-climbing* (and by other local search methods) is the existence of a neighbourhood relationship between nodes (called *graph* during the following) that is used to guide the local search process. In general, several domain-specific possibilities exist to define such a graph relationship. In component-based diagnosis where each hypothesis is a set of defect components that explain the fault behaviour of a device this graph relation could be defined between two hypotheses by:

$graph(hypothesis_1,hypothesis_2) :\leftrightarrow$
$\exists\ c\ (hypothesis_1 = hypothesis_2 \cup \{c\} \wedge c \notin hypothesis_2)$.

That is, the neighbour of a hypothesis are all hypotheses that contain one less defect component . It is clear that such a definition only makes sense if the preference of a hypothesis is related to the number of components it assumes to be defect. A further example for such a graph os provided in machine learning when one searches locally for rules or decision trees that cover a set of examples (cf. [34]). The graph relation is defined between rules that can be derived from each other by adding or deleting one premise. In general, *hill-climbing* can only be applied if a relation in the domain exist that can be used to define a graph for its search process.

One could argue why we do not just use a method that pairwise compares elements of the given set of hypotheses in random order as the complexity of the problem is only linear in the number of nodes. Still for large hypotheses sets, heuristic techniques must be applied. For example, the number of hypotheses increases exponentially in component-based diagnosis in the number of components (the set of

5. In terms of CommonKADS such a simple task would be solved by an elementary inference action *select* that is part of several more complex problem-solving methods [5].

dkr input
  sorts *nodes*
  predicates *input* : *nodes*
  axioms
    $\exists x\ input(x)$
enddkr

dkr output
  sorts *nodes*
  predicates *output* : *nodes*
enddkr
dkr node
  sorts *nodes*
  predicates *node* : *nodes*
enddkr

dkr new-node
  sorts *nodes*
  predicates *new-node* : *nodes*
enddkr
dkr new-nodes
  sorts *nodes*
  predicates *new-nodes* : *nodes*
enddkr

Figure 6. The dynamic knowledge roles of *hill-climbing*.

hypotheses is the power set of the set of components). Therefore, efficiency of the search process becomes important and a heuristic approach is necessary.

In the following we provide the operational specification together with its knowledge requirements, the competence theory, and finally the proof obligations of this method.

4.1. Operational Specification of Hill-Climbing

Hill-climbing finds a local optimum. The entire method is decomposed into the following three steps. The inference action *init* selects the start node. An inference action *generate* generates all neighbours of a node. The inference action *select* selects the node with the highest value. Figure 4 gives the knowledge flow diagram of the method.

4.1.1 Inference Actions.

We use algebraic specifications enriched by the modality operators of dynamic logic to specify the functionality of inference actions. We distinguish between predicates that have the same truth values in the initial state and in the state after the execution of an inference action (called *static predicates*) from the predicates that change as a result of executing the inference action (called *dynamic predicates*).[6] Figure 5 provides the definition of the three inference actions of the PSM. The functional specification is extended by an operational specification

6. Dynamic knowledge roles and static knowledge roles should not get mixed with static and dynamic predicates. The former are determined in the context of the entire problem-solving method and the latter are determined in the context of an individual inference action.

skr preference
  sorts
    nodes, pref-values
  functions
    $h : nodes \rightarrow pref\text{-}values$
  predicates
    $pref : pref\text{-}values \times pref\text{-}values$
  variables ***x,y,z* : *nodes***
  axioms[1]
    (1) reflexivity of *pref*:
      $\forall x\ pref(h(x),h(x))$
    (2) transitivity of *pref*:
      $\forall x,y,z\ (pref(h(x),h(y)) \wedge pref(h(y),h(z)) \rightarrow pref(h(x),h(z)))$
    (3) antisymmetry of *pref*:
      $\forall x\ ,y(pref(h(x),h(y)) \wedge pref(h(y),h(x)) \rightarrow h(x) = h(y))$
    (4) connexivity of *pref*:
      $\forall x\ ,y(pref(h(x),h(y)) \vee pref(h(y),h(x)))$
endskr

view graph
  sorts
    nodes
  predicates
    $graph : nodes \times nodes$
endview

view start
  sorts
    nodes
  predicates
    start : *nodes*
endview

Actually, the axioms of *preference* **are stronger than required. For hill-climbing, the linear ordering property of** *pref* **must hold only between neighborhoud nodes. That is, between nodes that are linked by the** *graph* **relationship.**

Figure 7. The static knowledge roles of *hill-climbing*.

init
if $\neg\exists x$ *node*(x) **then** *node* $:\leftrightarrow \varepsilon x.input(x)$ **endif**
output $:\leftrightarrow \lambda x.false$;
while $\neg\exists x$ *output*(x)
do
  generate;
  if $\exists x$ *new-nodes*(x)
    then
      select
      if $\forall x\ \forall y\ ((new\text{-}node(x) \wedge node(y) \rightarrow \neg pref(h(x),h(y)))$
        then *node* $:\leftrightarrow \lambda x.new\text{-}node(x)$
        else *output* $:\leftrightarrow \lambda x.node(x)$
      endif
    else *output* $:\leftrightarrow \lambda x.node(x)$
  endif
od

Figure 8. The control flow of hill-climbing.

(called implementation) that express the inference in a procedural way. We use a variant of dynamic logic for this purpose. The Modal Logic for Predicate Modification (MLPM) [17] was developed as a generalization of specification languages KARL [14] and $(ML)^2$ [26]. In addition, it provides an axiomatization that enables automated proofs. MLPM represents a state by the truth values of the predicates. An elementary state transition is achieved by changing the truth values of a predicate according to the truth values of a formula that is used to define the transition. Two different types of such elementary state transitions exist:

$p :\leftrightarrow \varepsilon x.\varphi$ and $p :\leftrightarrow \lambda x.\varphi$

The ε-operator expresses non-deterministic selection of one ground literal. A formula φ can be used to restrict the set of possible ground literals from which one is chosen. All other ground literals of the predicate p are set to false. The λ-operator allows updates of all ground literals of a predicate p according to the truth values of a formula φ. All ground literals are set to true for which the according variable assignments evaluate the formula to true. All other ground literals of the predicate p are set to false.

MLPM provides the usual procedural constructs such as sequence, if-then-else, choice, and while-loop to define complex transition. We will make use of these constructs in section 4.1.4 when we define the operational specification of the entire *hill-climbing* method. As inference actions are regarded to be primitive they are defined by only one elementary transition.

An interesting feature of the inference action *select* is that it recursively repeats the definition of the original optimization task. In addition, *select* requires to find a global maximum (of all neighbourhood nodes) whereas the competence of *hill-climbing* can only guarantee to find a local one.

4.1.2 Dynamic Knowledge Roles

Dynamic knowledge roles (dkr) are means to represent the state of the reasoning process and are modelled by algebraic specifications (see Figure 6). Axioms can be used to represent state invariants. We define the requirement that the input provided to the method has to be non-empty.

4.1.3 Static Knowledge Roles

Static knowledge roles (skr) are means to include domain knowledge into the reasoning process of a PSM. Again, they are modelled by algebraic specifications. Axioms are used to define assumptions over the domain knowledge. *Hill-climbing* requires three types of domain knowledge. A preference is required to select an optimal neighbourhood node and a graph is required that defines the search space of *hill-climbing*, i.e. the neighbourhood relationship. The properties of this relation (i.e., the topography of the search space) heavily influence behavior and result of *hill-climbing*. Finally, knowledge is required that selects the initial node used to start the search process. The static knowledge roles are defined in Figure 7.

4.1.4 Control Flow

The operational description of a PSM is completed by defining the control flow (see Figure 8) that defines the execution of the inference actions. Again, we use Modal Logic for Predicate Modification (MLPM) (see section 4.1.1). An elementary state transition is achieved by changing the truth values of a predicate according to the truth values of a formula that is used to define the transition. Complex transitions are built up by defining procedural control (i.e., sequence, branch, and loop) on top of these elementary transitions.

The methods works as follows: First, we select a start node with *init*. If *init* fails to select a start node (i.e., no node in input is an element of the predicate *start*, see Figure 5), we randomly select one node from *input* for this purpose. After having selected this first node, a while loop is executed that stops when a local optimum is written to *output*. Within the loop we first generate all neighbours of the current node. If *generate* fails to generate new nodes (i.e., the current node has no neighbours) we are finished and write the current node to *output*. Otherwise, we select a best neighbour and compare it with the current node. If the best neighbour is better than the current node we use it for the next iteration of the loop. In the other case, we are finished and write the current node to *output*.

4.2. Competence Theory

The competence theory describes the functionality of the PSM. Again algebraic specifications enriched by the modality operators of

competence *hill-climbing*

sorts

nodes, pref-values

static functions

$h : nodes \rightarrow pref\text{-}values$

static predicates

$input : nodes$

$pref : pref\text{-}values \times pref\text{-}values$

$graph : nodes \times nodes$

dynamic predicates

$output : nodes$

variables ***x,y,z* : *nodes***

axioms

input requirement

(1) non-emptyness of *input*:
$\exists x\ input(x)$

knowledge requirement

(2) reflexivity of *pref*:
$\forall x\ pref(h(x),h(x))$

(3) transitivity of *pref*:
$\forall x,y,z\ (pref(h(x),h(y)) \wedge pref(h(y),h(z)) \rightarrow pref(h(x),h(z)))$

(4) antisymmetry of *pref*:
$\forall x\ ,y(pref(h(x),h(y)) \wedge pref(h(y),h(x)) \rightarrow h(x) = h(y))$

(5) connexivity of *pref*:
$\forall x\ ,y(pref(h(x),h(y)) \vee pref(h(y),h(x)))$

post condition

(6) Each *output* was an element of *input*
$[hill\text{-}climbing]\ \forall x\ (output(x) \rightarrow input(x))$

(7) Only one *output* is provided
$[hill\text{-}climbing]\ \forall x\ ,y(output(x) \wedge output(y) \rightarrow x = y)$

(8) There is an *output*
$[hill\text{-}climbing]\ \exists x\ output(x)$

(9) local optimality of *output*:
$[hill\text{-}climbing]\ \forall x\ (output(x) \rightarrow \forall y\ (input(y) \wedge graph(x,y) \rightarrow pref(h(y),h(x))))$

endcompetence

Figure 9. The competence theory of *hill-climbing*.

dynamic logic can be used for this purpose (see Figure 9). As in the description of inference actions we distinguish static and dynamic predicates.

The competence theory in Figure 9 defines that *hill climbing* is able to find a *local* optimum of the given set of elements.

4.3. Proof Obligations PO-ii

Does the operational specification of *hill-climbing* in section 4.1. have the competence as defined in 4.2. and can we guarantee termination of the method? Both proofs are sketched in section 6. Actually, several "small" errors were found in the original specification (compare section 5). As *hill-climbing* is a very simple PSM this illustrates the need for verification and tool support during this activity. As PSMs are designed for reuse these proofs need not to be repeated for every application. They have to be done only once when a PSM is added to a library of reusable PSMs.

5 Adapter: Linking the Task Definition with the Competence of the PSM

Linking a task definition and a domain model with a PSM requires two activities. First, the different terminologies have to be related (i.e., the different ontologies have to be mapped). Second, we have to relate the strength of the PSM with the desired goal of the task definition.

To relate the given task definition with the PSM, we have to relate the sorts H_{all} and *plausibilities*, the function *pl*, and the predicates *hypotheses*, *pref*, and *diagnosis* of the task definition (see Figure 2) with the sorts *nodes* and *pref-values*, the function *h*, and the predicates *input*, *pref*, and *output* of the competence of the PSM (see Figure 9).[7]

The PSM *hill-climbing* has the competence to find a local optimum in a graph. The task under concern requires to select an optimal element from a set. In general, there are two possible strategies to close this gap (see [3]). One can introduce additional requirements on

7. In KIV, this mapping can be achieved by defining *hill-climbing* as a generic module that gets actualized by the signature and the axioms of the task (see [35] for more details).

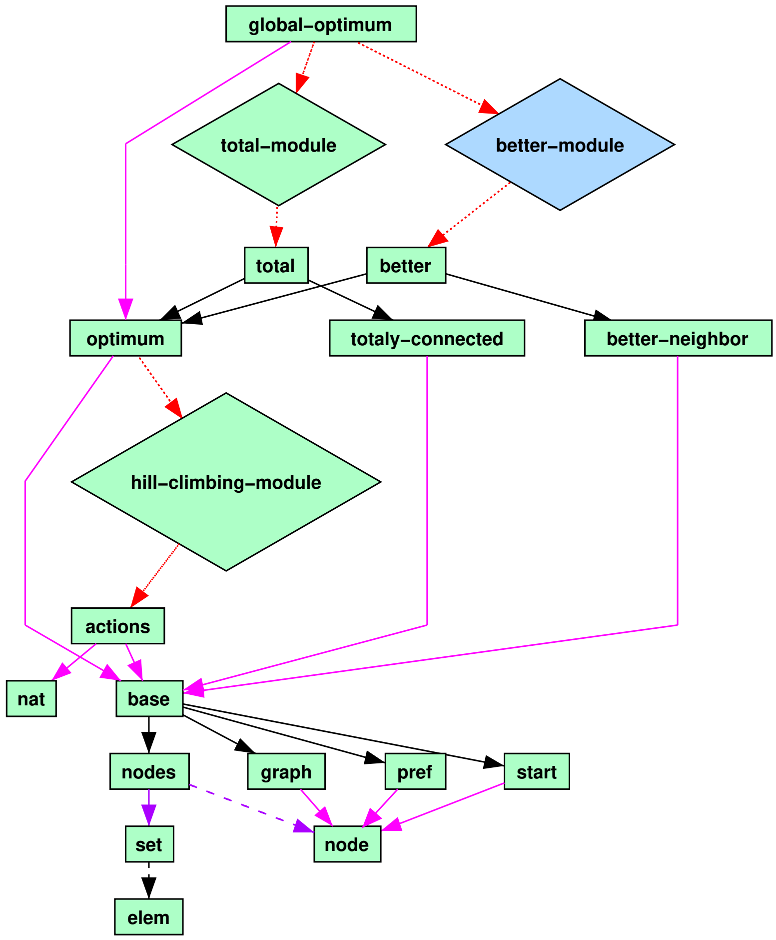

Figure 10. Specification and implementation of the PSM *hill-climbing* in KIV.

domain knowledge that enable *hill-climbing* to find a global optimum or one can weaken the task definition. The first type of assumptions is expressed by formulas over the terminology as it is defined in static knowledge roles and dynamic input knowledge roles of the PSM. Static knowledge roles define the requirements for domain knowledge. Assumptions that weaken the task definitions are defined in terms of the task definition.

A trivial assumption which ensures that *hill-climbing* finds a global optimum is to require that each node is directly connected with each node.

$$assumption_1: \forall x\, \forall y\, (node(x) \wedge node(y) \rightarrow graph(x,y))$$

In this case, *hill-climbing* collapses to a complete search in one step as all nodes of the graphs are neighbours of each possible start node. If we would improve the *init* inference action we could weaken the assumption to: Every starting node must be connected with all nodes. The requirement on *init* would be to select such a node.

A less drastic assumption is to require that each node (except the maxima) has a neighbour with a higher preference.

$$assumption_2: \forall\, x\, (node(x) \rightarrow \exists y\, (graph(x,y) \wedge pref(h(x),h(y)) \wedge h(x) \neq h(y)) \vee \forall z\, (node(z) \rightarrow pref(h(z),h(x))))$$

Actually, we will see that this assumptions is too weak to guarantee equivalence of task definition and PSM competence. One has to add an assumption over the input of the task (cf. section 6.3.). We realized this as a result of applying the theorem prover KIV. The missing piece of the assumption was detected as a remaining open premise of an interactively constructed, partial proof that failed to show the equivalence of task definition and PSM competence.

The definition of the second if-then-else choice in Figure 8 is very critical for the competence of the PSM. Originally, we had the following definition (1):

$$\forall x\, \forall y\, (new\text{-}node(x) \wedge node(y) \rightarrow pref(h(y),h(x))) \qquad (1)$$

As (1) can lead to infinite loops it was modified to (2) as an outcome of the termination proof.

$$\forall x\, \forall y\, (new\text{-}node(x) \wedge node(y) \rightarrow \neg pref(h(x),h(y))) \qquad (2)$$

The disadvantage of (2) is that *hill-climbing* stops the search process when it reached a plateau in the search space (i.e., no neighbour node is better, only worse or equal neighbour exist). Therefore, an alternative solution would have been to work with condition (1) but introduce stronger requirements on the *graph* relation which guarantee termination of the PSM. If this relation does not contain direct or indirect cycles termination can be proven.

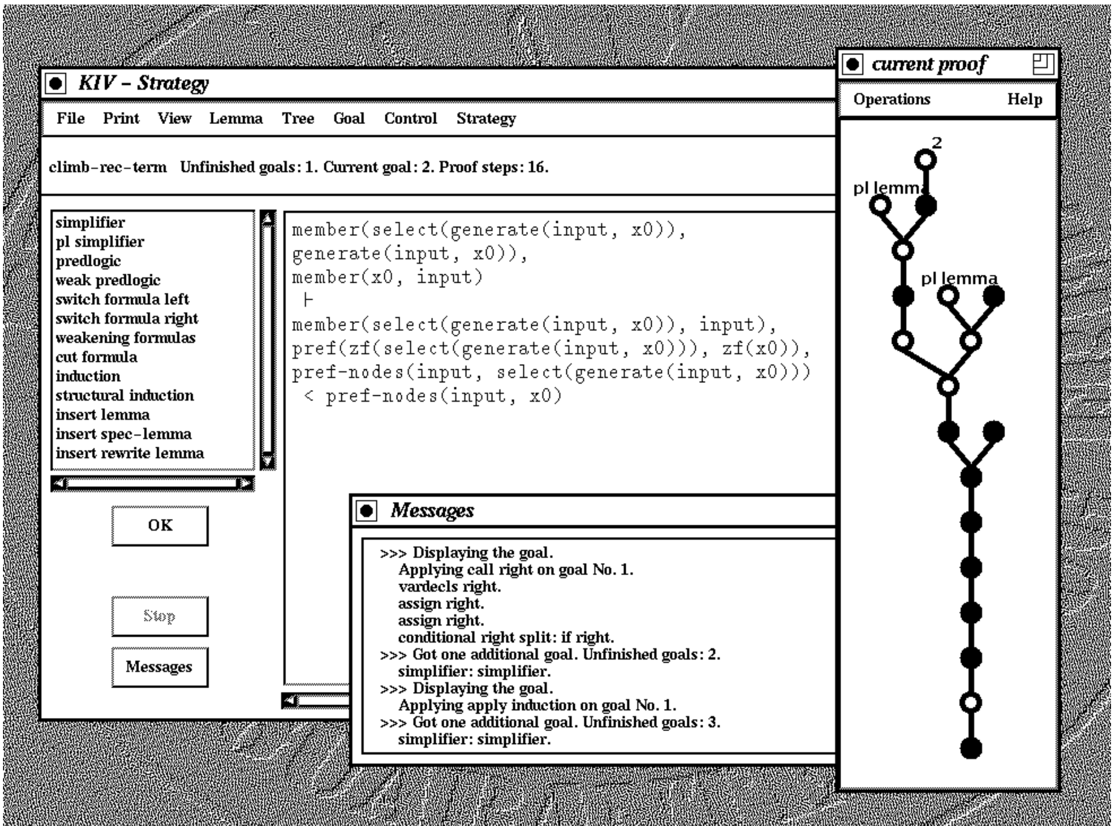

Figure 11. Verifying the PSM *hill-climbing* with KIV.

6 Verification with KIV

In software verification, tool support is extremely important. Large proofs and lemma bases with their (inter)dependencies have to be managed. Since in realistic applications correct programs (and specifications) are the exception rather than the rule, one has to keep track of (repeated) changes of lemmas and proofs. Thus, tool support for correctness management and reuse of proofs is necessary. Furthermore, the use of deduction systems permits to construct proofs substantially more accurate than can be done by hand. This increase in accuracy means an increase in reliability.

The KIV system (Karlsruhe Interactive Verifier) [36] is an advanced tool for the construction of provably correct software. It 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. [20]).

Our aim is to adapt the KIV system, originally designed for conventional software engineering, for development and verification of KBSs. For this purpose the KIV system is quite attractive. KIV supports dynamic logic which has been proved useful in specification of KBSs (cf. KARL, $(ML)^2$, and MLPM). Since the deduction machinery of KIV is basically a tactical theorem prover (in the LCF-style [21]), it is prepared for extensions and modifications. KIV allows structuring of specifications and modularisation of software systems. Finally, the KIV system offers well-developed proof engineering facilities:

- With the interactive proof environment even complicated proofs can be constructed.
- A high degree of automation is achieved by a number of implemented heuristics.
- Proof trees are visualized and can be manipulated with the help of a graphical user interface.
- Generation of counterexamples for unprovable goals is supported.
- An elaborated correctness management keeps track of lemma dependencies (and their modifications).
- Automatic reuse of proofs allows an incremental verification of corrected versions of programs and lemmas [38].

6.1. Applying KIV to the Hill-Climbing Example

In this section we report on results and experiences we gained in applying the KIV system to the *hill-climbing* example. Since we did not adapt the KIV system, we had to apply some small transformation on the formalization of *hill-climbing*. For example, dynamic predicates are modelled by variables and inference actions (*init*, *generate*, *select*) are represented by functions.

Figure 10 (which is original output from the KIV system using the graph visualization tool *daVinci*) shows how the modularisation facilities are employed. The actual algorithm is implemented in the *hill-climbing-module* (compare Figure 8). The inference actions are specified through axioms in the import of the module, i.e. in the specification *actions* (compare Figure 5, Figure 6 and Figure 7). The specification *optimum* corresponds to the competence of the *hill-climbing* method (compare Figure 9) and states that it computes a weak

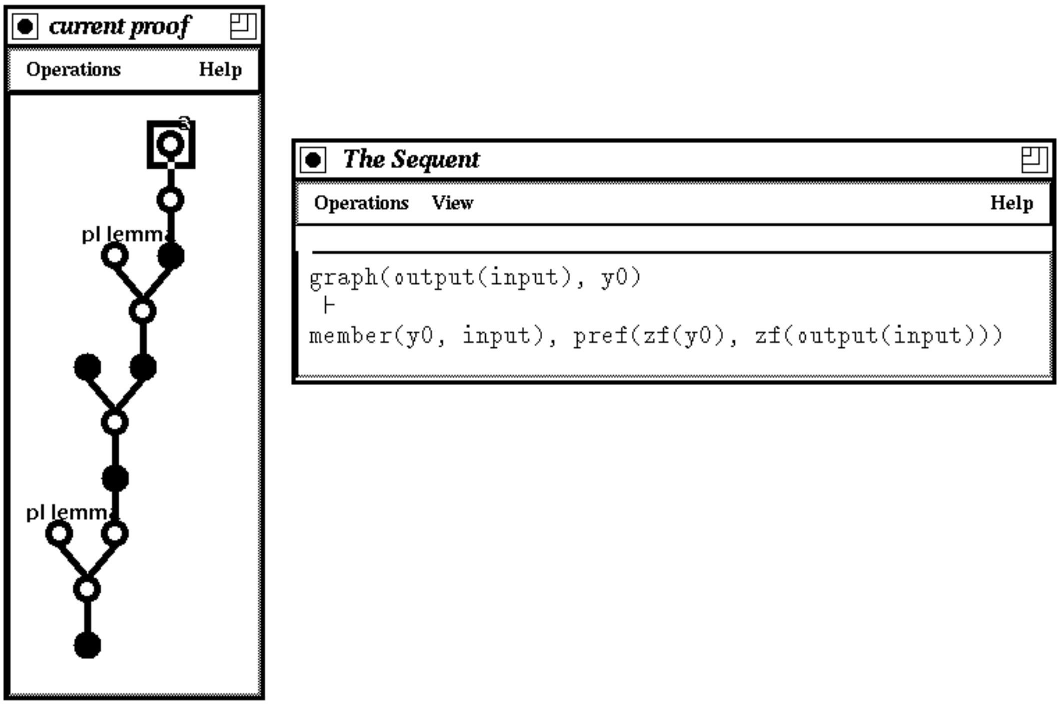

Figure 12. A partial proof gives hint for hidden assumptions.

local optimum. Extra assumptions are needed to get the global optimum required in the task. The two assumptions from section 5 are introduced by means of the specifications *totally-connected* (*assumption*$_1$) resp. *better-neighbour* (*assumption*$_2$) which restrict the import (domain) of the method.

In KIV proof obligations (formulated in dynamic logic) ensuring that an implementation (module) meets its export specification are automatically generated. For the *hill-climbing-module* these obligations are termination of the *hill-climbing* algorithm and its partial correctness with respect to the *optimum* specification (i.e., proof obligations PO-ii of section 2.2). The module *total-module* contains the proof obligation that total connectedness of the graph implies that the local optimum is also a global one (i.e., PO-iv of section 2.2 using *assumption*$_1$). The other module *better-module* contains the corresponding obligation for the assumption, that every node except the global maximum is connected to a better node (i.e., PO-iv of section 2.2 using *assumption*$_2$).

6.2. Verifying the PSM Hill Climbing

The obligations for the *hill-climbing-module* correspond to proof obligation (PO-ii) from section 2.2. Both, termination and partial correctness (i.e. *hill-climbing* yields a local optimum, if terminating) were proved within the KIV system. For the termination proof it is shown that the number of nodes in the input set that are strictly better than the selected node decreases in each iteration. The proof of partial correctness was done by induction on the number of iterations necessary to terminate. Necessary for succeeding with these proofs was the introduction of the requirement that ensures non-emptiness of the input of the method. The introduction of this requirement was a result of our proofs. In addition, several errors in our original specification were found concerning the case when the inference actions *init* and *generate* do not provide output (the *start* predicate is not defined for the input of the PSM or the current element of *node* has no neighbours).

Figure 11 is a screen dump of the KIV system when proving the termination of the PSM *hill-climbing*. 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. a open premise (leaf) of the (partial) proof tree (here goal number 2). 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.

6.3. Verifying Assumptions and Detecting Hidden Assumptions

The obligations for the *total-module* (*assumption*$_1$) and the *better-module* (*assumption*$_2$) correspond to proof obligation (PO-iv) from section 2.2. It has to be proven whether the competence of the PSM fulfils the task definition by introducing assumptions. The proof obligation for *assumption*$_1$ states that total connectedness of the graph (each node is directly linked with each node) implies that the local optimum is also a global one. Whereas this obligation has a rather trivial proof in KIV, the attempt at proving the obligation for the *better-module* failed, i.e. ends up with a remaining open premise (see Figure 12, the sequent window shows the unprovable premise). This result was a bit surprising for us. Supported by the graphical interface of KIV, interactively analysing the partial proof provides the explanation for the unexpected failure: the formula we wanted to prove was not true at all, i.e. the assumption that every node except the global maximum is connected to a better node, is not sufficient for the PSM *hill-climbing* to find a global optimum.

The analysis of the partial proof gives some hint for the construction of possible counter examples, but also a hint on how the assertion could be repaired. Here it leads straightforwardly to a strengthening of the assumption: the "better node" has to be member of the input of *hill-climbing* (cf. the sequent in Figure 12). That is, the assumption over the domain knowledge that is provided by the relationship *graph* must be supplemented by an assumption over the actual input of the PSM. Only if a better neighbour as required by $assumption_2$ is an element of the actual *input* the equivalence can be proven.

$$\begin{array}{l} assumption_2\text{'}: \\ \quad \forall\, x\, (node(x) \rightarrow \\ \qquad \exists y\, (\mathbf{input(y)} \wedge graph(x,y) \wedge pref(h(x),h(y)) \wedge \\ \qquad h(x) \neq h(y)) \vee \forall z\, (node(z) \rightarrow pref(h(z),h(x)))) \end{array}$$

Thus, the KIV system was used as an explorative tool to detect a (further) hidden assumption. To prove the corrected assertion in KIV, the partial proof of the original assertion can be reused, such that no further interaction is required.

7 Conclusion and Future Work

In the paper, we introduce a formal framework for specifying and verifying knowledge-based systems. One can specify tasks, problem-solving methods, domain knowledge and can verify whether the assumed relationships between them are guaranteed, i.e., which assumptions are necessary for establishing these relationships. Besides verification, the interactive verification tool KIV can be used to explore hidden assumptions necessary to relate the competence of a problem-solving method to the task definition.

As an experiment we have applied the KIV system to the *hill-climbing* example. The lessons learned from this case-study can be summarized as follows:

- Current software verification techniques can be valuable for the development of reliable KBSs; especially, for verification of PSMs and for detection of hidden assumptions. With interactive deduction system the proof obligations seem feasible. In the *hill-climbing* example the proofs constructed with KIV took a total of 73 proof steps, 50 of which were chosen automatically by heuristics.
- Formal development of KBSs requires tool support for modularisation of specifications and programs and for constructing, analysing, and reusing proofs. The KIV system offers these facilities.
- Even the small toy example provides a strong argument for formal methods. An additional hidden assumption and several specification errors and necessary modifications in the definition of the PSM were detected.

For a general assessment of our results further experience, especially in larger more realistic case studies has to be gathered (e.g., the VT task and the problem-solving method propose & revise [32]).

We also had to overcome some differences in our representation language and the language provided by KIV. Currently, KIV represent the state of a reasoning process by value assignments of dynamic variables whereas MLPM applies the state as algebra approach. That is, a state is represented by an algebra and state changes are expressed as changes of this algebra (compare [22], [23]). [41] provides an extension of KIV for a subset of *evolving algebras* [23] that makes a step in overcoming this difference. Still, the grainsize of the state transitions as they can defined in MLPM and the extended version of KIV differ. The latter only provides pointwise updates of functions whereas MLPM enables the complete update of a predicate as an elementary transition. A point for future work concerns the adaption of the KIV system, specialized to development and verification of KBSs. That includes extension of the current representation formalism and the development of appropriate proof tactics and heuristics.

Acknowledgement. We thank Richard Benjamins, Stefan Deckert, Frank van Harmelen, Maarten van Someren, Remco Straatman and Annette ten Teije for helpful comments on early drafts of the paper. Thanks to Martin Giese for carrying out the proofs in the KIV system.

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, Berlin, 1993.

[2] R. Benjamins: *Problem-Solving Methods for Diagnosis*, PhD Thesis, University of Amsterdam, Amsterdam, the Netherlands, 1993.

[3] R. Benjamins, D. Fensel, and R. Straatman: Assumptions of Problem-Solving Methods and Their Role in Knowledge Engineering. In *Proceedings of the 12. European Conference on Artificial Intelligence (ECAI-96)*, Budapest, August 12-16, 1996.

[4] M. Bidoit, H.-J. Kreowski, P. Lescane, F. Orejas and D. Sannella (eds.): *Algebraic System Specification and Development*, Lecture Notes in Computer Science (LNCS), no 501, Springer-Verlag, Berlin, 1991.

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

[6] A. Bundy (ed.): Catalogue of Artificial Intelligence Techniques, 3rd ed., Springer-Verlag, Berlin, 1990.

[7] T. Bylander: Complexity Results for Planning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney, Australia, August 1991.

[8] T. Bylander, D. Allemang, M. C. Tanner, and J. R. Josephson: The Computational Complexity of Abduction, Artificial Intelligence, 49, pages 25-60, 1991.

[9] R. Davis: Diagnostic Reasoning Based on Structure and Behavior, *Artificial Intelligence*, 24: 347-410, 1984.

[10] R. Davis and W. Hamscher: Model-based Reasoning: troubleshooting. W. Hamscher et al. (eds.), Readings in Model-based Diagnosis, Morgan Kaufmann Publ., San Mateo, 1992.

[11] H. Ehrig and B. Mahr: *Fundamentals of Algebraic Specifications 1*, Equations and Initial Semantics, EATCS Monographs on Theoretical Computer Science, vol 21, Springer-Verlag, Berlin, 1985

[12] H. Eriksson, Y. Shahar, S. W. Tu, A. R. Puerta, and M. A. Musen: Task Modeling with Reusable Problem-Solving Methods. To appear in *Artificial Intelligence*.

[13] D. Fensel: Assumptions and Limitations of a Problem-Solving Method: A Case Study. In *Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´95)*, Banff, Canada, February 26th - February 3th, 1995.

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

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

[16] D. Fensel, H. Eriksson, M. A. Musen, and R. Studer: Developing Problem-Solving by Introducing Ontological Commitments, *International Journal of Expert Systems: Research & Applications*, 9(3), 1996.

[17] 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.

[18] 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.

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

[20] 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, Berlin, 1995.

[21] M. Gordon, R. Milner and C. Wadsworth: Edinburgh LCF: A Mechanized Logic of Computation, *Lecture Notes in Computer Science (LNCS)*, no 78, Springer-Verlag, Berlin, 1979.

[22] R. Groenboom and G.R. Renardel de Lavalette: Reasoning about Dynamic Features in Specification Languages. In D.J. Andrews et al. (eds.), *Proceedings of Workshop in Semantics of Specification Languages*, October 1993, Utrecht, Springer Verlag, Berlin, 1994.

[23] Y. Gurevich: Evolving Algebras 1993: Lipari Guide. In E.B. Börger (ed.), *Specification and Validation Methods*, Oxford University Press, 1994.

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

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

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

[27] G. van Heijst and A. Anjewierden: Four Propositions Concerning the Specification of Problem Solving Methods. In *Sublementary Proceedings of the 9th European Knowledge Acquisition Workshop EKAW-96*, Nottingham, England, 14.-17. Mai 1996.

[28] Y. Ledru: Proof-based Development of Specifications with KIDS/VDM. In M. Naftalin et al. (eds.), *Formal Methods Europe, FME'94: Industrial Benefit of Formal Methods*, Lecture Notes in Computer Science (LNCS), no 873, Springer-Verlag, Berlin, 1994.

[29] M. R. Lowry: Algorithmic Synthesis through Problem Reformulation. In *Proceedings of the 6th National Conference on AI (AAAI'87)*, Seattle, Washington, July 13-17, 1987.

[30] B. Nebel: Artificial intelligence: A Computational Perspective. To appear in G. Brewka (ed.), *Essentials in Knowledge Representation*.

[31] C. Pierret-Golbreich and X. Talon: An Algebraic Specification of the Dynamic Behaviour of Knowledge-Based Systems, *The Knowledge Engineering Review*, submitted 1995.

[32] K. Poeck, D. Fensel, D. Landes, and J. Angele: Combining KARL And CRLM For Designing Vertical Transportation Systems, *The International Journal of Human-Computer Studies*, to appear, 1996.

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

[34] J.R. Quinlan: Learning Efficient Classification Procedures and Their Application to Chess End Games. In R.S. Michalski et al. (eds.), *Machine Learning. An Artificial Intelligence Approach, vol.1,* Springer-Verlag, Berlin, 1984, pp. 463-482.

[35] W. Reif: Correctness of Generic Modules. In Nerode & Taitslin (eds.), *Symposium on Logical Foundations of Computer Science,* Lecture Notes on Computer Science (LNCS), no 620, Springer-Verlag, Berlin, 1992.

[36] 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, Berlin, 1992.

[37] 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*, Lecture Notes in Computer Science (LNCS), no 1009, Springer-Verlag, Berlin, 1995.

[38] W. Reif and K. Stenzel: Reuse of Proofs in Software Verification. In Shyamasundar (ed.), Foundation of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science (LNCS), no 761, Springer-Verlag, Berlin, 1993.

[39] D. R. Smith: KIDS: A Semiautomatic Program Development System, *IEEE Transactions on Software Engineering*, 16(9):1024--1043, 1990.

[40] D. R. Smith and M. R. Lowry: Algorithm Theories and Design Tactics, Science of Computer Programming, 14:305-321, 1990.

[41] A. Schönegge: Extending Dynamic Logic for Reasoning about Evolving Algebras, research report 49/95, Institut für Logik, Komplexität und Deduktionssysteme, University of Karlsruhe, 1995.

[42] 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.

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

[44] J. Top and H. Akkermans: Tasks and Ontologies in Engineering Modeling, *International Journal of Human-Computer Studies*, 41:585—617, 1994.

[45] J. Treur: Temporal Semantics of Meta-Level Architectures for Dynamic Control of Reasoning. In L. Fribourg et al. (eds.), *Logic Program Synthesis and Transformation - Meta Programming in Logic, Proceedings of the 4th International Workshops, LOPSTER-94 and META-94*, Pisa, Italy, June 20-21, 1994, Lecture Notes in Computer Science, no 883, Springer Verlag-Berlin, 1994.

[46] 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.

[47] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma (1991): Specification Styles in Distributed System Design and Verification, *Theoretical Computer Science*, 98:179-206.

[48] B. Wielinga, J. M. Akkermans, and A. TH. Schreiber: A Formal Analysis of Parametric Design Problem Solving. In B. R. Gaines and M. A. Musen (eds.): *Proceedings of the 8th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop* (*KAW-95*), vol II, pp. 31/1—37/15, Alberta, Canada, 1995.

[49] B. J. Wielinga and A. Th. Schreiber: Conceptual Modelling of Large Reusable Knowledge Bases. In K. von Luck and H. Marburger (eds.): *Management and Processing of Complex Data Structures*, Springer-Verlag, Lecture Notes in Computer Science, no 777, pages 181—200, Berlin, Germany, 1994.

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