The Unified Problem-solving Method Development Language UPML

Dieter Fensel[1], Enrico Motta[3], V. Richard Benjamins[4], Monica Crubezy[7], Stefan Decker[2], Mauro Gaspari[5], Rix Groenboom[6], William Grosso[7], Frank van Harmelen[1], Mark Musen[7], Enric Plaza[8], Guus Schreiber[4], Rudi Studer[2], and Bob Wielinga[4]

[1] Division of Mathematics & Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, NL, {dieter,frankh}@cs.vu.nl

[2] University of Karlsruhe, Institute AIFB, D-76128 Karlsruhe, Germany, {dfe,sde,studer}@aifb.uni-karlsruhe.de

[3] The Open University, Knowledge Media Institute, Walton Hall, MK7 6AA, Milton Keynes, United Kingdom, e.motta@open.ac.uk

[4] University of Amsterdam, Department of Social Science Informatics (SWI), Roeterstraat 15, NL-1018 WB Amsterdam, The Netherlands, {richard,schreiber,wielinga}@swi.psy.uva.nl

[5] Department of Computer Science, University of Bologna, Italy gaspari@cs.unibo.it

[6] University of Groningen, Department of Computer Science, P.O. Box 800, NL-9700 AV Groningen, NL, rix@cs.rug.nl

[7] Knowledge Modeling Group at Stanford Medical Informatics, Stanford University, 251 Campus Drive, MSOB X-215, Stanford, California, USA crubezy@SMI.Stanford.Edu, grosso@SMI.Stanford.Edu, musen@SMI.Stanford.Edu

[8] Spanish Council of Scientific Research (CSIC), Artificial Intelligence Research Institute (IIIA), Campus UAB, 08193 Bellaterra, Barcelona, Spain, enric@iiia.csic.es

Abstract. Problem-solving methods provide reusable architectures and components for implementing the reasoning part of knowledge-based systems. The *Unified Problem-solving Method description Language UPML* has been developed to describe and implement such architectures and components to facilitate their semiautomatic reuse and adaptation. In a nutshell, UPML is a framework for developing knowledge-intensive reasoning systems based on libraries of generic problem-solving components. The paper describes the components and adapters, architectural constraints, development guidelines, and tools provided by UPML. UPML is developed as part of the IBROW project; which provides an internet-based brokering service for reusing problem-solving methods.

1 Introduction

Knowledge-based systems are computer systems that deal with complex problems by making use of knowledge. This knowledge may be acquired from humans or automatically derived using abductive, deductive, and inductive techniques. This knowledge is mainly represented declaratively rather than encoded using complex algorithms. This declarative representation of knowledge economizes the process of developing and maintaining these systems and improves their understandability. Therefore, knowledge-based systems originally used simple and generic inference mechanisms to infer outputs for cases provided. Inference engines, like unification, forward or backward resolution, and inheritance, dealt with the dynamic part of deriving new information. However, human experts can exploit knowledge about the dynamics of the problem-solving *process* and such knowledge is required to enable problem-solving in practice and not only in principle. [Clancey, 1983] provided several examples where knowledge engineers implicitly encoded control knowledge by ordering production rules and premises of these rules which, together with the generic inference engine, delivered the desired dynamic behaviour. Making this knowledge explicit and regarding it as an important part of the entire knowledge contained in a knowledge-based system is the rationale that underlies *problem-solving methods PSMs* (cf. [Stefik, 1995], [Benjamins & Fensel, 1998], [Benjamins & Shadbolt, 1998], [Fensel, 2000]). Problem-solving methods refine the generic inference engines mentioned above to allow a more direct control of the reasoning process. Problem-solving methods describe this control knowledge independent of the application domain and thus enable the reuse of this strategical knowledge for different domains and applications. Finally, problem-solving methods abstract from a specific representation formalism, in contrast to the general inference engines that rely on a specific representation of the knowledge. Problem-solving methods (PSMs) decompose the reasoning task of a KBS in a number of subtasks and inference actions that are connected by knowledge roles. Therefore PSMs are a special type

of software architecture ([Shaw & Garlan, 1996]): *software architecture* for describing the *reasoning* part of KBSs.

Several libraries of problem solving methods are now available (c.f., [Marcus, 1988]; [Chandrasekaran et al., 1992]; [Puppe, 1993]; [Breuker & Van de Velde, 1994]; [Benjamins, 1995]; [Musen 1998]; [Motta, 1999]) and a number of problem-solving method specification languages have been proposed, ranging from informal notations (e.g. CML [Schreiber et al., 1994]) to formal modeling languages (see [Fensel & van Harmelen, 1994], [Fensel, 1995] for summaries).

The IBROW project[1] [Benjamins et al., 1998], [Fensel & Benjamins, 1998a] was established with the aim of enabling semi-automatic reuse of PSMs. This reuse is provided by integrating libraries in an internet-based environment. A broker is provided that selects and combines PSMs of different libraries. A software engineer interacts with a broker that supports him in this configuration process. As a consequence, a description language for these reasoning components (i.e., PSMs) must provide comprehensible high-level descriptions with substantiated formal means to allow automated support by the broker. Therefore, we developed the *Unified Problem-Solving Method description Language UPML* (cf. [Fensel et al., 1999a], [Fensel et al., 1999b], [Fensel et al., 1999c]).

UPML is an architectural description language specialized for a specific type of systems providing *components*, *adapters* and a configuration for connecting the components be using the adapters (called *architectural constraints*). Finally *design guidelines* provide ways to develop a system constructed from the components and connectors that satisfies the constraints. UPML brings together the experiences that have been made using the numerous modeling languages developed for knowledge-based systems. Examples are CML [Schreiber et al., 1994], DESIRE [van Langevelde et al., 1993], KARL [Fensel et al., 1998b], MCL [Fensel et al., 1998a], $(ML)^2$ [van Harmelen & Balder, 1992], MODEL-K [Karbach & Voß, 1992], MoMo [Voß & Voß, 1993], Noos [Arcos & Plaza, 1996]. In terms of software architecture languages (cf. [Clements, 1996]), UPML is a language specialized for a specific type of systems.

In the meantime, UPML has been adopted by a large number of research groups and has, for example, been used to specify the design library of [Motta, 1999] at Milton Keynes (see [Motta et al., 1998]) and parts of the PSM library at Stanford (cf. [Musen 1998]). The former library is fully available on the web, allowing only the configuration of design problem solvers. It has also been applied to Office Allocation, Elevator Configuration,

1. IBROW started with a preliminary phase under the 4th European Framework and has become a full-fledged Information Society Technologies (IST) project under the 5th European Framework Program since January 2000. Results of its initial phase are described in [Benjamins et al., 1999], [Benjamins & Fensel, 1998], and [Fensel et al., 1999b]. Project partners are the University of Amsterdam; the Open University, Milton Keynes, England; the Spanish Council of Scientific Research (IIIA) in Barcelona, Spain; the Institute AIFB, University of Karlsruhe, Germany: Stanford University, US: Intelligent Software Components S. A., Spain; and the Vrije Universiteit Amsterdam.
http://www.swi.psy.uva.nl/projects/IBROW3/home.html

Engineering Design (sliding bearing design, truck cab design, casting technology design) and Health-care. Hopefully it will become a widely used standard approach for describing and developing knowledge-based systems with reusable components. This will increase overall productivity in developing systems and will simplify exchange of work between different groups. Presenting UPML, its main ideas and concepts is the purpose of this paper.

The content of the paper is organized as follows. In Section 2, we will sketch the architectural framework that is provided by UPML. Sections 3 discusses the different elements of our architecture: ontologies, tasks, domain models, problem-solving methods, refiners, and bridges. Section 4 introduces the architectural constraints that ensure well-defined architectures, the development guidelines of UPML, and the tool environment which accompanies UPML. Finally, we bring our conclusions, a discussion of related work, and an outlook in Section 5.

2 The Architecture of UPML

UPML unifies and generalizes the conceptual models for describing knowledge-based systems that were developed by several approaches in knowledge engineering. In particular the *model of expertise* was developed in CommonKADS [Schreiber et al., 1994] for distinguishing different knowledge types and describing a knowledge-based system at a conceptual level. However, its goal was *not* to describe the different software components of which a knowledge-based system is built. In consequence, we had to (1) decouple different element of this model, (2) encapsulate these different elements and (3) explicitly model their interactions. Therefore, in [Fensel & Groenboom, 1997] and [Fensel & Groenboom, 1999] we introduced an architecture for knowledge-based systems as a modification and extension of the CommonKADS model of expertise. UPML further develops this architecture and introduces additional elements in it.

The UPML architecture for describing a knowledge-based system consists of six different elements (see Figure 1): a *task* that defines the problem that should be solved by the knowledge-based system, a *problem-solving method* that defines the reasoning process of a knowledge-based system, and a *domain model* that describes the domain knowledge of the knowledge-based system. Each of these elements is described independently to enable the reuse of task descriptions in different domains, the reuse of problem-solving methods for different tasks and domains, and the reuse of domain knowledge for different tasks and problem-solving methods. *Ontologies* (cf. [Gruber, 1993], [Mizoguchi et al., 1995]) provide the terminology used in tasks, problem-solving methods, and domain definitions. Again this separation enables knowledge sharing and reuse. For example, different tasks or problem-solving methods can share parts of the same vocabulary and definitions. A fifth element of a specification of a knowledge-based system are *adapters* which are necessary to adjust the other (reusable) parts to each other and to the specific application problem.[2] UPML provides two types of adapters: *bridges* and *refiners*. Bridges explicitly model the relationships between two different parts of an architecture, e.g. between domain and task or task and problem-solving method. Refiners can be used to express the stepwise adaptation of other elements of a specification, e.g. a task is refined or a problem-solving method is refined ([Fensel, 1997], [Fensel & Motta, 1998]). Generic problem-solving methods and tasks can be refined to produce more specific ones by applying a sequence of refiners to them. Again, separating generic and specific parts of a reasoning process enhances reusability. The main distinction between bridges and refiners is that bridges change the input and output of components making them fit together whereas refiners may change internal details like subtasks of a problem solving method. In the following we will see the use of a bridge to connect the problem-solving method *hill-climbing* with the task *diagnostic problem solving* (i.e., we model a task-specific refinement of a problem-solving method via a bridge) and the use of a refiner to specialize a generic search method until it

[2] Adapters correspond to the *transformation operators* of [Van de Velde, 1994].

becomes *hill-climbing* (i.e., we use a refiner to specialize the algorithmic paradigm of a method).

The different parts of the architecture will be discussed in the following section.

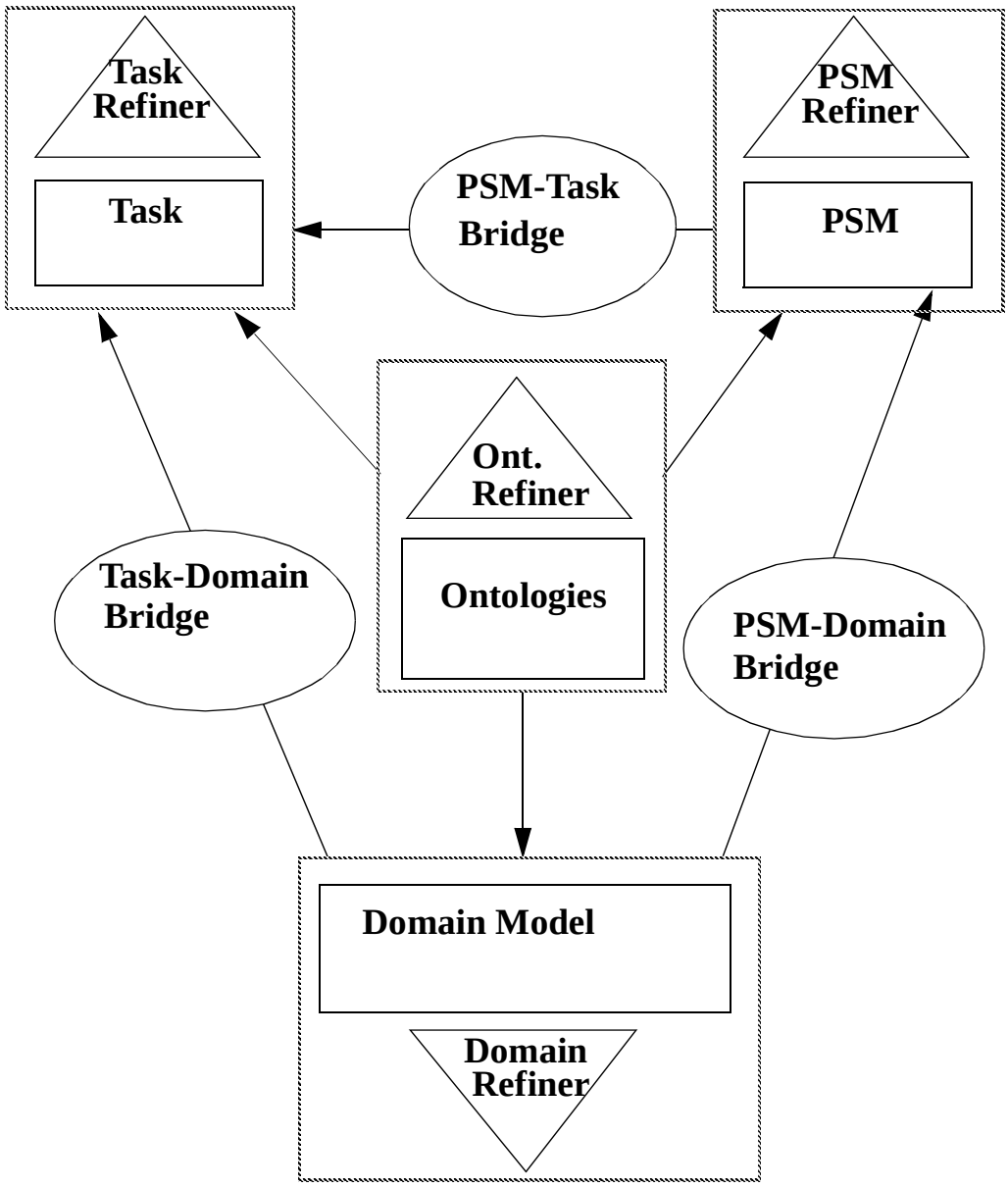

Fig. 1 The UPML architecture for knowledge-based systems.

3 The Different Elements of the UPML Architecture

In the following, we discuss the different elements of the UPML architecture and how they are connected. First we introduce the graphical notation of UPML and survey the structure of an entire example in Section 3.1. Then, we introduce the different elements of a specification. We introduce ontologies in Section 3.2. Ontologies provide the definition of signatures and axioms that are used by the other parts of the architecture. Section 3.3 introduces tasks that define the problems solved by knowledge-based systems. Section 3.4 provides the definition of domain models and Section 3.5 shows how tasks and domain models are connected via bridges. Section 3.6 introduces the core of UPML, the specification frame for describing problem-solving methods. Section 3.7 shows how such descriptions of problem-solving methods can be refined to provide a structured way to develop and describe problem-solving methods and their different variants. Section 3.8 provides the bridges between problem-solving methods and tasks and domains. Finally, the overall structure of UPML is defined in Section 3.9.

3.1 Graphical Notations of UPML

The graphical notation of UPML can be used to clarify the overall structure of the architecture of a system. Figure 2 illustrates our running example. We have a task *complete and parsimonious diagnoses*, a domain model *anesthesiology*, and a problem-solving method *search* defining five subtasks. This generic method becomes refined to *hill-climbing* by refining two of its subtasks. There are three bridges that link task, domain, and method; and four ontologies provide the terminology for the different specification elements.

The example may also clarify the relationship between subtasking and the refinement of a problem-solving method. Subtasking corresponds to the **part-of** construct of knowledge representation formalisms. A problem-solving method decomposes its entire reasoning tasks into subtasks. The refinement of problem-solving methods as introduced in [Fensel, 1997] corresponds to the **is-a** relationship of knowledge representation formalisms. The example we will use is hill-climbing, a subclass of general search methods. Therefore it specializes some of its arguments (i.e., *Initialize* and *Update Nodes*, and the competence description).

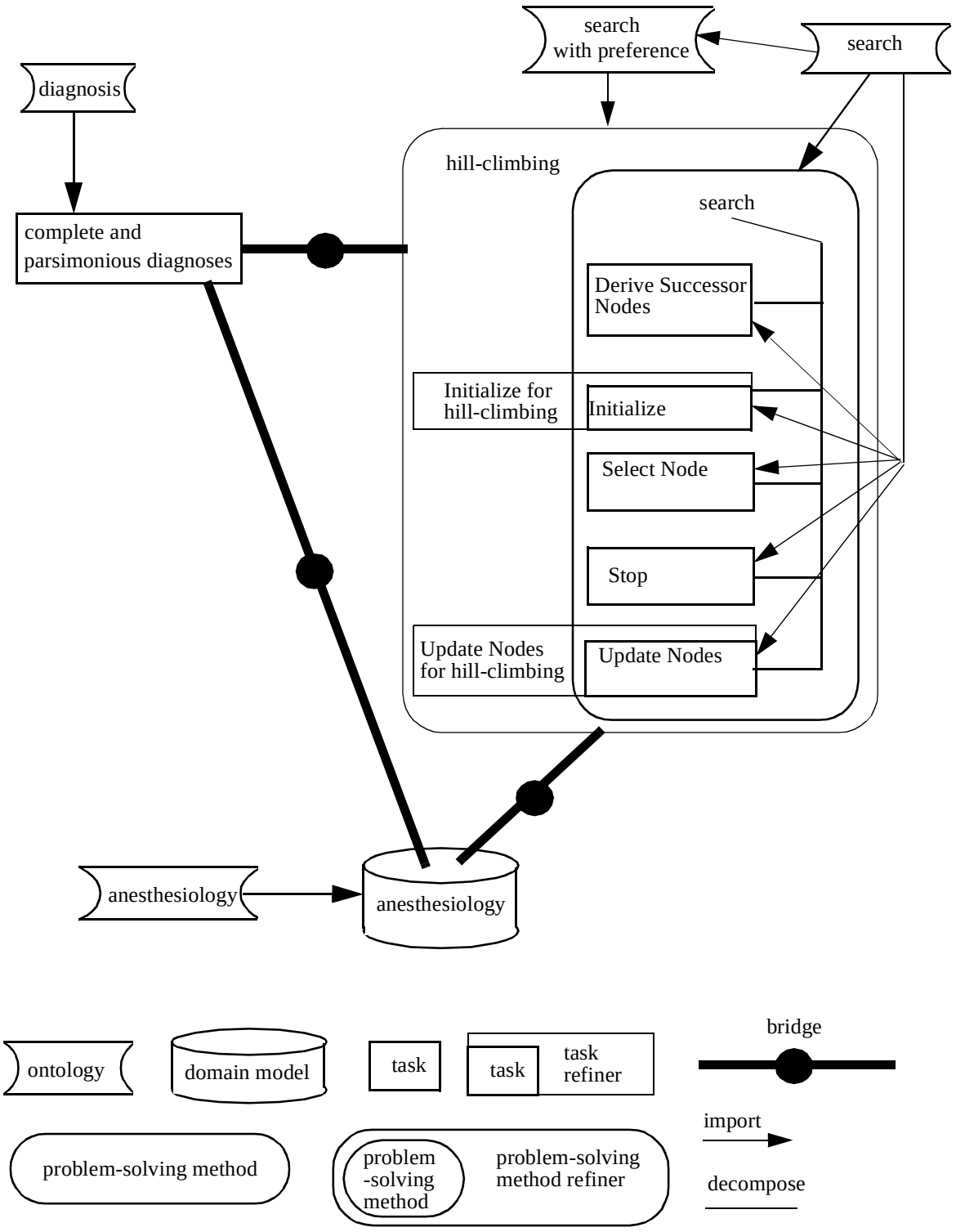

Fig. 2 The graphical representation of the example.

3.2 Ontology

An ontology provides "an explicit specification of a conceptualization" [Gruber, 1993], which can be shared by multiple reasoning components communicating during a problem solving process (cf. [Top & Akkermans, 1994], [Mizoguchi et al., 1995]). In our framework ontologies are used to define the terminology and properties used to define tasks, problem-solving methods, and domain models. UPML does not commit itself to a specific language style for defining an ontology. However, we provide two styles as possible ways for specifying signatures and axioms. First, we provide logic with sorts close to the specification style of $(ML)^2$ and MCL [Fensel et al., 1998a]. Second, we provide a frame-based representation using concepts and attributes as it was proposed by CML and KARL. We provide these two different language styles because these languages cover different needs.

1 The simple and therefore „elegant" sorted logic variant consists of first-order logic, some reified second order constructs (i.e., syntactically second-order but semantically within first-order) and a simple sort concept.

2 Frame-logic [Kifer et al., 1995] is a powerful language for rich ontological modeling. Its syntax is more complex (i.e. richer) providing modeling primitives at a higher epistemological level. Objects, classes, attributes, domain and range restrictions, multiple attribute inheritance etc. are provided. Again, these modeling constructs are integrated into a semantical first-order framework using reification. Frame logic provides more direct support in modeling ontologies; however, its syntax is more complex and less well known than the syntax of standard predicate logic.

At present, UPML standardizes the overall architecture of a system but does not fix the language that is used to define the elementary elements of this architecture. The development of a language called OIL is under developed that will be used for this purpose.

The next section will provide several examples for ontologies, however, we will restrict ourselves to the more simpler style of sorted logic. We will use the following naming conventions. Sort names are words beginning with an uppercase, constant, function, predicate, and variable names begin with a lowercase. Non-quantified variables are implicitly all-quantified.

3.3 Task

A *task ontology* specifies a theory, i.e. a signature and a logical characterization of the signature elements, that is used to define tasks (i.e., a problem type). An example of a task

ontology which is used to provide the elements for defining a diagnostic problem is illustrated in Figure 3 . The ontology introduces two elementary sorts *Finding* and *Hypothesis* that will be grounded later in a domain model. The former describes phenomenons and the latter describes possible explanations. The two constructed sorts *Findings* and *Hypotheses* are sets of elements of these elementary sorts. The function *explain* connects findings with hypotheses. Domain knowledge must further characterize this function. Three predicates are provided. An order < is used to define optimality (i.e., *parsimonity*) of hypotheses and finally, completeness, which ensures that a hypothesis explains a set of findings.

The description of a *task* specifies goals that should be achieved in order to solve a given problem. A second part of a task specification is the definition of assumptions about domain knowledge and preconditions on the input. These parts establish the definition of a problem that is to be solved by the knowledge-based system. In contrast to most approaches in software engineering this problem definition is kept domain independent, which enables the reuse of generic problem definitions for different applications. A second distinguishing feature is the distinction between preconditions on input and assumptions about knowledge.

ontology *diagnoses*
pragmatics
The task ontology defines diagnoses for a set of observations;
Dieter Fensel;
May 2, 1998;
D. Fensel: Understanding, Developing and Reusing Problem-Solving Methods. Habilitation, Faculty of Economic Science, University of Karlsruhe, 1998;
signature
elementary sorts
Finding; Hypothesis
constructed sorts
Findings : **set of** *Finding; Hypotheses* : **set of** *Hypothesis*
constants
observations : *Findings; diagnosis* : *Hypotheses*
functions
explain: *Hypotheses* → *Findings*
predicates
< : *Hypotheses* x *Hypotheses;*
complete: *Hypotheses* x *Findings;*
parsimonious: *Hypotheses*
axioms
A hypothesis is complete for some findings iff it explains all of them.
$complete(H,F) \leftrightarrow explain(H) = F;$
A hypothesis is parsimonious iff there is no smaller hypothesis with larger or equal explanatory power.
$parsimonious(H) \leftrightarrow \neg\exists H' (H' < H \wedge explain(H) \subseteq explain(H'))$

Fig. 3 A task ontology for diagnostic problems.

In an abstract sense, both can be viewed as input. However, distinguishing case data, which are processed (i.e., input), from knowledge, which is used to define the goal, is a characteristic feature of *knowledge*-based systems. Preconditions are conditions on dynamic inputs. Assumptions are conditions on knowledge consulted by the reasoner but not transformed. Often, assumptions can be checked in advance during the system building process, preconditions cannot. They rather restrict the valid inputs. Input and output role definitions provide the terms that refer to the input and output of the task. These names must be defined in the signature definition of the task (i.e., either in the imported ontology or in the auxiliary terminology). The assumptions ensure (together with the axioms of the ontology) that the task can always be solved for permissable input (input for which the preconditions hold). For example, when the goal is to find a global optimum, then the assumptions have to ensure that such a global optimum exists (i.e., that the preference relation is non-cyclic). Whether we write an assumption or not as an axiom of an ontology is a modeling decision. If it is highly reusable and only weakly coupled with the specific goals of the task then it should be written as an axiom of the ontology imported by the task. Otherwise, it is more useful to directly write it as an assumption of the specific task.

A task definition imports ontologies that define its vocabulary and other tasks which it refines. The latter enable hierarchical structuring of task specifications. For example, parametric design can be defined as a refinement of design (cf. [Fensel & Motta, 1998]).

An example of a task specification is given in Fig. 4. The goal specifies a complete and parsimonious (i.e., minimal) diagnosis. It is guaranteed that such a diagnosis exists if the domain knowledge can provide a complete diagnosis for each (non-empty) input. We are able to guarantee the existence of a complete and parsimonious explanation if we can guarantee that < is non-reflexive and transitive and we assume the finiteness of the set of hypotheses.

3.4 Domain Model

The *domain knowledge* is necessary to define the task in the given application domain and to carry out the inference steps of the chosen problem-solving method. Properties and assumptions differ in their way we assume their truth. Properties can be derived from the domain knowledge, whereas assumptions have to be assumed to be true, i.e. they have not been proven or they cannot be proven. Assumptions capture the implicit and explicit assumptions made while building a domain model of the real world. *Properties* and *assumptions* are both used to characterize the domain knowledge. They are the counterparts of the requirements on domain knowledge introduced by the other parts of a specification. Some of these requirements may be directly inferred from the domain knowledge (and are therefore properties of it), whereas others can only be derived by introducing assumptions about the environment of the system and the actual input. For example, typical external

task *complete and parsimonious diagnoses*
pragmatics
The task asks for a complete and minimal diagnosis;
Dieter Fensel;
May 2, 1998;
D. Fensel: Understanding, Developing and Reusing Problem-Solving Methods. Habilitation, Fakulty of Economic Science, University of Karlsruhe, 1998;
ontology
diagnoses
specification
roles
input *observations*; **output** *diagnosis*
goal
task(**input** *observations*; **output** *diagnosis*) $\leftrightarrow$
complete(*diagnosis*, *observations*) $\wedge$ *parsimonious*(*diagnosis*)
preconditions
observations $\neq \varnothing$
assumptions
If we receive input there must be a complete hypothesis.
observations $\neq \varnothing \rightarrow \exists H$ *complete*(H, *observations*);
Nonreflexivity of <.
$\neg (H < H)$;
Transitivity of <.
$(H < H') \wedge (H' < H'') \rightarrow (H < H'')$;
Finiteness of H.
Finite(H)

Fig. 4 The task specification of a diagnostic task.

assumptions in model-based diagnosis are: the fault model is complete (no fault appears that is not captured by the model), the behavioral description of faults is complete (all fault behaviors of the components are modeled), the behavioral discrepancy that is provided as input is not the result of a measurement fault, etc. (cf. [Fensel & Benjamins, 1998b]).

Our framework for defining a *domain model* provides three elements: a meta-level characterization of properties of the domain model, assumptions of the domain model, and the domain knowledge itself. Again, ontologies are an externalized means for defining the terminology. Figure 5 provides an ontology for *Anesthesiology*. The medical domain model we have chosen for our example is a subset of a large case-study in the formalization of domain knowledge for an anesthesiological support system. Figure 5 provides the sort *cause* and *manifestation* and a *causal relation* between them.

The support system of our running example is designed to diagnose a (limited) number of hypotheses based on real-time acquired data. This data is obtained from the medical database system Carola [de Geus & Rotterdam, 1992] which performs on-line logging of measurements. In this domain we deal with abstract notions, derived from interpretations of

ontology *anesthesiology*
pragmatics
The domain ontology provides faults for malfunctions for *anesthesiology*;
Rix Groenboom;
[Groenboom, 1997];
http://www.medical-library.org
signature
elementary sorts
Cause; Manifestation
constructed sorts
Causes **set of** *Cause*
constants
highHeartRate, highPartm, toolowCOP, wakingUp : *Manifestation;*
centralization, pain, edema, lowAnesthesia : *Cause*
predicates
cause relation : *Cause* x *Manifestation*

Fig. 5 A domain ontology of *anesthesiology.*

measurements. The exact meaning of *HighPartm* (which stands for a High mean arterial blood-pressure) is defined elsewhere in the formal domain model (see [Groenboom, 1997]). Another technical term is *ToolowCOP*, which refers to a too low Cellular Oxygen Pressure. Figure 6 provides an example for a domain model definition. The properties ensure that there is a cause for each manifestation and that causes do not conflict. That is, different causes do not lead to an inconsistent set of manifestations. In our domain this is guaranteed by the fact, that we do not have knowledge about negative evidence (i.e., a symptom may rule out an explanation). Assuming more causes only leads to a larger set of symptoms that can be explained. The *complete-fault-knowledge assumption* guarantees that there are no other unknown faults, like hidden diseases for example. Only under this assumption can we deductively infer causes from observed symptoms. However, it is a critical assumption when relating the output of our system to the actual problem and domain (cf. [Fensel & Benjamins, 1998b]).[3]

3.5 Task-Domain Bridge

An *adapter* maps the different terminologies of task definition, problem-solving method and domain model to each other. Moreover, it gives further assumptions that are needed to relate the competence of a problem-solving method with the functionality given by the task definition. The task, the problem-solving method, and the domain model can be described independently and selected from libraries because adapters relate these parts of a

3. Note that we do not assume complete knowledge of symptoms.

domain model *anesthesiology*

pragmatics

The domain provides faults for malfunctions for *anesthesiology*;
Rix Groenboom;
[Groenboom, 1997];
http://www.medical-library.com

ontology

anesthesiology

properties

there is a cause for each manifestation

$s \exists h$ *cause relation*(h, s);

the fault knowledge is monotonic

$H \subseteq H' \rightarrow$

$\{s \mid h \in H \wedge$ *cause relation*$(h,s)\} \subseteq \{s \mid h \in H' \wedge$ *cause relation*$(h,s)\}$

assumptions

complete fault knowledge:
We know all possible causes of the provided manifestations.

$h =$ *lowAnesthesia* $\wedge s =$ *wakingUp* $\vee$
$h =$ *centralization* $\wedge s =$ *highPartm* $\vee$
$h =$ *pain* $\wedge s =$ *highPartm* $\vee$
$h =$ *pain* $\wedge s =$ *wakingUp* $\vee$
$h =$ *pain* $\wedge s =$ *highHeartRate* $\vee$
$h =$ *edema* $\wedge s =$ *highHeartRate* $\vee$
$h =$ *edema* $\wedge s =$ *oolowCOP* $\vee$
$\neg$*cause relation*(h,s)

domain knowledge

cause relation(*lowAnesthesia*,*wakingUp*);
cause relation(*centralization*,*highPartm*);
cause relation(*pain*,*highPartm*);
cause relation(*pain*,*wakingUp*);
cause relation(*pain*,*highHeartRate*);
cause relation(*edema*,*highHeartRate*);
cause relation(*edema*,*toolowCOP*)

Fig. 6 The domain model *anesthesiology.*

specification to each other and establish their relationship in a way that meets the specific application problem. The adapter is responsible for consistently combining and adapting the three different components to the specific aspects of the given application—as they are designed to be reusable they need to abstract from specific aspects of application problems. In addition, adapters introduce *assumptions* necessary for closing the gap between a problem definition (task) and the competence of a problem-solving method. As we will later see, adapters also will be used to express the *refinement* of tasks, problem-solving methods, and domain models. In this case, adapters also specify reusable knowledge. They not only provide an application-specific glue, but also specify refinements expressing a problem type. Adapters can be piled up to express the stepwise refinement of problem-solving methods.

One specific adapter type of UPML is the *task-domain bridge*. This bridge type instantiates tasks for specific domains and therefore enables that they be described domain-independently and reusably. Mapping of different terminologies can either be achieved directly by renaming or indirectly by linking their properties via mapping axioms. A bridge may be forced to state assumptions about domain knowledge to ensure that the mapped terminologies respect the definitions of a task specification. These assumptions are the subset of assumptions of the task specification that are not part of the properties of the domain model, i.e., which are not fulfilled by the current model. Like all other elements of the architecture bridges may make use of ontologies (called bridge ontologies in their case).

Figure 7 shows our running example. Manifestations, causes and the order on hypotheses can be renamed directly. The definition of the function *explain* requires a more complex logical expression that transforms a binary relation into a function having sets as values.

3.6 Problem-Solving Method

Problem-solving methods describe the reasoning steps and types of knowledge which are needed to perform a task. In addition to some minor differences between the approaches, there is strong consensus that a problem-solving method: (1) decomposes the entire reasoning task into more elementary substeps, (2) defines the types of knowledge that are needed by the inferences to be done, and (3) defines control and knowledge flow between the inferences (cf. [Schreiber et al., 1994]). In addition, [Van de Velde, 1988] and [Akkermans et al., 1993] define the *competence* of a problem-solving method independently from the specification of its operational reasoning behavior (i.e., a functional black-box specification). Proving that a problem-solving method has some competence has

td bridge *complete and parsimonious explanation @ anesthesiology*

pragmatics

The bridge connects the task of finding a complete and minimal explanation with domain knowledge on *anesthesiology*

rename

Manifestation $\Rightarrow$ *Finding;*

Cause $\Rightarrow$ *Hypothesis;*

A smaller hypothesis has higher preference

$\supset \Rightarrow <$

mapping axioms

A finding is explained by a hypothesis iff there is an element of the hypothesis that is a cause of the finding.

$s \in explain(H) \leftrightarrow \exists\, h\, (h \in H \wedge cause\ relation(h,s))$

Fig. 7 The td-bridge *complete and parsimonious explanation @ anesthesiology.*

the clear advantage that the selection of a method for a given problem and the verification whether a problem-solving method fulfills its task can be performed independently from details of the internal reasoning behavior of the method.

We will describe the UPML Architecture of Problem-Solving Methods and we will provide some illustrations.

3.6.1 The UPML Architecture of Problem-Solving Methods

UPML distinguishes two different types of problem-solving methods: *complex problem-solving methods* that decompose a task into subtasks and *primitive problem-solving methods* that make assumptions about domain knowledge to perform a reasoning step. Therefore, the latter correspond to inference actions in CommonKADS. They do not have an internal structure, i.e. their internal structure is regarded as an implementational aspect of no interest for the architectural specification of the entire knowledge-based system.

UPML provides six elements for describing a complex problem-solving method:

- the already provided concept of (1) *pragmatics* is extended by *utility* which provides a ratio between the costs and the benefits of the method.

- The (2) *costs* of a problem-solving method have several different dimensions (cf. [O'Hara & Shadbolt, 1996], [Fensel & Straatman, 1998], [Fensel & Benjamins, 1998b]): interaction costs (with user and other aspects of the environment) and computation costs (in terms of average, worst, or typical cases). It is clear that a cost descriptions must regard both aspects and weigh them according to domain and application-specific circumstances. UPML does not make a specific commitment because there is no general and agreed to way to describe the costs of problem-solving methods (or algorithms in general).

- The (3) *communication policy* describes the communication style of the method and its components. It defines the ports of each building block and for each port how the block reacts to incoming events and how it provides outcoming events (compare [Allen & Garlan, 1997], [Yellin & Strom, 1997]). In regard to the entire method, the communication protocol describes the interaction of the system with its environment. In case of an inference action it describes how a component interacts with other components. The interaction style of the different components must be compatible to each other and to the overall control introduced by the problem-solving method.[4]

4. For example, a component *A* cannot wait of the input of a component *B* when the control of the problem-solving methods decides to start with *A*. In general, the control defined by the method realizes a subset of all possible and meaningful interaction patterns of the components.

- The (4) *ontology* provides a signature used for describing the method (cf. [Studer et al., 1996], [Fensel et al., 1997], [Chandrasekaran et al., 1998], [Gennari et al., 1998]).

- The (5) *competence description* provides functional specification of the problem-solving method. It introduces preconditions restricting valid inputs and the postconditions describe what the method provides as output.

- Finally, the (6) *operational specification* complements this with the description of the actual reasoning process of the method. Such an operational description explains how the desired competence can be achieved. It defines the data and control flow between the main reasoning steps so as to achieve the functionality of the problem-solving method. For this purpose it introduces intermediate roles (the stores for input and output of the intermediate reasoning steps), the procedures, and the control. For specification of control we rely on MCL [Fensel et al., 1998a] that integrates $(ML)^2$ [van Harmelen & Balder, 1992] and KARL [Fensel et al., 1998b] into a coherent semantical framework. However, we made the syntax of MCL more readable and we will explain its primitives below with an example

A complex method decomposes a task into subtasks and therefore recursively relies on other methods that process its subtasks. Such a subtask may describe a complex reasoning task that may further be decomposed by another problem-solving method or may be performed directly by a simple problem-solving method.

The specification of a primitive problem-solving method closely resembles the definition of a complex one with two significant differences: A primitive problem-solving method does not provide an operational specification and the definition of the competence differs slightly. Assumptions describe properties of the domain knowledge that is needed by the method. The assumptions about domain knowledge are the equivalent of the assumptions about the subtasks of a complex problem-solving method.

3.6.2 An Example Problem-Solving Method

We present the generic search method of [Fensel & Motta, to appear] as example. First we provide the definition of the method ontology in Figure 8. Basically it defines a sort *object* which is used to define *input* and *output* of the method and a predicate *stop criterion* which will be used to characterize the postcondition of the method.

Figure 9 provides the specification of the *search* method. The method receives a set of objects as input and provides one object as output. It assumes that the *input* is not empty. It decomposes the entire reasoning process into five subtasks: *Initialize*, *Derive Successor Nodes*, *Select Node*, *Stop*, and *Update Nodes* which are defined in Figure 10. The method makes four assumptions about its subtasks:

- *Initialize* must select a non-empty subset from the input.
- *Select Node* must select an element from its input.
- *Stop* must be true for some instantiations.
- *Update Nodes* must provide a non-empty subset of the union of its inputs.

In its postcondition the method guarantees that the stop criterion holds for its output. Figure 9 also describes how such a state fulfilling the postcondition can be achieved. It defines that a recursive search is performed after the initialization. This recursive search iterates three steps. First, we select an object from its input, then we derive the successors of this object and use the input and successors to create a new object set. If the selected object and the newly created object set fulfill the stop criterion we stop the search and receive the object as output. Otherwise we continue recursion with the newly constructed object set. UPML distinguishes whether we get **one** or **all** outputs of a task and inference.[5]

ontology *search*
pragmatics
The ontology provides an ontology for describing search;
signature
elementary sorts
Object
constructed sorts
Objects **set of** *Object*
constants
input : *Objects;*
output : *Object;*
node : *Object;*
nodes : *Objects;*
successor nodes : *Objects*
predicates
initial select : *Objects* x *Objects;*
select node : *Object* x *Objects;*
stop criterion : *Object* x *Objects;*
successor : *Objects* x *Objects;*
update node : *Objects* x *Objects* x *Objects*

Fig. 8 A PSM ontology of *search.*

5. Instead of predicates functions can also be used to model state transitions via tasks. In this case, the **all** operator updates the entire function that collects the output of the operator whereas the **one** operator updates the function that collects the output of the operator for one argument (cf. [Fensel et al., 1998a]).

problem solving method *search*
pragmatics
The search method stops when a node is found that fulfils the stop criterion
ontology
search
competence
roles
input *input*; **output** *output*
preconditions
$input \neq \emptyset$
subtasks
Derive Successor Nodes; Initialize; Select Node; Stop; Update Nodes
postconditions
The stop criterion holds for the output and a subset of a predecessor of the output.
$\exists x$ (*stop criterion*(*output*,*x*) $\wedge$ $x \subseteq y \wedge$ *successor*(*y*,*output*)
If the stop criterion holds for *x* in a context *y* it also holds for all subsets of *y*
(i.e., for all smaller contexts).
stop criterion(*x*,*y*) $\wedge$ $z \subseteq y \rightarrow$ *stop criterion*(*x*,*z*)
operational description
intermediate roles
intermediate *node;* **intermediate** *nodes;* **intermediate** *successor nodes*
procedures
Recursive Search
control
search()
begin
/* Initializes and starts the search. */
nodes:= **all** *x* **.** *Initialize*(**output** *x*, **input** *input*);
output := **one** *x* **.** *Recursive Search*(**output** *x*, **input** *nodes*)
end
Recursive Search(**output** *x*, **input** *nodes*)
begin
node := **one** *x* **.** *Select Node*(**output** *x*, **input** *nodes*);
successor nodes := **all** *x* **.** *Derive Successor Nodes*(**output** *x*, **input** *node*);
nodes := **all** *x* **.** *Update Nodes*(**output** *x*, **input** *nodes*, **input** *successor nodes*);
if *Stop* (**input** *node*, **input** *nodes*)
then return *node*
else *Recursive Search*(**output** *x*, **input** *nodes*)
endif
end

Fig. 9 A problem-solving method *search*.

The states are described by the "constants" *input*, *node*, *nodes*, *output*, and *successor nodes*. Constants may have different values in different states according to the multiple-world semantics of our specification approach.[6] In fact, the logic MCL [Fensel et al., 1998a] which provides the formal base for our representation of control, allows more complex structures to store a state in the left side of expressions expressing state transitions.

6. In that sense they are not constants. They rather correspond to a constant "address" (i.e., a name) of a storage cell with changing content according to the current state of computation.

task *Derive Successor Nodes*
ontology
search
specification
roles
input *node;*
output *successor nodes*
goal
task(**input** *node*; **output** *successor nodes*) →
successor(*successor nodes*, *node*)
task(**input** *node*; **output** *successor nodes*) ∧ *successor nodes* = ∅ →
¬*successor*(*successor nodes*, *node*)

task *Initialize*
ontology
search
specification
roles
input *input;* **output** *nodes*
goal
task(**input** *input*; **output** *nodes*) →
initial select(*nodes*,*input*) ∧
nodes ⊆ *input* ∧
nodes ≠ ∅
precondition
input ≠ ∅
assumptions
Initial select selects something if there is anything.
y ≠ ∅ → ∃*x initial select*(*x*,*y*);
Initial select selects a subset of the set it selects from.
initial select(*x*,*y*) →
(*x* ⊆ *y* ∧ *x* ≠ ∅)

task *Select Node*
ontology
search
specification
roles
input *nodes;* **output** *node*
goal
task(**input** *nodes*; **output** *node*) →
select node(*node*,*nodes*)
node ∈ *nodes*
preconditions
nodes ≠ ∅
assumptions
Select selects something if there is anything.
y ≠ ∅ → ∃*x select node*(*x*,*y*);
Select selects an element of the set it selects from.
select node(*x*,*y*) → *x* ∈ *y*

task *Stop*
ontology
search
specification
roles
input *node;*
input *nodes*
goal
task(**input** *node*; **input** *nodes*) →
stop criterion(*node*,*nodes*)
assumptions
It is possible to fulfil the stop criterion.
∃*x*,*y stop criterion*(*x*,*y*);
If the stop criterion holds for *x* in a context *y* it also holds for all subsets of *y* (i.e., for all smaller contexts).
stop criterion(*x*,*y*) ∧ *z* ⊆ *y* →
stop criterion(*x*,*z*)

task *Update Nodes*
ontology
search
specification
roles
input *nodes;*
input *successor nodes;*
output *x*
goal
Update node provides a non-empty subset of its input.
task(**input** *nodes*; **input** *successor nodes*, **output** *x*) →
update node(*x*,*nodes*,*successor nodes*) ∧
x ⊆ *nodes* ∪ *successor nodes* ∧ *x* ≠ ∅
precondition
Update nodes receives a non-empty input.
nodes ∪ *successor nodes* ≠ ∅
assumptions
Update node an output for non-empty input.
y ∪ *z* ≠ ∅ →
∃*x update node*(*x*,*y*,*z*);
Update node provides a non-empty subset of its input.
y ∪ *z* ≠ ∅ ∧ *update node*(*x*,*y*,*z*) →
(*x* ⊆ *y* ∪ *z* ∧ *x* ≠ ∅)

Fig. 10 The subtasks of *search.*

Complex functions representing list, records or more complex data structures can be used. For more details see [Fensel & Groenboom, 1996], [Fensel et al., 1998a]. For the given example, constants are rich enough to be used as data structure.

Figure 10 provides the definition of the five subtasks of *search*. The definition of the subtasks is externalized from the specification of the problem-solving method to enable their reuse in other problem-solving methods.

3.7 Problem-Solving Method Refiner

The search method we sketched in section 3.6 is still very generic. That is, it can be used for nearly any type of task and domain. However, it requires much adaptation effort if applied. [Fensel, 1997] and [Fensel & Benjamins, 1998a] introduce a principled approach for the adaptation of problem-solving methods that provides more refined and therefore usable methods but avoids the exponential increase in the number of components required. This is achieved by *externalizing* the adaptation of methods. The generic method can still be used for cases in which none of its already implemented refinements fit the task and domain-specific circumstances. Moreover, refinements themselves can also be specified as reusable components and be used to refine different methods. A more detailed discussion of the different dimensions of the refinement space and means to move within this space can be found in Section 4.2.1.2 (see also [Fensel & Motta, to appear]). We only touch on this aspect here as it requires specific primitives in UPML.

UPML provides PSM refiners that take the specification of a problem-solving method as input and provide a refined version of this method as output. In that sense this refiner corresponds to design operators in KIDS/Specware [Smith, 1996] which is an approach for semiautomatic program refinement. However, a refiner in UPML cannot change the algorithmic structure of the problem-solving methods. They are used to refine declarative definitions in the competence description (i.e., they refine a logical theory and not a program).

The UPML template for specifying PSM refiners enables the refinement of each element of the competence description of a problem-solving method. An example for a PSM refiner is given in Figure 12. It refines the generic problem-solving method *search* to a *hill climbing* type of search. The refinements are the following:

- It ensures that the task *Initialize* returns only one object (i.e., a set with one object).

- The task *Update Nodes* is refined to *Update Nodes for hill climbing*. This actually causes the most refinement effort. First, we require an order "<" on *objects*—defined in the new ontology (see Figure 11) and also used for refining the postconditions—and we

ontology *search with preference*

pragmatics

The ontology provides a ontology for describing search according to a preference

import

search

signature

predicates

$<$: *Object* x *Object*

axioms

Non-symetry of $<$.

$\neg\, x_1 < x_1$;

Transitivity of $<$.

$x_1 < x_2 \wedge x_2 < x_3 \rightarrow x_1 < x_3$

Fig. 11 A PSM ontology of *search with preference*

formulate several axioms that *Update Nodes for hill climbing* has to respect. The new axioms ensure that we select an element of the *successor nodes* if one exists which is better than the (unique) element of the current *nodes*, and that we select such a better element. They ensure that we select the element of *nodes* if no better successor exists. This update policy captures the essence of hill climbing that selects better neighbors and stops if no better neighbor has been found. Finally, we require that it selects a subset that has only one element.

- We refine the postcondition of the method. *Stop criterion* is fulfilled for output if the second set does not contain a better object and if the current output has no successors (see Figure 12).

More refiners can be found in [Fensel & Motta, to appear].

A serious shortcoming of UPML is that it is currently not possible to refine a role into several new roles. For example, in the case of refining a design task you may want to refine a general input role into a role that receives constraints and one that receives requirements. However, this is not possible in the current version of UPML. That is, UPML provides hierarchical refinement of subtasks, but not of roles. More serious work would be necessary to include both in a non-conflicting manner into UPML. Similar problems are well-known from Petri nets where hierarchical refinement of transitions has to be combined with the refinement of places.

PSM refiner *Search* → *hill climbing*

pragmatics

The search method searches for a local optimum

ontology

search with preference

competence

refined subtasks

Initialize → *Initialize for hill climbing;*

Update Nodes → *Update Nodes for hill climbing*

refined postconditions

There are no better successors of the output.

$\neg\exists x_1\ (x_1 \in x \wedge successor(x,output) \wedge output < x_1)$

Task refiner *Initialize* → *Initialize for hill climbing*

ontology

search with preference

competence

refined goal

Initial select selects one element.

$initial\ select(x,y) \wedge x_1 \in x \wedge x_2 \in x \rightarrow x_1 = x_2$;

Task refiner *Update Nodes* → *Update Nodes for hill climbing*

ontology

search with preference

competence

refined goal

Update node selects from *z* if it contains an element that is better than all elements from *y*.

$update\ node(x,y,z) \wedge \exists x_1\ (x_1 \in z \wedge (x_2 \in y \rightarrow x_2 < x_1)) \rightarrow x \subseteq z$;

Otherwise it selects from *y*.

$update\ node(x,y,z) \wedge \neg\exists x_1\ (x_1 \in z \wedge (x_2 \in y \rightarrow x_2 < x_1)) \rightarrow x \subseteq y$;

Update node only selects elements for which no better element in z exist.

$update\ node(x,y,z) \wedge \neg\exists x_1, x_2\ (x_1 \in z \wedge x_2 \in x \wedge x_2 < x_1)$;

Update node selects one element.

$update\ node(x,y,z) \wedge x_1 \in x \wedge x_2 \in x \rightarrow x_1 = x_2$

Fig. 12 PSM and task refiners that refine *search* to *hill climbing*

3.8 PSM-Task and PSM-Domain Bridges

Problem-solving methods are specified separate from tasks as proposed by [Beys et al., 1996]. This enables the reuse of these methods for different tasks and conversely, the application of different methods to the same tasks. The explicit connection of tasks and methods is provided by PSM-Task bridges in a way similar to the one shown in Section 3.5 for the connection of task and domain. An example for a PSM-Task bridge is given in

pt bridge *hill climbing @ complete and parsimonious explanation*

pragmatics

The bridge connects the task of finding a complete and minimal explanation with the *PSM hill climbing*

rename

Object ⇒ *Hypotheses;*
output ⇒ *diagnosis*

mapping axioms

(1) The input is the set of all hypotheses.
input = {*h* | *h* ∈ *hypothesis*};
(2) A successor is a set with one element less.
successor(*H*,*H'*) ↔ ∃*h* (*h* ∈ *H* ∧ *H'* = *H* \þ{*h*});
(3) Only complete hypotheses (complete for the provided observations) are regarded as being better.
$H <_{PSM} H' \leftrightarrow H <_{Task} H' \wedge complete(H', observations)$

assumptions

(1) The input explains all observations.
explain(*input*) = *observations*;
(2) Monotonicity of the explain function (i.e., never smaller hypotheses explain more.
H ⊆ *H'* → *explain*(*H*) ⊆ *explain*(*H'*)

Fig. 13 The pt-bridge *hill climbing @ complete and parsimonious explanation*.

Figure 13. It connects *hill climbing* with the task of finding a *complete and parsimonious explanation.* The bridge provides three main mappings in addition to some simple renamings:

- The input of the method is the set of all hypotheses.
- Successors of an object (which is a set of hypotheses) are sets with one less element.
- The order used by the method is the order defined by the task restricted to complete explanations (i.e., only complete explanations have a higher preference).

However, these mappings alone would not guarantee that the *output* of the method fulfills the *goal* of the task. The task of finding a complete and parsimonious explanation is NP-hard in the number of hypotheses [Bylander et al., 1991]. *Hill climbing* only performs a local search on complete sets (see Figure 13). In consequence we need two axioms that close the gap between the task definition and the competence of the problem solving method. The method must receive a correct initial set (cf. Figure 13). Constructing an initial correct set is beyond the scope of the method. It is assumed as being provided by the domain knowledge, by a human expert, or by another problem-solving method. The method only minimizes this correct set. Second, the method is only able to find a *locally* minimal sets. Local minimality means that there is no correct subset of the output that has one less element. Still this is not strong enough to guarantee parsimonity of the explanation in the general case. Smaller subsets may exist that are complete explanations. In [Fensel &

Schönegge, 1998], we have proven that the global-minimality of the task definition is implied by the local-minimality if we introduce the *monotonic-problem assumption* (see [Bylander et al., 1991] and Figure 13). How such properties can be verified is described in [Fensel & Schönegge, 1997], and [Fensel & Schönegge, 1998] provides a method called *inverse verification* that can be used to derive such assumptions (see Section 4.2.1.2).

The bridge type that connects problem-solving methods with domain models remains to be introduced. Notice that parts of these mappings have already been established through connecting the method with a task and the task with a domain. Only knowledge requirements that are exclusively introduced for the method need further mappings. A method usually introduces further requirements on heuristic domain knowledge used to guide its search process. This knowledge is not required to define the task and is therefore not part of its mapping. However, in our simple example the entire mapping is achieved by merely importing the task-domain and PSM-task bridges (see Figure 14). A more systematic study in the second phase of IBROW that explores the relationships between different bridges and refiners using category theory will hopefully bring more insight and methodological background to better understand such phenomena.

3.9 The Meta Ontology of UPML

We used Protégé (cf. [Puerta et al., 1992], [Eriksson et al., 1999]) to develop a *meta* ontology of UPML. Protégé is a knowledge acquisition tool generator. After defining an ontology it semiautomatically generates a graphical interface for collecting the knowledge that is described by the ontology. The ontology can be described in terms of classes and attributes and organized with an is-a hierarchy and attribute inheritance. However, the first version of the meta ontology of UPML was not based on well-funded development guidelines. For example, refiners that were basically binary relations between concepts were also modeled as concepts. Secondly, the notion of a library was missing (i.e., available only implicitly). Third, the import attribute was completely overloaded. The underlying principle of this ontology was to maximize attribute inheritance between concepts in order to detect hidden dependencies between UPML concepts. The second version is based on a clear *meta*-meta-level ontology consisting of concepts, binary relationships, and restricted

pd bridge *hill climbing @ anesthesiology*
pragmatics
The bridge connects the domain *anesthesiology* with the *PSM hill climbing*
import
pt bridge *hill climbing @ complete and parsimonious explanation*
td bridge *complete and parsimonious explanation @ anesthesiology*

Fig. 14 The pd-bridge *hill climbing @ anesthesiology.*

Entity
    attribute → type

Concept < Entity

Binary Relation < Entity
    $argument_1 \rightarrow Concept_1$
    $argument_2 \rightarrow Concept_2$

Restricted Binary Relation < Binary Relation
    in = $argument_1 \rightarrow Concept_1$
    out = $argument_2 \rightarrow Concept_2$
with $Concept_1 = Concept_2$

Fig. 15 The Meta-Meta-ontology of UPML.

binary relationships. All three entities may have attributes (see Figure 17). The main concepts of UPML that are defined with this basic ontology are given in (Figure 17). The main concepts are *Library*, *Ontology*, *Domain Model*, *PSM*, and *Task*. Except for *uses*, all attributes model *part-of* relationships. Subconcepts (*subclass-of* relationship) of PSM are *Complex PSM* and *Primitive PSM*. Binary relations connect two different component types. The root binary relation of UPML is *Bridge*. (Figure 17) Restricted Binary Relations connect two components of the same type. The root restricted binary relation of UPML is *Refiner* (cf. Figure 18).

Library < Concept
- pragmatics → Pragmatics
- ontology → Ontology
- domain model → Domain Model
- complex PSM → Complex PSM
- primitive PSM → Primitive PSM
- task → Task
- ontology refiner → Ontology Refiner
- cpsm refiner → CPSM Refiner Refiner
- ppsm refiner → PPSM Refiner Refiner
- task refiner → Task Refiner
- psm-domain bridge → PSM-Domain Bridge
- psm-task bridge → PSM-Task Bridge
- task-domain bridge → Task-Domain Bridge

Ontology < Concept
- *uses* → Ontology
- pragmatics → Pragmatics
- signature → Signature
- theorems → Formula
- axioms → Formula

Domain Model < Concept
- *uses* → Domain Model
- pragmatics → Pragmatics
- ontologies → Ontology
- theorems → Formula
- assumptions → Formula
- knowledge → Formula

PSM < Concept
- pragmatics → Pragmatics
- ontologies → Ontology
- cost → Cost
- communication → Communication
- precondition → Formula
- postcondition → Formula
- input roles → Role
- output roles → Role

Task < Concept
- *uses* → Task
- pragmatics → Pragmatics
- ontologies → Ontology
- goal → Formula
- input roles → Role
- output roles → Role
- precondition → Formula
- assumptions → Formula

Complex PSM < PSM
- subtasks → Task
- operational description → Operational Description

Primitive PSM < PSM
- knowledge roles→ Role
- assumptions → Formula

Fig. 16 The main concepts of the meta-ontology of UPML.

4 Architectural Constraints, Guidelines, and Tools

Architectural constraints ensure well formed specifications. The individual components and their connection via bridges and refiners must fulfill certain properties to provide a valid *product*. Development guidelines structure the *process* of developing such system models. A process model with defined transitions helps to navigate through the space of possible system designs. The development guidelines and their underlying theoretical background are described in more detail in [Fensel & Motta, 1998] and [Fensel & Motta, to appear]. We also developed some tools to support these processes. The tools fall into three categories: Editor, Browser, and Verifier. Their order reflects their respective difficulty.

4.1 Constraints

Architectural constraints ensure well-defined components and composed systems. The conceptual model of UPML decomposes the overall specification and verification tasks into subtasks of smaller grain size and clearer focus (cf. [van Harmelen & Aben, 1996]). The architectural constraints of UPML consists of requirements that are imposed on the intra- and interrelationships of the different parts of the architecture. They either ensure a valid part (for example, a task or a problem-solving method) by restricting possible relationships between its subspecifications or they ensure a valid composition of different elements of the architecture (for example, they are constraints on connecting a problem-solving method with a task). The constraints on well-defined components apply for tasks,

Bridge e < Binary Relation
argument$_1$ → Concept$_1$
argument$_2$ → Concept$_2$
pragmatics → Pragmatics
ontologies → Ontology
renaming → STRING
mapping axioms → Formula
assumptions → Formula

PSM-Domain Bridge < Bridge
argument$_1$ → Domain
argument$_2$ → PSM
uses → PSM-Domain Bridge,
Task-Domain Bridge,
PSM-Task Bridge

PSM-Task Bridge e < Bridge
argument$_1$ → PSM
argument$_2$ → Task
uses → PSM-Task Bridge

Task-Domain Bridge e < Bridge
argument$_1$ → Task
argument$_2$ → Domain
uses → Task-Domain Bridge

Fig. 17 Binary relations in the meta-ontology of UPML.

Refiner < Restricted Binary Relation
pragmatics → Pragmatics
ontologies → Ontology
in → Concept
out → Concept

Domain Refiner < Restricted Binary Relation
in → Domain Model
out → Domain Model
theorems → Formula
assumptions → Formula
knowledge → Formula
axioms → Formula
renaming → Renaming

Ontology Refiner < Restricted Binary Relation
in → Ontology
out → Ontology
signature → Signature
theorems → Formula
axioms → Formula
renaming → Renaming

Task Refiner < Restricted Binary Relation
in → Task
out → Task
goal → Formula
input roles → Role
output roles → Role
precondition → Formula
assumptions → Formula
axioms → Formula
renaming → Renaming

PSM Refiner < Restricted Binary Relation
in → PSM
out → PSM
cost → Cost expression
communication → STRING
axioms → Formula
input roles → Role
output roles → Role
precondition → Formula
postcondition → Formula
renaming → Renaming

CPSM Refiner < PSM Refiner
in → CPSM
out → CPSM
subtasks → Task

PPSM Refiner < PSM Refiner
in → PPSM
out → PPSM
knowledge roles→ Role
assumptions → Formula

Fig. 18 Restricted binary relations in the meta-ontology of UPML.

domain models, and PSMs. The constraints for composition are introduced by constraints that apply to bridges. In UPML, these constraints are presented as formal restrictions.

For a task specification we require consistency, i.e:[7]

T$_1$ *ontology axioms* ∪ *preconditions* ∪ *assumptions* must have a model.

Otherwise we would define an inconsistent task specification which would be unsolvable. In addition, the following must hold (cf. [Fensel & Groenboom, 1997], [Fensel & Schönegge, 1997]):

[7]. For readability, in the following we write *goal* when strictly speaking we mean *task(...)*, as defined in the *goal*-slot of a *task*.

T_2 Each model of *ontology axioms* ∪ *preconditions* ∪ *assumptions*
must be an elementary substructure of at least one model of *goal*[8]

That is, if the ontology axioms, preconditions, and assumptions are fulfilled by a domain for a given case then the goal of a task must be achievable. This constraint ensures that the task model makes the underlying assumptions of a task explicit. For example, when defining a global optimum as a goal of a task it must be ensured that a preference relation exists and that this relation has certain properties. It must be ensured that $x < y$ and $y < x$ (i.e., symmetry) is prohibited because otherwise the existence of a global optimum cannot be guaranteed.

In the example from Figure 3 and Figure 4, finding a model for *goal* amounts to finding a complete and parsimonious diagnosis. The axioms from the domain ontology (Figure 3) define both of these notions. The existence of a complete diagnosis follows easily from the precondition and first task-assumption (Figure 4). The existence of a parsimonious diagnosis the other three task-assumptions: since any diagnosis is assumed to be finite (fourth assumption), and since a partial ordering < is assumed on diagnoses (second and third assumption), for any complete diagnosis there must also be a smallest contained diagnosis (ie a parsimonious diagnosis), since the finiteness of the diagnosis rules out infinitely descending chains in the partial ordering.

These are the two architectural constraints UPML imposes to guarantee well-defined task specifications. A third optional constraint ensures minimality of assumptions and preconditions (called weakest preconditions in software engineering, cf. [Dijkstra, 1975]) and therefore maximizes the reusability of the task specification. It prevents overspecifity of assumptions and preconditions. Otherwise they would disallow applying a task to a domain even in cases where it would be possible to define the problem in the domain.

T_3 Each model of *goal* must be an elementary extension of a model of
ontology axioms ∪ *preconditions* ∪ *assumptions*

How minimality of assumptions can be proven and how such assumptions can be found is sketched in Section 4.2.1.2 (see also [Fensel & Schönegge, 1998]).

A domain model in UPML must fulfil the following constraints:

D_1 Each model of
domain ontology axioms ∪ *domain assumptions* ∪ *domain knowledge*
must be an elementary substructure of at least one model of *domain properties*

8. A structure R is an *elementary substructure* of a structure S *iff* the universe of R is a subset of the universe of S and the interpretation of each relation, function and constant symbol in R is the restriction of the corresponding interpretation in S (see e.g. [Keisler, 1977]). In other words: S can be constructed by "extending" R.

D_2 *domain ontology axioms* ∪ *domain assumptions* ∪ *domain knowledge* must have a model.

Again, minimality of assumptions (**D_3**) may be asked for.

Similar constraints hold for problem-solving methods:

$PPMS_1$ *PSM assumptions* ∪ *PSM preconditions* ∪ *PSM ontology axioms* must have a model.

In the case of complex problem-solving methods, we need further constraints that formulate desired properties of the operational specification.

$CPSM_1$ The following must hold in the case of *weak* correctness:
PSM ontology axioms ∪ *preconditions of the PSM* ∪ *goals of subtasks*
├— [*MCL program of operational specification*] *PSM postconditions*[9]

$CPSM_2$ In addition in the case of *strong* correctness the following must hold:
PSM ontology axioms ∪ *preconditions of the PSM* ∪ *goals of subtasks*
├— <*MCL program of operational specification*> *true*[10]

In the former case we ensure that if the method terminates it terminates in a state that respects the postconditions of the method. In the latter case we also guarantee termination. Again, one can investigate and require that axioms, preconditions, and subtasks are minimal (**$CPSM_3$** and **$CPSM_4$**).

Finally, ontologies should not be inconsistent, i.e.,

O_1 *ontology axioms* must have a model.

O_2 *ontology axioms* must entail *ontology theorems*.

Bridges are accompanied by architectural constraints that restrict the possible combinations of parts of an architecture in UPML. The following must hold for bridges:

TDB_1 There exists a model of
renaming(domain ontology ∪ *domain knowledge* ∪ *domain assumptions)* ∪
task ontology ∪ *task preconditions* ∪ *task assumptions* ∪
bridge ontology ∪ *mapping axioms* ∪ *bridge assumptions*

---

9. [] is a modal operator of dynamic logic, cf. [Harel, 1984]. [*p*]α means after every execution of program *p* the formula α holds.

10. <> is a modal operator of dynamic logic, cf. [Harel, 1984]. <*p*>α means after some execution of program *p* the formula α holds

TDB_2 Domain and bridge entail the assumptions of the task
renaming(domain ontology ∪ domain knowledge ∪ domain assumptions) ∪
task ontology ∪ bridge ontology ∪ mapping axioms ∪ bridge assumptions
|= task assumptions

TPB_1 There exists a model of
renaming(PSM preconditions ∪ PSM assumptions ∪ PSM postconditions) ∪
task ontology ∪ task preconditions ∪ task assumptions ∪
bridge ontology ∪ mapping axioms ∪ bridge assumptions

TPB_2 Task, problem-solving method and bridge entail that the goal of the task is fulfilled by the postcondition of the method
renaming(PSM preconditions ∪ PSM assumptions ∪ PSM postconditions) ∪
task ontology ∪ task preconditions ∪ task assumptions ∪
bridge ontology ∪ mapping axioms ∪ bridge assumptions |= task goal

PDB_1 There exists a model of
renaming(domain ontology ∪ domain knowledge ∪domain assumptions) ∪
imported task-domain bridge ∪
PSM preconditions ∪ PSM assumptions ∪ PSM postconditions
bridge ontology ∪ mapping axioms ∪ bridge assumptions

PDB_2 Domain and bridge entail the assumptions of the problem-solving method
renaming(domain ontology ∪ domain knowledge ∪ domain assumptions) ∪
PSM ontology ∪ bridge ontology ∪ mapping axioms ∪ bridge assumptions
|= PSM assumptions

Again, further constraints arise when we ask for minimality of assumptions introduced by bridges.

Finally, refiners express relationships between components of the same type. The following must hold:

$CPSMR_1$ There exists a model of
PSM ontology axioms ∪ refined PSM preconditions ∪
refined goals of subtasks

$CPSMR_2$ The following must hold in the case of *weak* correctness:
PSM ontology axioms ∪ refined PSM preconditions
∪ refined goals of subtasks
|— [MCL program of operational specification] refined PSM postconditions

$CPSM_3$ In addition in the case of *strong* correctness it must hold:
PSM ontology axioms ∪ *refined PSM preconditions*
∪ *refined goals of subtasks*
⊢ <*MCL program of operational specification*> *true*

$PPSMR_1$ There exists a model of
PSM ontology axioms ∪ *refined PSM preconditions*
∪ *refined PSM assumptions*

TR_1 There exists a model of
refined task ontology axioms ∪ *refined task preconditions*
∪ *refined task assumptions*

TR_2 Each model of
refined task ontology axioms ∪ *refined task preconditions* ∪
task refined assumptions
must be an elementary substructure of *refined task goal*

TR_3 Each model of *refined task goal* must be an elementary extension of
refined task ontology axioms ∪ *refined task preconditions* ∪
refined task assumptions

OR_1 *refined ontology axioms* must have a model.

OR_2 *refined ontology axioms* |= *refined ontology theorems*.

This list of constraints is not meant to be exhaustive. Further useful requirements for well-defined specifications may be found in the course of further experience with UPML in practice.

4.2 Design Guidelines

Design guidelines define a process model for building complex KBSs out of elementary components. In the following, we will discuss three aspects of the process model of UPML. (1) How to develop an application system from reusable components. (2) How to develop a library of reusable task definitions and problem-solving methods. (3) Which components of UPML correspond to an implementation and how such components can be implemented in an object-oriented framework.

4.2.1 How to Develop an Application System

The overall process is guided by tasks that provide generic descriptions of problem classes. After selecting, combining and refining tasks, they are connected with PSMs and their combination is instantiated for the given domain. Elementary sorts of tasks and PSMs generally have to be connected to domain sorts. Further mapping axioms may be required to link the remaining terminology, and assumptions have to be made in cases where it is needed to bridge the gap between different components. In the following, we will discuss two main principles for this process:

- The twofold role assumptions may play for application development,
- Inverse Verification as a technique to find such assumptions, and

4.2.1.1 The Two Effects of Assumptions

Establishing the proper relationship between a PSM and a task usually requires *correctness* and *completeness* of the PSM relative to the goals of the task. However, a perfect match is unrealistic in many cases. In general, most problems tackled with KBSs are inherently complex and intractable (see e.g. [Bylander et al., 1991]). A PSM has to describe not just a realization of the functionality, but one which takes into account the constraints of the reasoning process and the complexity of the task. PSMs achieve an efficient realization of functionality by making *assumptions* [Fensel & Straatman, 1998]. These assumptions put restrictions on the context of the PSM, such as the domain knowledge and the possible inputs of the method or the precise definition of the goal to be achieved when applying the PSM. Notice that such assumptions can work in two directions to achieve this result. First, they can restrict the complexity of the problem, that is, weaken the task definition in such a way that the PSM competence is sufficient to realize the task. Second, they can strengthen the competence of the PSM by assuming (extra) domain knowledge. In the first case, a simpler problem is solved and in the second case generality is given up (i.e., the number of domains are reduced where the method can be applied to).

- *Weakening of functionality*: Reducing the desired functionality of the system and therefore reducing the complexity of the problem by introducing assumptions about the precise task definition. An example of this type of change is no longer requiring an optimal solution, but only an acceptable one, or making the single-fault assumption in model-based diagnosis.
- *Strengthening of domain assumptions*: Introducing assumptions about the domain knowledge (or the user of the system) which reduces the functionality or the complexity of the part of the problem that is solved by the PSM. In terms of complexity analysis, the domain knowledge or the user of the system is used as an oracle that solves complex parts of the problem. These requirements therefore strengthen the functionality of the method.

Both strategies are complementary. Both types of assumptions serve the same purpose of closing the gap between the PSM and the task goal which should be achieved by it. It is not an internal property of an assumption that decides its status, instead it is the functional role it plays during system development or problem solving that creates this distinction. Formulating it as a requirement involves considerable effort in acquiring domain knowledge during system development, and formulating it as a restriction involves additional external effort during problem solving if the given case does not fulfill the restrictions and cannot be handled properly by the limited system. More details can be found in [Fensel & Benjamins, 1998b].

4.2.1.2 Inverse Verification

A method for finding and constructing such assumptions is *inverse verification* (cf. [Fensel & Schönegge, 1998]). We use failed proofs as a search method for assumptions and we analyze these failures to construct and refine them. In other words, we attempt to prove that a problem-solving method achieves a goal and the assumptions appear during the proof as *missing pieces* in proving the correctness of the specification. A mathematical proof written down in a textbook explains why a lemma is true under some preconditions (i.e., assumptions and other sublemmas). The proof establishes the lemma by using the preconditions and some deductive reasoning. Taking a look at the proof *process* we get a different picture. Usually, initial proof attempts fail when they run into unprovable subgoals. These failed proof attempts point to necessary features that are not present from the beginning. Actually they make aware of further assumptions that have to be made in order to obtain a successful proof. Taking this perspective, a proof process can be viewed as a search and construction process for assumptions. Gaps found in a failed proof provide initial characterizations of missing assumptions. They appear as sublemmas that were necessary to proceed with the proof. An assumption that implies such a sublemma which would close the gap in the proof is a possible candidate for which we are looking. That is, formulating this sublemma as an assumption is an initial step in finding and/or constructing assumptions that are necessary to ensure that a problem-solving method behaves well in a given context. Using an open goal of a proof directly as an assumption normally leads to very strong assumptions. That is, these assumptions are *sufficient* to guarantee the correctness of the proof, but they are often neither *necessary* for the proof nor *realistic* in the sense that application problems will fulfill them. Therefore, further work is necessary to find improved characterizations for these assumptions. This could be achieved by a precise analysis of their role in the completed proof to retrace unnecessary properties.

Such proofs can be done semiformally in a textbook manner. However, providing specification formalisms with a formal syntax and formal semantics allows mechanized proof support. The great amount of details that arise when proving properties of software (and each problem-solving method eventually has to be implemented) indicates the necessity of such mechanization. Therefore, we provide a formal notion for problem-

solving methods and semi-automatic proof support by using KIV [Reif, 1995]. KIV provides an *interactive* tactical theorem prover that makes it suitable for hunting hidden assumptions. We expect many proofs to fail. Using a theorem prover that returns with such a failed attempt adds nothing. Instead of returning with a failure, KIV returns with open goals that could not be solved during its proof process. As opposed to verification, we do not start a proof with the goal of proving correctness here. Instead, we start an impossible proof and view the proof process as a search and construction process for assumptions. This explains the name *inverse verification.* More details on KIV are provided in Section 4.3.3.

[de Kleer, 1986] describes a truth-maintenance system (ATMS) that could in principle be applied to our problem. However, applying this technique introduces two strong (meta-) assumptions: (1) All the assumptions required to close the gap between the goals of the task and the competence of the problem-solving method must already be known and provided to the system. (2) The system needs to know the impacts of the assumptions, i.e. their influence on the truth of the formulas describing the competence of the problem-solving method and the goals of the task.

Constructive approaches to derive such assumptions can be found in approaches to abduction ([Cox & Pietrzykowski, 1986]), in program debugging with inductive techniques (cf. [Shapiro, 1982], [Lloyd, 1987]), in explanation-based learning (cf. [Minton et al., 1989], [Minton, 1995]) or more generally in inductive logic programming ([Muggleton & Buntine, 1988], [Muggleton & De Raedt, 1994]). However, these approaches achieve automation by making strong (meta-) assumptions about the syntactical structure of the representation formalisms of the components, the representations of the “error“, and the way an error can be fixed. Usually, Prolog or Horn logic is the assumed representation formalism, and errors or counter-examples are represented by a set of input-output tuples or a finite set of ground literals. Modification is done by changing the internal specification of a component. In this scenario, error detection boils down to backtracking a resolution-based derivation tree for a “wrong“ literal. We have to aim for new formulas, however, (i.e., an assumption may be represented by a complex first-order formula) and our “counter-examples“ are not represented by a finite set of ground literals, but by a complex first-order specification. Also most of the approaches mentioned do not regard architectural descriptions of the entire reasoning system.[11]

4.2.2 How to Develop reusable Component Libraries

We view problem solving method development as a process taking place in a three-dimensional space, defined by problem-solving strategies, domain assumptions, and task

11. An exception are the approaches to explanation-based learning that use explicit architecture axioms [Minton, 1995].

commitments (see Figure 19). These three dimensions are described below.

- **Problem-solving strategy.** This is a high-level description which specifies a type of problem solving rather than an actual algorithm, i.e. we describe a entire class of algorithms. A problem-solving strategy fixes some basic data structures, provides an initial task-subtask decomposition and a generic control regime. This generic control regime is meant to be shared by all problem-solving methods which subscribe to the same problem solving strategy. Examples of problem solving strategies are: *Generate&Test, Local Search* and *Problem Reduction* ([Smith & Lowry, 1990]).

- **Domain knowledge assumptions.** These are assumptions on the domain knowledge that is required to instantiate a problem-solving method in a particular application. These assumptions specify the types and the properties of the knowledge structures which need to be provided by a domain model, in addition to those required to fulfil task-specific commitments. For instance, when solving a design problem by means of Propose&Revise, a domain needs to provide the knowledge required to express *procedures* and *fixes*, in addition to the task-related knowledge needed to formulate the specific design problem—e.g. parts and constraints. Domain assumptions are necessary to enable efficient problem solving for complex and intractable problems ([Fensel & Straatman, 1998], [Fensel & Benjamins, 1998b]). More generally, the reliance on such domain-specific knowledge is the defining feature of knowledge-intensive approaches to problem solving.

- **Problem (i.e., task) commitments.** These specify ontological commitments to the type of problem that is to be solved by the problem-solving method. These commitments are expressed by subscribing to a particular *task ontology.* For example, a parametric design task ontology provides definitions for terms such as design model, parameter and constraint - see [Motta, 1999] for a detailed specification of a task ontology. The ontological commitments introduced by a task can be used to refine the competence of a problem-solving method, the structure of its computational state and the nature of the state transitions it can execute (cf. [Eriksson et al., 1995], [Fensel et al., 1996a], [Fensel et al., 1997]). For instance, a generic search method can thus be transformed into a specialized method for model-based diagnosis or parametric design. Such a task-specific refinement still produces a reusable method specification given that this is formulated independently of a particular application domain. A diagnostic problem solving method may be formulated in terms which are specific to diagnostic problem solving, but it can be reused in different technical or medical diagnostic applications. The advantage of refining problem-solving methods in a task-specific way is that the resulting model provides much stronger support for knowledge acquisition and application development than a task-independent one - i.e. the method becomes more *usable*.

Figure 19 visualizes the three dimensions of our problem solving method space by means

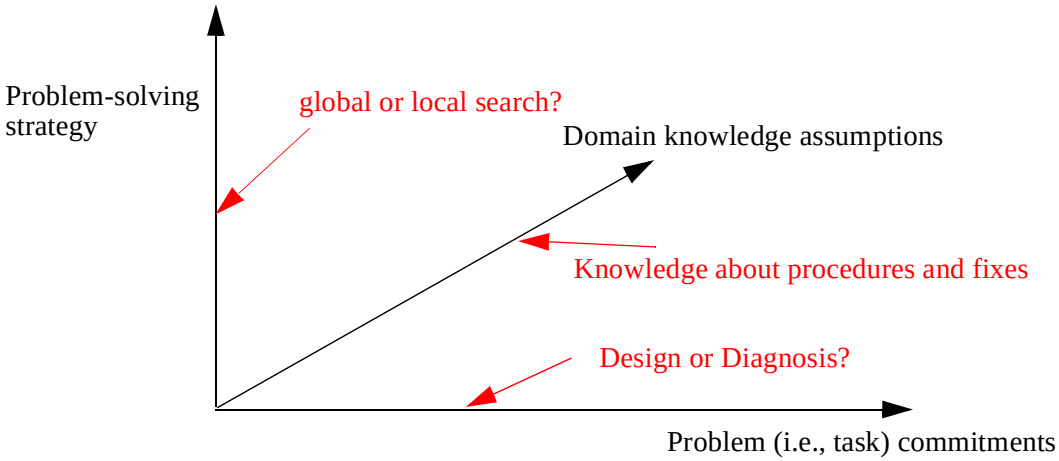

Fig. 19 The three dimensions of PSM description and development.

of arrows. Although this representation may be taken to imply that each dimension is characterized by a total order, this is not actually the case. Different tasks, such as diagnosis or design, and different problem solving schemes, such as local search or search by pruning (e.g. branch and bound), may not be derived from each other. However, they can be derived from more abstract definitions. Hence, each dimension is defined by an acyclic graph. The graph is defined by the refinement relationship between the elements of the design space and reflects the partial order defined by refinements. Having said this, we will focus only on one type of task (design) and one type of problem-solving scheme (local search) and these graphs therefore collapse into totally ordered ones.

A clear identification and separation of problem-solving strategy, problem commitments, and domain assumptions enables a principled development of a problem-solving method and structuring libraries of problem-solving methods. Current approaches usually merge these different aspects, thus limiting the possibilities for reuse, obscuring the nature of the methods, and making it difficult to reconstruct the process which led to a particular specification. In our approach the development and adaptation of problem-solving methods is characterized as a navigation process in this three-dimensional space. Movement through the three-dimensional space is represented by means of adapters. More details can be found in [Fensel & Motta, 1998] and [Fensel & Motta, to appear].

4.2.3 How to Implement Systems and Components

Although UPML seems to be rather abstract, there is a direct correspondence between the parts of UPML and object-oriented languages. In order to provide a framework for translating UPML specifications into implemented problem-solving methods we developed and refined some specific *Design Patterns* (cf. [Gamma et al., 1995]) that guide this translation process. A general design decision is that the problem-solving method

components can be reused without any modification. Modifications (e.g. adaptation and configuration) are only found in the bridges.

Problem-solving methods & Tasks: Each problem-solving method in an UPML specification can be mapped to a class implementing this problem-solving method. The subtasks of this problem-solving method are mapped to methods of the problem-solving method class. A problem-solving method communicates with other components via roles which are realized by bridges when configuring the whole problem-solving method.

Our running example depicted in Figure 9 provides a specification for a generic search problem-solving method. This problem-solving method has one input role *input*, one output role *output*, and the intermediate roles *node*, *nodes*, and *successor nodes*. Figure 20 depicts the Java-translation of the same specification: roles are translated into instance variables of the search class. Input- and output roles communicate with other components using a *setRole* and *getRole*-method, implemented from the general supperclass *PSMComponent*. The subtasks of the UPML specification are translated into methods of the PSM class. They also communicate with other *PSMComponents* using the methods *getSubTaskRole*, *setSubTaskRole* and *excecuteSubtasks*, which are implemented from the supperclass *PSMComponent*. Please note, that nothing is said here, how this subtasks are defined. The execution is delegated to adapters and the configuration can be done while designing the problem-solving method.

Ontologies & Domain Models: Ontologies are mapped to an ordinary class hierarchy, which defines the basic terminology used in the domain model and the problem-solving method. In the example above, the PSM ontology has to define node, nodes, object, objects etc. Notice, that these ontologies can be application-specific and that details of the actual definition of these classes are not used inside the problem-solving method. So the details of the data structure definitions can be implemented in an application dependent manner.

Refiners: Refiners are realized by method inheritance: if a problem-solving method is refined it means that a new subclass of an existing problem-solving method class is defined, which overwrites some of the subtasks of the original problem-solving method. Figure 21 depicts part of an refiner which refines the general search problem-solving method shown in Figure 20 into hill climbing.

Bridges: Bridges enable the basic communication infrastructure between several problem-solving method components (providing the subtask-PSM-mapping) and the domain. Because it has to be the most flexible part of the specification (it has to handle all incompatibilities between problem-solving method components) we can only formulate weak requirements. However, a bridge has to at least provide a common interface, such that problem-solving methods and bridges can be plugged together in a flexible way. The api provided by this interface can be structured into two groups: the first set of methods deals with the configuration of a problem-solving method, e.g. the Task-PSM mapping. Methods

```
class search extends PSMComponent{
  Objects input;
  Object output;
  Object node;
  Objects nodes;
  Objects sucessornodes;

  public void main(){
    input = (Objects) getRole("input");
    nodes = Initialize(input);
    output = RecursiveSearch(nodes);
    setRole("output",output);
  }
  Object RecursiveSearch(Objects nodes){
    node= SelectNode(nodes);
    successornodes = DeriveSuccessorNodes(node);
    nodes = UpdateNodes(nodes, successornodes);
    if Stop(node, nodes)
      return node
    else return RecursiveSearch(nodes);
  }
  Objects Initialize(Objects input){
    setSubTaskRole("Initialize","input",input);
    executeSubTask("Initialize");
    return (Objects) getSubTaskRole("Initialize","nodes");
  }
  Object SelectNode(Objects nodes){
    setSubTaskRole("SelectNode","nodes",nodes);
    executeSubTask("SelectNode");
    return (Object) getSubTaskRole("SelectNode","node");
  }
  Objects DeriveSuccessorNode(Object node){
    setSubTaskRole("DeriveSuccessorNode","node",nodes);
    setSubTaskRole("DeriveSuccessorNode","input",input);
    executeSubTask("DeriveSuccessorNode");
    return (Objects) getSubTaskRole("DeriveSuccessorNode","successornodes");
  }
  Objects UpdateNodes(Objects nodes, Objects successornodes){
    setSubTaskRole("UpdateNodes","nodes",nodes);
    setSubTaskRole("UpdateNodes","successornodes",successornodes);
    executeSubTask("UpdateNodes");
    return (Object) getSubTaskRole("UpdateNodes","x");
  }
  boolean Stop(Object node, Objects nodes){
    setSubTaskRole("Stop","node",node);
    setSubTaskRole("Stop","nodes",nodes);
    executeSubTask("Stop");
    return ((Boolean) getSubTaskRole("Stop","output")).booleanValue();
  }
```

Fig. 20 Java Translation of the problem-solving method search.

```
Refiner:
  class hillclimbing extends search{
    Objects Initialize(Objects input){
      return input.getOneElement();
  }
  Objects DeriveSuccessorNode(Object node){
    return getMax(node,input);
    ....
  }
  }

Bridge:
public interface BridgeComponent{
  void addPSM(String SubTaskName, PSMComponent PSM);
  void setRole(PSMComponent PSM, String PSMRoleName, Object content);
  Object getRole(PSMComponent PSM, String PSMRoleName);
  void setSubTaskRole(String SubTaskName, String SubTaskRoleName, Object content);
  void addMapping(String SubTaskName, String SubTaskRoleName, String
PSMRoleName);
  void executeSubTask(String SubTaskName);
  Object getSubTaskRole(String SubTaskName, String SubTaskRoleName);
  void addDomainRole(String SubTaskName, String PSMRoleName, DomainComponent
DomainKB);
}
```

Fig. 21 Implementation of Refiners and Bridges.

belonging to this group are e.g. *addPSM* and *addMapping* (see Figure 21). The second group of methods handles the execution of subtasks and set handling of roles. A bridge is usually domain- and problem-specific, a general type of bridge is often useful and sufficient. This kind of adapter just performs basic mappings. This adapter can be configured at runtime.

4.3 Tool Environment of UPML

An editor for UPML was built using Protégé-2000, the latest version of a series of tools developed in the Knowledge Modeling Group at Stanford Medical Informatics to assist developers in the construction of large electronic knowledge bases [Grosso et al., 1999]. The output of the editor is translated into Frame-logic [Kifer et al., 1995] allowing browsing and querying of UPML specifications with Ontobroker (cf. [Fensel et al., 1998c], [Fensel et al., 1999d]). The use of the interactive theorem prover KIV [Reif, 1995] for verifying architectural descriptions of knowledge-based systems. In the following, we will explain these tools in more detail.

4.3.1 The UPML Editor

We used Protégé-2000 (cf. [Puerta et al., 1992], [Eriksson et al., 1999]) to implement an editor for UPML specifications. Protégé allows developers to create, browse and edit domain ontologies in a frame-based representation, which is compliant with the OKBC knowledge model [Chaudhri et al., 1998]. From an ontology, Protégé automatically constructs a graphical knowledge-acquisition tool that allows application specialists to enter the detailed content knowledge required to define specific applications. Protégé allows developers to custom-tailor this knowledge-acquisition tool directly by arranging and configuring the graphical entities on forms, that are attached to each class in the ontology for the acquisition of instances. This allows application specialists to enter domain information by filling in the blanks of intuitive forms and by drawing diagrams composed of selectable icons and connectors. Protégé-2000 allows knowledge bases to be stored in several formats, among which a CLIPS-based syntax and RDF.

Using Protégé-2000, we modeled the meta-ontology of UPML (described in section 3.9) as a hierarchy of classes, with slots and facets attached to them. This work helped us better identify the concepts and relations of UPML necessary to provide a guided framework for defining UPML specifications. Based on the ontology, Protégé generated a graphical editor for instantiating specific UPML specifications, like the `Diagnostic Problem Solving` example explained throughout this paper. We custom-tailored the editor, in particular so that the knowledge-acquisition process centers around the use of diagram metaphors. For instance, we defined specific kinds of diagrams to enter the task decomposition of a complex PSM and the structure of its corresponding operational control. Figure 22 provides snapshots of the UPML editor.

4.3.2 The UPML Browser and "Reasoner"

The output of the UPML editor delivers text files of the ontology and UPML specifications in a Lisp like syntax (CLIPS). We implemented a simple cgi-script that translates these files into Frame Logic. The reason for this was that we want to use Ontobroker[12] as a browsing and query interface for UPML specifications. Ontobroker is an advanced tool for browsing and querying WWW information sources. It provides a hyperbolic browser and querying interface for formulating queries, an inference engine used to derive answers, and a webcrawler used to collect the required knowledge from the web. Figure 23 illustrates the hyperbolic presentation of the UPML meta ontology: classes in the center are depicted with a large circle, whereas classes at the periphery are denoted with a small circle. The visualization technique allows a quick navigation to classes far away from the center as well as a closer examination of classes and their vicinity. The structure of the frame-based representation language is used to define a tabular querying interface that frees users from

12. http://www.aifb.uni-karlsruhe.de/www-broker.

typing logical formulas (see Figure 23). When a user selects a class from the hyperbolic ontology view, the class name appears in the class field of the tabular query interface and the user can select one of the attributes from the attribute choice menu because the pre-selected class determines the possible attributes. The discussed tool set is implemented in Java and available through the WWW.[13] Figure 24 provides the reply of Ontobroker^{UPML} for the query that asks for all attributes of the task called „Complete and parsimonious diagnosis“.

In addition to human users, the query interface can also be used by other software agents like the Ibrow broker [Benjamins et al., 1998], [Fensel & Benjamins, 1998a]. In this case,

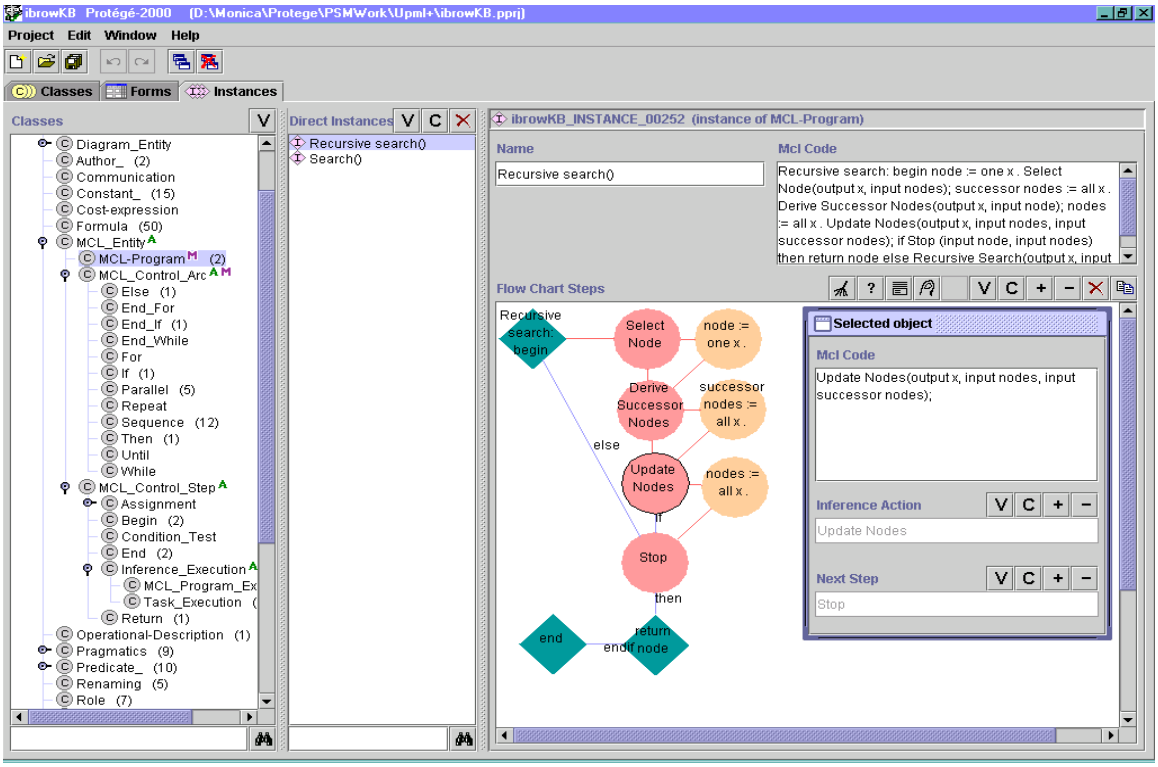

Fig. 22 Snapshots of the UPML editor built with Protégé-2000. The left panel displays the hierarchy of classes defined in the meta-ontology of UPML, the central panel lists the instances of the selected classes (here Complex-PSM and CPSM-Refiner) and the right panel shows the form associated with the class of the selected instance (here the “Search” Complex-PSM). The specific slot values for this instance are acquired and browsed either directly from this form, or via forms attached to the classes that define the slots. For instance, the form for a complex PSM includes a task decomposition diagram that allows to specify the competence slots of the PSM with automatically filled-in instances of input/output roles and tasks. The left superimposed form shows the “Update Node” task instance. The right superimposed form displays the MCL control structure over the subtasks of the PSM, also acquired through a diagram metaphor.

13. http://www.aifb.uni-karlsruhe.de/WBS/ibrow

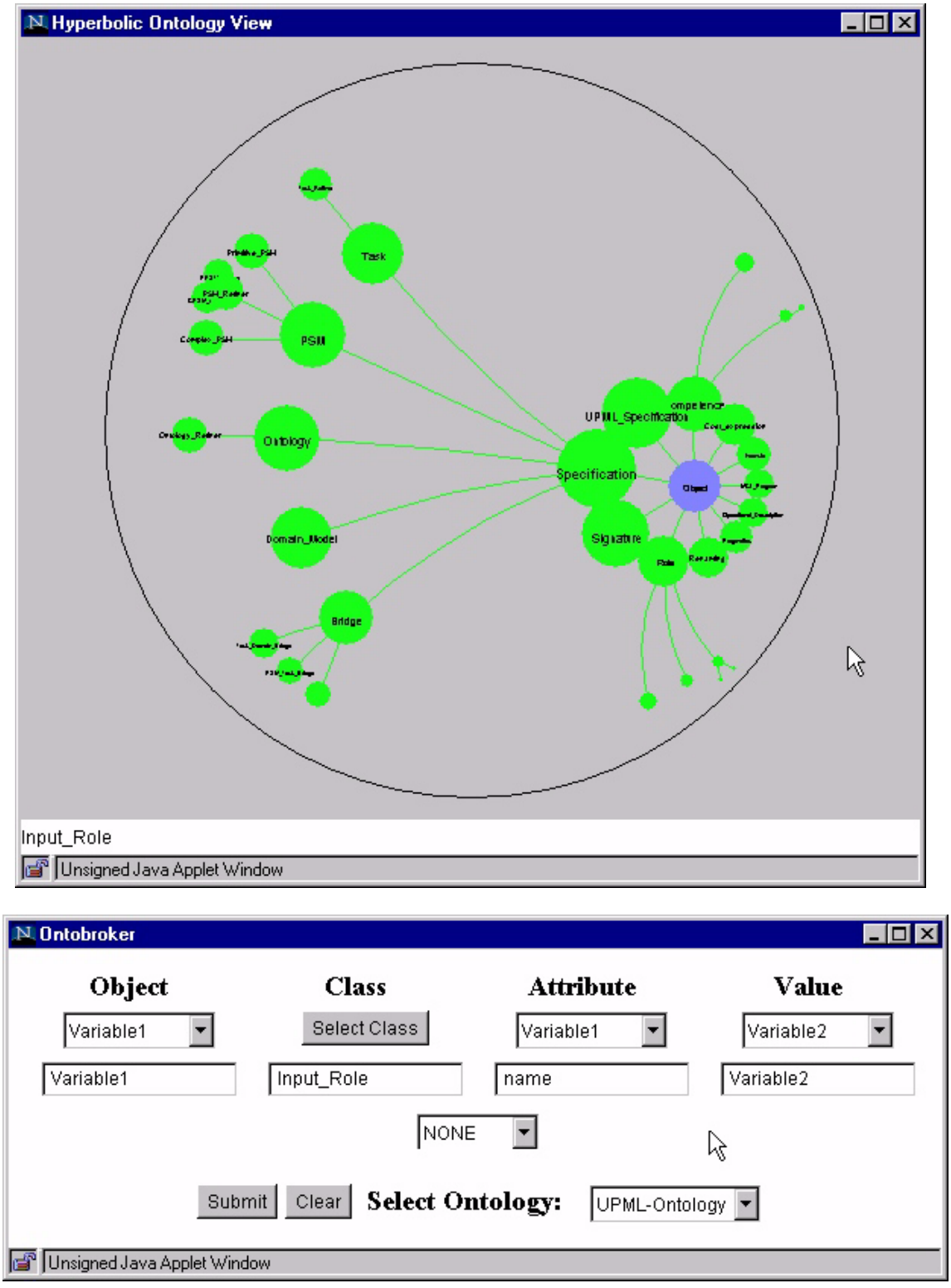

Fig. 23 The browsing and querying interface of Ontobroker[UPML].

Ontobroker provides an advanced query interface to various libraries used by the Ibrow broker to select the appropriate components and adapters to compose a system meeting to the requirements of a human client (cf. [Fensel et al., 1999d]). The relationship of the Ibrow broker and Ontobroker roughly corresponds to the relationship of mediator and wrappers in information integration architectures [Wiederhold, 1992].

4.3.3 The UPML Verifier KIV

KIV is an interactive theorem prover 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. It has been successfully applied in case-studies up to a size of several thousand lines of code and specification. KIV allows structuring of specifications and modularization of software systems. Therefore, the conceptual model of our specification can be realized by the modular structure of a specification in KIV. Finally, the KIV system offers well-developed proof engineering facilities: Proof obligations are generated automatically. When applied to UPML, the architectural constraints from section 4.1 are the basis for such automatically generated proof obligations. 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, human intervention 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. The use of the interactive theorem prover KIV [Reif, 1995] for verifying architectural descriptions of knowledge-based systems is described in [Fensel & Schönegge, 1997], [Fensel & Schönegge, 1998], [Fensel et al., 1998d].

To give an impression of how KIV works we include a screen dump of the KIV system in Figure 25. 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 expand the proof tree. Commands include the selection of heuristics, backtracking, pruning the proof tree, saving the proof, etc.

Ontobroker found the following:

V1 = task
V2 = "instance_00003"
V4 = "Complete and parsimonious diagnosis"
V5 = pragmatics
V8 = explanation
V9 = "The task asks for a complete and minimal diagnoses"
V10 = author
V11 = "Dieter Fensel"
V12 = last_date_of_modification
V13 = "May 2, 1998"
V14 = reference
V15 = "D. Fensel: Understanding, Developing and Reusing Problem-Solving Methods. Habilitation, Fakultaet f|r Wirtschaftswisse
V16 = URL
V17 = " http://www.psm-library.com "
V18 = postconditions_goal
V19 = "complete(diagnosis, observations) & parsimonious(diagnosis)"
V20 = preconditions
V21 = "observations <> empty"
V22 = assumptions
V23 = "If we receive input there must be a complete hypothesis. observations <> empty -> Exists H complete(H, observations)"
V24 = roles
V25 = "observations"
V26 = "diagnosis"
V27 = imported_ontologies
V28 = "instance_00004"
V29 = name
V30 = "diagnoses"
V32 = "instance_00046"
V34 = "instance_00150"
V35 = "instance_00153"
V36 = "instance_00154"
V37 = "instance_00155"
V38 = "instance_00151"
V39 = "instance_00001"
V40 = "instance_00152"

V1 = task
V2 = "instance_00003"
V4 = "Complete and parsimonious diagnosis"
V5 = pragmatics
V8 = explanation
V9 = "The task asks for a complete and minimal diagnoses"
V10 = author
V11 = "Dieter Fensel"
V12 = last_date_of_modification

Fig. 24 The answer of Ontobroker^{UPML} for a query.

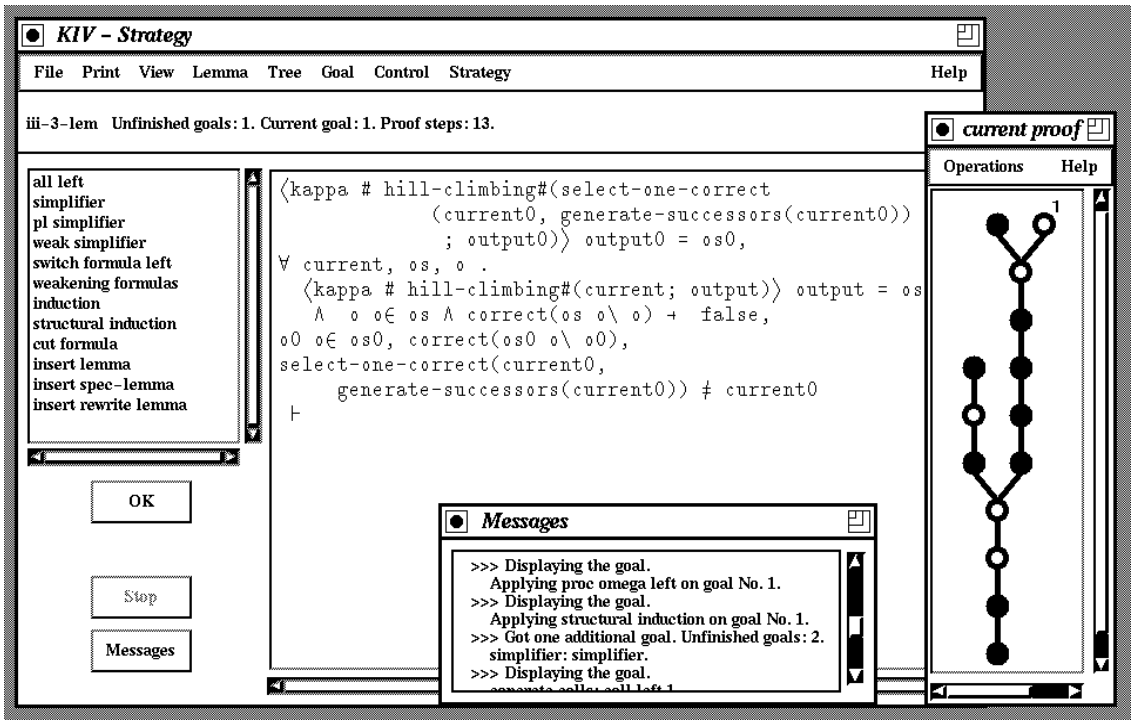

Fig. 25 Verifying a PSM with KIV.

5 Conclusions, Related Work and Future Work

This paper presents UPML, a language for the architectural specifications of knowledge-based systems. *Tasks, problem-solving methods, domain models,* and *ontologies* are the different components. *Refiners* and *bridges* model refinement and the combination of components. Refiners structure the development and representation of methods and their various variants. Bridges enable the user to represent methods decoupled from domains and tasks to enable their reusable description and implementation. In the following, we will compare UPML with related work and we provide an outlook of future work.

5.1 UPML As A Software Architecture

In this section, we will rephrase the different elements of a design model for knowledge-based systems in terms of software architectures.

A *task* defines the problem that is supposed to be solved by the system. For complexity reasons this desired functionality may differ from the functionality actually provided by the system. Distinguishing the desired functionality from the actual competence of the system provides the advantage to have an explicit notion for this gap (cf. [Fensel & Benjamins, 1998b]). A second particular feature of a task definition is its domain-independency. This enables reuse of problem definitions in different domains. Classification tasks, diagnostic tasks, and design tasks can be defined independently from the domain in which they are reused. The (well proven) assumption is that there are problem types that appear in different domains. For example, problems solved by model-based diagnosis appear in a broad variety of domains (electronic circuits, fluid systems, copying machines, etc.) and the same type of design problem appears for office allocation problems, elevator design, sliding bearing design, problems of simple mechanics, initial vehicle (truck) design, design and selection of casting technologies, and sheet metal forming technology for manufacturing mechanical parts, etc. In consequence, a task does not only introduce a goal (i.e., a notion of the desired functionality) but also a generic description of the type of domains it can be instantiated to. Its requirements on domain knowledge provide a domain-independent characterization of its domain-dependency.

A *complex problem-solving method* describes different aspects of a software architecture introducing modularization and control into otherwise declarative systems. The *operational* description of a problem-solving method defines:

- A number of tasks and inferences that corresponds to components in software architectures (where the component that solves a task also has an internal architectural structure);

- a control regime, which defines a script that regulates the order of execution for the components; and

- intermediate roles that provide the data flows between the components. That is, these dynamic knowledge roles correspond to adapters and connectors in software architectures.

A problem-solving method abstracts from the domains it is applied to. The primitive problem-solving methods introduce requirements on domain knowledge which are generic characterizations of domain types it can be successfully applied to. A particular feature of problem-solving methods is therefore that they assume large amounts of the functionality as provided by this domain knowledge. They assume a kind of (deductive) database that provides significant factual knowledge and reasoning services to perform a task. It decomposes the entire task into smaller parts and defines some script that regulates their order of execution, however, the main inference service is assumed to be provided by domain knowledge.

The *domain model* corresponds to a deductive database enriched by integrity constraints (the meta-level characterizations) and assumptions that provide additional axioms for reasoning. For example, the complete-fault-knowledge assumptions allows us to infer a cause from a domain knowledge base if no other known fault is consistent with the observed data.

Finally, *bridges* that glue tasks, problem-solving methods and domain models together are nothing more than adapters and connectors in software architectures. However, the systematic study of refining architectures via *refiners* has yet to be done in the work on software architectures.

Putting UPML in the general context of software architectures allows the reuse of existing work in this area that relates description languages for software architectures with UML. [Mevidovic & Rosenblum, 1999] compare an approach of software architectures with UML at a conceptual level and [Hofmeister et al., 1999] describe a pragmatic way of how UML can be used (or more concrete, *extended*) to model software architectures. Basically, UML allows the introduction of new modeling elements and therefore it is not really difficult to express ones framework in it. [Mevidovic & Rosenblum, 1999] have a very interesting approach by directly relating the concepts of a software architecture with key concepts (like classes of UML). Actually they used built-in extension mechanisms on existing meta-classes rather than extending the meta model itself. Their message in a nutshell is that such an approach is possible, however:

- it requires some artificial modeling (UML makes some implicit assumptions—mainly related to OO programming that are not necessary from a SA point of view—and also the reverse holds, i.e., architectural styles make many assumptions that need to made

explicitly be modeling them in UML),

- some conceptual distinctions get lost (for example, components and connectors are both modeled as classes), and

- UML lacks the notion of hierarchical refinement (somewhat surprisingly).

5.2 Related Approaches in Knowledge Engineering

UPML is close in spirit to CML was been developed in CommonKADS project (cf. [Schreiber et al., 1994]). CML provides a layered conceptual model of knowledge-based systems by distinguishing domain, inference, and task layers according to the CommonKADS model of expertise. UPML took this model as a starting point, but refined it in the component oriented style of software architectures. UPML decomposes a knowledge-based system via an architecture in a set of related elements: tasks, problem-solving methods, domain models, and bridges that establish their relationships. CML does not provide task-independent specifications of problem-solving methods nor the encapsulation mechanism of UPML for problem-solving methods. The operational specification of a method is an internal aspect that is externally captured in its functionality by the competence of the method in UPML. Finally, CML provides no means to refine tasks and problem-solving methods. In general, UPML is much more oriented to problem-solving method reuse (i.e., component reuse) than CML. Finally, CML is a semiformal language whereas UPML can be used as a semiformal language (using its structuring primitives) and as a formal language (UPML supports logical formalisms to formally define the slots).

UPML also has many similarities with other standardization efforts in the area of knowledge-based systems. OKBC [Chaudhri et al., 1998], jointly developed at SRI International and Stanford University, provides a set of functions that support a generic interface to underlying frame representation systems. The *Knowledge Interchange Format* [KIF] is a computer-oriented first-order logic language for the interchange of knowledge among disparate programs. The *Knowledge Query and Manipulation Language* [KQML] is a language and protocol for exchanging information and knowledge. KQML can be used as a language for an application program to interact with an intelligent system or for two or more intelligent systems to share knowledge in support of cooperative problem solving. The distinctive feature of UPML is that it deals with the sharing and exchange of problem-solving methods, i.e. software components that realize complex reasoning tasks of knowledge-based systems.

5.3 Outlook

UPML is a description language and therefore does not need an operational semantics. However, the IBROW^3 broker [Benjamins et al., 1998] must be able to reason about expressions in the description language. In that sense, one can view the broker as a special-purpose „interpreter“ of the language. Also, we require some kind of more general inference engine for the language to establish some reliability of descriptions and the derivation of abstract properties, for example ensuring deadlock freedom of a combination of some components or ensuring that a problem-solving method, together with assumptions, is able to achieve a goal (for this latter task, the architectural constraints from section 4.1 can be exploited). The use of formal techniques for component retrieval is discussed e.g. in [Penix et al., 1997], [Schumann & Fischer, 1997], and [Zaremski & Wing, 1997]. In addition to (internal) reasoning service of the broker, it must interact with a client to select, combine, and adapt a problem solver. The hypertext metaphor seems appropriate as the interaction style of the WWW-based broker. [Isakowitz & Kauffman, 1996] discuss software component reuse as a hypertext-based browsing process.

Acknowledgment. We thank John Gennari, Karima Messaadia, Mourad Oussalah, John Park, Rainer Perkun, Annette ten Teije, and Andre Valente for valuable comments on early drafts of the paper. Thanks to Jeff Butler for correcting our English.

6 References

[Akkermans et al., 1993] 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 Artificial Intelligence (LNAI) 723, Springer-Verlag, Berlin, 1993.

[Allen & Garlan, 1997] R. Allen and D. Garlan: A Formal Basis for Architectural Connection, *ACM Transactions on Software Engineering and Methodology*, 6(3):213—249, July 1997.

[Arcos & Plaza, 1996] J. L. Arcos and E. Plaza: Inference and reflection in the object-centered representation language Noos, *Future Generation Computer Systems Journal, Special issue on Reflection and Meta-level AI Architectures*, 12(2-3):173—188, 1996.

[Benjamins, 1995] V. R. Benjamins: Problem Solving Methods for Diagnosis And Their Role in Knowledge Acquisition, *International Journal of Expert Systems: Research and Application*, 8(2):93—120, 1995.

[Benjamins et al., 1998] V. R. Benjamins, E. Plaza, E. Motta, D. Fensel, R. Studer, B. Wielinga, G. Schreiber, Z. Zdrahal, and S. Decker: An Intelligent Brokering Service for Knowledge-Component Reuse on the World-WideWeb. In *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´98)*, Banff, Canada, April 18-23, 1998.

[Benjamins et al., 1999] V. R. Benjamins, B. Wielinga, J. Wielemaker, and D. Fensel : Brokering Problem-Solving Knowledge at the Internet. In *Knowledge Acquisition, Modeling, and Management, Proceedings of the European Knowledge Acquisition Workshop (EKAW-99)*, D. Fensel et al. (eds.), Lecture Notes in Artificial Intelligence, LNAI 1621, Springer-Verlag, May 1999.

[Benjamins & Fensel, 1998] V. R. Benjamins and D. Fensel: Special issue on problem-solving methods of the *International Journal of Human-Computer Studies (IJHCS)*, 49(4):305-313,1998.

[Benjamins & Shadbolt, 1998] V. R. Benjamins and Nigel Shadbolt: Special Issue on Knowledge Acquisition and Planning, *International Journal of Human-Computer Studies (IJHCS)*, 48(4), 1998.

[Beys et al., 1996] P. Beys, R. Benjamins, and G. van Heijst: Remedying the Reusability-Usability Trade-off for Problem-solving Methods. In *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´96)*, Banff, Canada, November 9-14, 1996.

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

[Bylander et al., 1991] T. Bylander, D. Allemang, M. C. Tanner, and J. R. Josephson: The Computational Complexity of Abduction, *Artificial Intelligence*, 49:25—60, 1991.

[Chaudhri et al., 1998] V. K. Chaudhri, A. Farquhar, R. Fikes, P. D. Karp, and J. P. Rice: *Open Knowledge Base Connectivity 2.0*, Knowledge Systems Laboratory, KSL-98-06, January 1998. http://www-ksl-svc.stanford.edu:5915/doc/project-papers.html

[Chandrasekaran et al., 1992] B. Chandrasekaran, T.R. Johnson, and J. W. Smith: Task Structure Analysis for Knowledge Modeling, *Communications of the ACM*, 35(9):124—137, 1992.

[Chandrasekaran et al., 1998] B. Chandrasekaran, J. R. Josephson, and R. Benjamins: The Ontology of Tasks and Methods, In *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´98)*, Banff, Canada, April 18-23, 1998.

[Chaudhri et al., 1998] V. K. Chaudhri, A. Farquhar, R. Fikes, P. D. Karp, and J. P. Rice: OKBC: A programmatic foundation for knowledge base interoperability. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98*), pages 600–607. AAAI Press, 1998.

[Clancey, 1983] W.J. Clancey, The Epistemology of a Rule-Based Expert System—a Framework for Explanation, *Artificial Intelligence* 20:215—251, 1983.

[Clements, 1996] P. C. Clements: A Survey of Architecture Description Languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, Dagstuhl, Germany, March 1996.

[Cox & Pietrzykowski, 1986] P. T. Cox and T. Pietrzykowski: Causes of Events: Their Computation and Application. In *Proceedings of the 8th International Conference on Automated Deduction*, Oxford, England, July 27 - August 1, Lecture Notes in Computer Science (LNCS) 230, Springer-Verlag, 1986.

[Decker et al., 1999] S. Decker, M. Erdmann, D. Fensel, and R. Studer: Ontobroker: Ontology based Access to Distributed and Semi-Structured Information. In R. Meersman et al. (eds.), *Semantic Issues in Multimedia Systems*, Kluwer Academic Publisher, Boston, to appear 1999.

[de Kleer, 1986] J. de Kleer: An Assumption-based TMS, *Artificial Intelligence*, 28, 1986.

[Dijkstra, 1975] E. W. Dijkstra: Guarded Commands, Nondeterminancy, and Formal Derivation of Programs, *Communication of the ACM*, 18:453—457, 1975.

[Eriksson et al., 1995] 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.

[Eriksson et al., 1999] H. Eriksson, R. W. Fergerson, Y. Shahar, and M. A. Musen: Automatic Generation of Ontology Editors. In *Proceedings of the Twelfth Banff Knowledge Acquisition for Knowledge-based systems Workshop*, Banff, Alberta, Canada, 1999.

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

[Fensel, 1997] D. Fensel: The Tower-of-Adapter Method for Developing and Reusing Problem-Solving Methods. In E. Plaza et al. (eds.), *Knowledge Acquisition, Modeling and Management*, Lecture Notes in Artificial Intelligence (LNAI) 1319, Springer-Verlag, 1997.

[Fensel, 2000] D. Fensel: *Understanding, Development, Description, and Reuse of Problem-Solving Methods*, Lecture Notes in Artificial Intelligence (LNAI), Springer-Verlag, 2000.

[Fensel & Benjamins, 1998a] D. Fensel and V. R. Benjamins: Key Issues for Problem-Solving Methods Reuse. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, Brighton, UK, August 1998.

[Fensel & Benjamins, 1998b] D. Fensel and R. Benjamins: The Role of Assumptions in Knowledge Engineering, *International Journal of Intelligent Systems (IJIS)*, 13(7), 1998.

[Fensel et al., 1996a] D. Fensel, H. Eriksson, M. A. Musen, and R. Studer: Conceptual and Formal Specification of Problem-Solving Methods, *International Journal of Expert Systems*, 9(4), 1996.

[Fensel et al., 1997] D. Fensel, E. Motta, S. Decker, and Z. Zdrahal: Using Ontologies For Defining Tasks, Problem-Solving Methods and Their Mappings. In E. Plaza et al. (eds.), *Knowledge Acquisition, Modeling and Management*, Lecture Notes in Artificial Intelligence (LNAI) 1319, Springer-Verlag, 1997.

[Fensel et al., 1998a] D. Fensel, R. Groenboom, and G. R. Renardel de Lavalette: Modal Change Logic (MCL): Specifying the Reasoning of Knowledge-based Systems, *Data and Knowledge Engineering (DKE)*, 26(3):243-269, 1998.

[Fensel et al., 1998b] D. Fensel, J. Angele, and R. Studer, The Knowledge Acquisition and Representation Language KARL, *IEEE Transactions on Knowledge and Data Engineering*, 10(4):527-550, 1998.

[Fensel et al., 1998c] D. Fensel, S. Decker, M. Erdmann und R. Studer: Ontobroker: The Very High Idea. In *Proceedings of the 11th International Flairs Conference (FLAIRS-98)*, Sanibal Island, Florida, USA, 131-135, May 1998.

[Fensel et al., 1998d] D. Fensel, F. van Harmelen, W. Reif und Annette ten Teije: Formal Support for Development of Knowledge-Based Systems, *Information Technology Management: An International Journal*, special issue on Lessons Learned About Safety-Critical Software, 1998.

[Fensel et al., 1999a] D. Fensel, V. R. Benjamins, S. Decker, M. Gaspari, R. Groenboom, W. Grosso, M. Musen, E. Motta, E. Plaza, G. Schreiber, R. Studer, and B. Wielinga: The Component Model of UPML in a Nutshell. In *WWW Proceedings of the 1st Working IFIP Conference on Software Architectures (WICSA1)*, San Antonio, Texas, USA, February 1999.

[Fensel et al., 1999b] D. Fensel, E. Motta, V. R. Benjamins, S. Decker, M. Gaspari, R. Groenboom, W. Grosso, M. Musen, E. Plaza, G. Schreiber, R. Studer, and B. Wielinga: The Unified Problem-solving Method Development Language UPML. In ESPRIT Projekt 27169 IBROW3: An Intelligent Brokering Service for Knowledge-Component Reuse on the World-Wide Web, Deliverable 1.1, Chapter 1.

[Fensel et al., 1999c] D. Fensel, V. R. Benjamins, E. Motta, and B. Wielinga: UPML: A Framework for knowledge system reuse. In *Proceedings of the International Joint Conference on AI (IJCAI-99)*, Stockholm, Sweden, July 31 - August 5, 1999.

[Fensel et al., 1999d] D. Fensel, J. Angele, S. Decker, M. Erdmann, H.-P. Schnurr, S. Staab, R. Studer, and A. Witt: On2broker: Semantic-Based Access to Information Sources at the WWW. In *Proceedings of the World Conference on the WWW and Internet (WebNet 99)*, Honolulu, Hawai, USA, October 25-30, 1999.

[Fensel & Groenboom, 1996] D. Fensel and R. Groenboom: MLPM: Defining 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.

[Fensel & Groenboom, 1997] 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.

[Fensel & Groenboom, 1999] D. Fensel and R. Groenboom: A Software Architecture for Knowledge-Based Systems, *The Knowledge Engineering Review*, 14(3), 1999.

[Fensel & van Harmelen, 1994] 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.

[Fensel & Motta, 1998] D. Fensel and E. Motta: Structured Development of Problem Solving Methods, In *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´98)*, Banff, Canada, April 18-23, 1998.

[Fensel & Motta, to appear] D. Fensel and E. Motta: Structured Development of Problem Solving Methods, to appear in *IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE)*.

[Fensel & Schönegge, 1997] D. Fensel and A. Schönegge: Using KIV to Specify and Verify Architectures of Knowledge-Based Systems. In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering (ASEC-97)*, Incline Village, Nevada, November 3-5, 1997.

[Fensel & Schönegge, 1998] D. Fensel and A. Schönegge: Inverse Verification of Problem-Solving Methods, *International Journal of Human-Computer Studies (IJHCS)*, 49(4):339-362,1998.

[Fensel & Straatman, 1998] D. Fensel and R. Straatman: The Essence of Problem-Solving Methods: Making Assumptions to Gain Efficiency, to appear in *The International Journal of Human Computer Studies (IJHCS)*, 48(2):181—215, 1998.

[Gamma et al., 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: *Design Patterns*, Addison-Wesley Pub., 1995.

[Gennari et al., 1998] J. H. Gennari, W. Grosso, and M. Musen: A Method-Description Language: An Initial Ontology with Examples. In *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´98)*, Banff, Canada, April 1998.

[de Geus & Rotterdam, 1992] F. de Geus and E. Rotterdam: *Decision Support in Anesthesia*, Ph.D. thesis, University of Groningen, 1992.

[Groenboom, 1997] R. Groenboom: *Formalizing Knowledge Domains - Static and Dynamic Aspects*, Ph.D. thesis, University of Groningen, Shaker Publ., 1997.

[Grosso et al., 1999] W. E. Grosso, H. Eriksson, R. W. Fergerson, J. H. Gennari, S. W. Tu, and M. A. Musen: Knowledge Modeling at the Millennium (The Design and Evolution of Protégé-2000). In *Proceedings of the Twelfth Workshop on Knowledge Acquisition, Modeling and Management (KAW99)*, Banff, Alberta, Canada, October 16-21, 1999.

[Gruber, 1993] T. R. Gruber: A Translation Approach to Portable Ontology Specifications, *Knowledge Acquisition*, 5:199—220, 1993.

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

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

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

[Hofmeister et al., 1999] C. Hofmeister, R. L. Nord, and D: Soni: Describing Software Architectures with UML. In P. Donohoe (eds.), *Software Architecture*, Kluwer Academic Publ., 1999.

[Isakowitz & Kauffman, 1996] T. Isakowitz and R. J. Kauffman: Supporting Search for Reusable Software Objects, IEEE Transactions on Software Engineering, 22(6):407—423, 1996.

[Karbach & Voß, 1992] W. Karbach and A. Voß: Reflecting About Expert Systems in MODEL-K. In *Proceedings of Expert Systems and their Applications, 12th International Workshop, vol 1*

(Scientific Conference), June 1-6, Avignon, 1992.

[Keisler, 1977] H. J. Keisler: Fundamentals of Model Theory. In John Barwise (ed.), *Handbook of Mathematical Logic*, North Holland 1977.

[KIF] KIF: http://logic.stanford.edu/kif/kif.html.

[Kifer et al., 1995] M. Kifer, G. Lausen, and J. Wu: Logical Foundations of Object-Oriented and Frame-Based Languages, *Journal of the ACM*, vol 42, 1995.

[KQML] KQML: http://www.cs.umbc.edu/kqml/.

[van Langevelde et al., 1993] I. van Langevelde, A. Philipsen, and J. Treur: A Compositional Architecture for Simple Design Formally Specified in DESIRE. In J. Treur and Th. Wetter (eds.): *Formal Specification of Complex Reasoning Systems*, Ellis Horwood, New York, 1993.

[Lloyd, 1987] J. W. Lloyd: Declarative Error Diagnosis, *New Generation Computing*, 5:133—154, 1987.

[Marcus, 1988] S. Marcus (ed.): *Automating Knowledge Acquisition for Experts Systems*, Kluwer Academic Publisher, Boston, 1988.

[Mevidovic & Rosenblum, 1999] N. Mevidovic and D. S. Rosenblum: Accessing the Suitability of a Standard Design Method for Modeling Software Architectures. In P. Donohoe (eds.), *Software Architecture*, Kluwer Academic Publ., 1999.

[Minton, 1995] S. Minton: Quantitative Results Concerning the Utility of Explanation-Based Learning. In A. Ram and D. B. Leake (eds.): *Goal-Driven Learning*, The MIT Press, 1995.

[Minton et al., 1989] S. Minton, S. Carbonell, C. Knoblock, D. R. Kuokka, O. Etzioni, and Y. Gil: Explanation-based Learning: A Problem Solving Perspective, *Artificial Intelligence*, 40:63—118, 1989.

[Mizoguchi et al., 1995] R. Mizoguchi, J. Vanwelkenhuysen, and M. Ikeda: Task Ontologies for reuse of Problem Solving Knowledge. In N. J. I. Mars (ed.), *Towards Very Large Knowledge Bases*, IOS Press, 1995.

[Motta, 1999] E. Motta: *Reusable Components for Knowledge Modeling*, IOS Press, Amsterdam, 1999.

[Motta et al., 1998] E. Motta, M. Gaspari, and D. Fensel: *UPML Specification of a Parametric Design Library*, Deliverable D4.1. Esprit Project 27169, IBROW3, 1998.

[Muggleton & Buntine, 1988] S. Muggleton and W. Buntine: Machine Invention of First-Order Predicates by Inverting Resolution. In *Proceedings of the 5th International Conference on Machine Learning (ICML-88)*, Michigan, US, 1988.

[Muggleton & De Raedt, 1994] S. Muggleton and L. De Raedt: Inductive Logic Programming: Theory and Methods, *Journal of Logic Programming*, 19/20:629—679, 1994.

[Musen 1998] M. A. Musen. Modern Architectures for Intelligent Systems: Reusable Ontologies and Problem-Solving Methods. In C.G. Chute, Ed., 1998 *AMIA Annual Symposium*, Orlando, FL, 46-52. 1998.

[O'Hara & Shadbolt, 1996] K. O'Hara and N. Shadbolt: The Thin End of the Wedge: Efficiency and the Generalized Directive Model Methodology. In N. Shadbolt (eds.), *Advances in Knowledge Acquisition*, LNAI 1076, Springer-Verlag, 1996.

[Oussalah & Messaadia, 1999] M. Oussalah and K. Messaadia: The Ontologies of Semantic and Transfer Links. In D. Fensel et al. (eds.), *Proceedings of the EKAW-99*, Lecture Notes on Artificial Intelligence (LNAI), Springer-Verlag Berlin, 1999.

[Penix et al., 1997] J. Penix, P. Alexander, and K. Havelund: Declarative Specification of Software Architectures. In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering (ASEC-97)*, Incline Village, Nevada, November 3-5, 1997.

[Puerta et al., 1992] A. R. Puerta, ‚J. W. Egar, S. W. Tu, M.A. and Musen: A Multiple-method Knowledge-Acquisition Shell for the Automatic Generation of Knowledge-acquisition Tools, *Knowledge Acquisition*, 4(2):171—196, 1992.

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

[Reif, 1995] 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) 1009, Springer-Verlag, Berlin, 1995.

[Schreiber et al., 1994] 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.

[Schumann & Fischer, 1997] J. Schuman and B. Fischer: NORA/HAMMER: Making Deduction-Based Software Component Retrieval Practical. In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering (ASEC-97)*, Incline Village, Nevada, November 3-5, 1997.

[Shapiro, 1982] E. Y. Shapiro: *Algorithmic Program Debugging*, The MIT Press, 1982.

[Shaw & Garlan, 1996] M. Shaw and D. Garlan: S*oftware Architectures. Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

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

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

[Stefik, 1995] M. Stefik: *Introduction to Knowledge Systems*, Morgan Kaufman Publ., San Francisco, 1995.

[Studer et al., 1996] R. Studer, H. Eriksson, J. Gennari, S. Tu, D. Fensel and M. Musen: Ontologies and the Configuration of Problem-Solving Methods. In *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´96)*, Banff, Canada, November 1996.

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

[Van de Velde, 1988] 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.

[Van de Velde, 1994] W. van de Velde: A Constructive View on Knowledge Engineering. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, Budapest, Hungaria, August , 1994.

[Voß & Voß, 1993] H. Voß and A. Voß: Reuse-Oriented Knowledge Engineering with MoMo. In *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering (SEKE´93)*, San Francisco Bay, June 14-18, 1993.

[Wiederhold, 1992] G. Wiederhold: Mediators in the Architecture of Future Information Systems,

IEEE Computer, 25(3):38—49, 1992.

[Yellin & Strom, 1997] D. M. Yellin and R. E. Strom: Protocol Specifications and Component Adapters, ACM Transactions on Programming Languages and Systems, 19(2):292—333, 1997.

[Zaremski & Wing, 1997] A. M. Zaremski and J. M. Wing: Specification Matching of Software Components, ACM Transactions on Software Engineering and Methodology, 6(4):335—369, 1997.