OIL & UPML: A Unifying Framework for the Knowledge Web

D. Fensel[1], M. Crubezy[2], F. van Harmelen[1], and M. I. Horrocks[3],

[1] Department of Computer Science, Vrije Universiteit Amsterdam, {dieter,frankh}@cs.vu.nl

[2] Knowledge Modeling Group at Stanford Medical Informatics, Stanford University USA, crubezy@SMI.Stanford.Edu

[3] Department of Computer Science, University of Manchester, UK, horrocks@cs.man.ac.uk

Abstract. Currently computers are changing from single isolated devices to entry points in a world wide network of information exchange and business transactions called the World Wide Web (WWW). A prerequisite for successfully integrating various information sources are standardized and machine-processable descriptions of their semantics. In this paper, we will briefly describe two proposals and will discuss how both can be combined. First, we discuss OIL that is proposed as description language for ontology interchange. That is, it is designed for specifying static information. Second, we sketch UPML, which is developed for describing reasoning components. UPML helps to automatically configure distributed reasoning components that can be used as inference service via networks. Integrating both description types is a necessary step in the direction of a **knowledge web** where the distinction between static and dynamic information sources will become transparent for the user. The main contribution of the paper is the comparison of both approaches. We achieve this comparison by discussing severals ways of combining OIL and UPML. We analyze the meaning of each perspective and stress what enhancements would be necessary to improve their usefulness.

1 Introduction

Support in data, information, and knowledge exchange is becoming a key issue in current computer technology. Given the exponential growth of on-line available information, automatic processing of this information becomes necessary for keeping it maintainable and accessible. Providing shared and common domain structures becomes essential. Being used to describe the structure and semantics of information exchange, ontologies will become a key asset in information exchange. Such technologies will play a key role in areas such as knowledge management and electronic commerce, which are market places which incredible growth rates in the near future. Information sources may not only be passive entities. Instead, active software components will be used as services via networks. These components do not only provide support in information retrieval and extraction but also provide direct support in task achievement. Again, machine-understandable representation of their semantics is required for the automated selection and combination of these reasoning services. Therefore, it is natural that a number of proposals and projects deal with these concerns. In the US, research fundings agencies have already encountered the importance of such an issues by setting up the DAML program[1] that aims for machine processable semantics of information sources accessible for agents.

Already the Worldwide Web (WWW) has drastically changed the availability of electronically available information. This first generation of the World Wide Web has already changed our daily practice and these changes will become even more significant in the near future. However, the web itself will have to change when it should reach the next level of service. Currently, the Web is an incredible large mainly static information source. The main burden in information access, extraction and interpretation, however, is left to the human user. Tim Berners-Lee coined the vision of a Semantic Web that provides much more automated services based on machine-processable semantics of data and heuristics that make use of these meta data. The explicit representation of the semantics of data accompanied with domain theories (i.e., Ontologies) will enable a Knowledge Web that provides a qualitatively new level of service. It weaves together a net linking incredible large parts of the human knowledge and complements it with machine processability. Various automated services will support the human user in achieving goals via accessing and providing information present in a machine-understandable form. This process will ultimately lead to an extremely knowledgeable system with various specialized reasoning services that may support us in nearly all aspects of our daily life becoming as central as access to electric power. **For this knowledge web it is important to link together semantic descriptions of information sources with semantic descriptions of heuristic reasoners using these information sources.** Especially because we expect that the difference between both will become transparent for the human user, i.e., it does not make any difference to him whether a browser renders a static information source or a virtual page that is generated on the fly.

In this paper we will compare two proposals developed in relation to two European IST projects.

- The **On-To-Knowledge project**[2] applies ontologies to electronically available information to improve the quality of knowledge management in large and distributed organizations. Ontologies are used to explicitly represent semantics of semi-structured information. This enables sophisticated automatic support for acquiring, maintaining, and accessing information. In cooperation with other external partners *OIL* has been developed (cf. [Fensel et al., to appear (b)], [Horrocks et al., to appear]) that will be used to define and exchange ontologies between heterogeneous and distributed information sources.
- The objective of the **Ibrow project**[3] ([Benjamins et al., 1999], [Fensel & Benjamins, 1998]) is to develop intelligent brokers that are able to distributively configure reusable components into knowledge-based systems through the World-Wide Web. The WWW is changing the nature of software development to a distributive plug & play process, which requires a new kind of managing software: intelligent software brokers. Ibrow will integrate research on heterogeneous databases, interoperability and Web technology with knowledge-system technology and ontologies. A result of Ibrow has been the development of a specification language for reasoning components called *UPML* (cf. [Fensel et al., 1999]).

It is quite natural to compare the languages OIL and UPML developed in the two projects. One should expect many similarities because reasoning components could be viewed as active information sources, i.e., as components providing some information as a result of some input. Taking a more

1. http://www.darpa.mil/iso/ABC/BAA0007PIP.htm.

2. *On-To-Knowledge: Content-driven Knowledge-Management Tools through Evolving Ontologies.* Project partner are the Vreije Universiteit Amsterdam (VU); the Institute AIFB, University of Karlsruhe, Germany; AIdministrator, the Netherlands; British Telecom Laboratories, UK; Swiss Life, Switzerland; CognIT, Norway; and Enersearch, Sweden. http://www.ontoknowledge.com/

3. *IBROW: An Intelligent Brokering Service for Knowledge-Component Reuse on the World-Wide Web.* 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/ibrow/home.html

detailed look on their relationships it turns out that there are at least six possible ways on how to try to combine both languages: We can ask what OIL can be providing for UPML, and we can ask what UPML can be providing for OIL. Each of these two cases comes along with three subpossibilities.

- **First, OIL can be used as a meta-language to define UPML.** A language like UPML could be viewed as a specific ontology where the language primitives are concepts to talk about a certain domain. In this case this domain is the description of reasoning components. OIL should be capable for such a purpose because it must be possible to express an ontology in it. Here we will examine, how the language primitives of UPML can be expressed in OIL. We use OIL in a similar way like the meta-meta model of MOF[4] is used to express the meta model of other modeling frameworks. A meta ontology of UPML has already been described in [Fensel et al., 1999] and we will examine how it can be expressed in OIL.
- **Second, OIL can be used as a language for writing down UPML specifications.** Here a component specification in UPML should correspond to an ontology in OIL. Therefore, several components should be represented via several ontologies, each for one component. Viewing the specifications of reasoning components as ontologies has been proposed in [Mizoguchi et al., 1995] and we will examine how OIL fits for this purpose.
- **Third, OIL can be used as an object language for UPML.** Mainly, UPML defines an architecture for the description of reasoning components but has not yet provided a defined language for defining the elementary units of a component. Currently, it provides three different styles: natural language definitions (like CML, [Schreiber et al., 1994]), order-sorted logic (like $(ML)^2$ [van Harmelen & Balder, 1992]), and frame logic (like KARL, [Fensel et al., 1998]). In this paper, we will examine how OIL could fill in the gap as a defined standard language for the logical specification of the elementary elements of a UPML specification.
- **Fourth, can UPML be used as a meta-language to define OIL?** A language like OIL could be viewed as a specific ontology where the language primitives are concepts to talk about a certain domain. In this case this domain would be the specification of ontologies. In principle, UPML would be applicable for such a purpose because one out of its six components are ontologies. We could define an ontology in UPML defining the language primitives of OIL. However it is not clear what we would gain from such an exercise. Therefore, we will not examine this possibility further on.
- **Fifth, UPML can be used as a language for writing down ontologies in OIL.** Here an ontology in OIL corresponds to an ontology in UPML. This looks interesting because it would provide the structuring mechanisms of UPML for OIL ontologies. Currently, OIL does only provide an import mechanism to combine ontologies. UPML provides bridges and refiners to combine and adapt ontologies. Following this combination strategy delivers an architectural structure on top of OIL.
- **Sixth, can UPML be used as an object language for OIL?** No, this does not make any sense. OIL is already a language and has no undefined elementary slots that require further logical refinement. Therefore, we will not examine this possibility further on.

The contents of the paper are organizes as follows. In Section 2, we provide a brief introduction to OIL and in Section 3 we provide a brief introduction into UPML. Both sections are necessary to keep the paper self-contained. Section 4 provides the actual contribution of the paper. We will investigate four different strategies to relate OIL and UPML. We provide conclusions in Section 5.

[4]. The Meta Object Facility (MOF) standard is a proposal of the OMG's group for expressing various modeling frameworks in a joined representation (cf. [OMG, 1997]).

2 OIL

Ontologies are a popular research topic in various communities such as knowledge engineering, natural language processing, cooperative information systems, intelligent information integration, knowledge management. They provide a shared and common understanding of a domain that can be communicated across people and application systems. They have been developed in Artificial Intelligence to facilitate knowledge sharing and reuse. Recent articles covering various aspects of ontologies can be found in [Uschold & Grüninger, 1996], [van Heijst et al., 1997], [Gomez Perez & Benjamins, 1999], [Fensel, to appear (b)]. Ontologies are a good candidate for providing the shared and common domain structures which are required for truly semantic integration of information sources. The question then becomes: *how to describe and exchange such ontologies?* A prerequisite for such a widespread use of ontologies for information integration and exchange is the achievement of a joint standard for describing ontologies. Take the area of databases as an example. The huge success of the relational model would have never been possible without the SQL standard that provided an implementation independent way for storing and accessing data. Any approach that tries to achieve such a standard for the areas of ontologies has to answer the questions on what are the appropriate modeling primitives for representing ontologies and how to define their semantics, as well as, what is the appropriate syntax for representing ontologies.

[Horrocks et al., to appear] defines the *Ontology Interchange Language (OIL)* as a standard proposal. In this section we will give a brief description of the OIL language; more details can be found in [Horrocks et al., to appear] and [Fensel et al., to appear (b)]. A small example ontology in OIL is provided in Figure 1. This language has been designed so that: (1) It provides most of the modeling primitives commonly used in frame-based and Description Logic (DL) oriented Ontologies. (2) It has a simple, clean and well defined semantics. (3) Automated reasoning support, (e.g., class consistency and subsumption checking) can be provided. It is envisaged that this core language will be extended in the future with sets of additional primitives, with the proviso that full reasoning support may not be available for ontologies using such primitives.

An ontology in OIL is represented via an *ontology container* and an *ontology definition* part. We will discuss both elements of an ontology specification in OIL. We start with the ontology container and will then discuss the backbone of OIL, the ontology definition.

Ontology Container: We adopt the components as defined by Dublin Core Metadata Element Set, Version 1.1[5] for the *ontology container* part of OIL.

Apart from the container, an OIL ontology consists of a **set of definitions**:

- **import** A list of references to other OIL modules that are to be included in this ontology. XML schemas and OIL provide the same (limited) means for composing specifications. One can include specifications and the underlying assumptions is that names of different specifications are different (via different prefixes).
- **rule-base** A list of rules (sometimes called axioms or global constraints) that apply to the ontology. At present, the structure of these rules is not defined (they could be horn clauses, DL style axioms etcetera), and they have no semantic significance. The rule base consists simply of a **type** (a string) followed by the unstructured rules (a string).
- **class and slot definitions** A list of zero or more class definitions **(class-def**) and slot definitions

5. http://purl.oclc.org/dc/

(slot-def), the structure of which will be described below.

A class definition **(class-def**) associates a class name with a class description. A **class-def** consists of the following components:

- **type** The type of definition. This can be either **primitive** or **defined**; if omitted, the type defaults to **primitiv**e. When a class is **primitiv**e, its definition (i.e., the combination of the following **subclass-of** and **slot-constraint** components) is taken to be a necessary but not sufficient condition for membership of the class.
- **subclass-of** A list of one or more class-expressions, the structure of which will be described below. The class being defined in this **class-def** must be a sub-class of each of the class expressions in the list.
- **slot-constraint** A list of zero or more **slot-constrain**ts, the structure of which will be described below. The class being defined in this **class-def** must be a sub-class of each of the slot-constraints in the list (note that a slot-constraint defines a class).

A **class-expression** can be either a class name, a **slot-constrain**t, or a boolean combination of class expressions using the operators **AND**, **OR** or **NOT**. Note that class expressions are recursively defined, so that arbitrarily complex expressions can be formed.

```
ontology-container
    title "African animals"
    creator "Ian Horrocks"
    subject "animal, food, vegetarians"
    description "A didactic example ontology describing
    African animals"
    description.release "1.01"
    publisher "I. Horrocks"
    type "ontology"
    format "pseudo-xml"
    format "pdf"
    identifier
        "http://www.cs.vu.nl/~dieter/oil/TR/oil.pdf"
    source "http://www.africa.com/nature/animals.html"
    language "OIL"
    language "en-uk"
    relation.hasPart
        "http://www.ontosRus.com/animals/jungle.onto"

ontology-definitions
    slot-def eats
        inverse is-eaten-by
    slot-def has-part
        inverse is-part-of
        properties transitive
    class-def animal
    class-def plant
        subclass-of NOT animal
    class-def tree
        subclass-of plant
class-def branch
    slot-constraint is-part-of
        has-value tree
class-def leaf
    slot-constraint is-part-of
        has-value branch
class-def defined carnivore
    subclass-of animal
    slot-constraint eats
        value-type animal
class-def defined herbivore
    subclass-of animal, NOT carnivore
    slot-constraint eats
        value-type
            plant OR
            slot-constraint is-part-of plant
class-def giraffe
    subclass-of animal
    slot-constraint eats
        value-type leaf
class-def lion
subclass-of animal
slot-constraint eats
    value-type herbivore
class-def tasty-plant
    subclass-of plant
    slot-constraint eaten-by
        has-value herbivore OR carnivore
```

Fig. 1 An example ontology in OIL

A **slot-constraint** is a list of one or more constraints (restrictions) applied to a **slot**. A slot is a binary relation (i.e., its instances are pairs of individuals), but a slot-constraint is actually a class definition—its instances are those individuals that satisfy the constraint(s). For example, if the pair (Leo; Willie) is an instance of the slot *eats*, Leo is an instance of the class lion and Willie is an instance of the class wildebeest, then Leo is also an instance of the **has-value** constraint wildebeest applied to the slot *eats*. A **slot-constraint** consists of the following main components:

- **name** A slot name (a string). The slot is a binary relation that may or may not be defined in the ontology. If it is not defined it is assumed to be a binary relation with no globally applicable constraints, i.e., any pair of individuals could be an instance of the slot.
- **has-value** A list of one or more **class-expressions**. Every instance of the class defined by the slot constraint must be related via the slot relation to an instance of each **class-expression** in the list. For example, the **has-value** constraint:
  slot-constraint *eats*
  has-value zebra, wildebeest
  defines the class each instance of which *eats* some instance of the class zebra and some instance of the class wildebeest. Note that this does not mean that instances of the slot-constraint eat *only* zebra and wildebeest: they may also be partial to a little gazelle when they can get it.
- **value-type** A list of one or more **class-expressions**. If an instance of the class defined by the slot-constraint is related via the slot relation to some individual *x*, then *x* must be an instance of each **class-expression** in the list.
- **max-cardinality** A non-negative integer n followed by a **class-expression**. An instance of the class defined by the slot-constraint can be related to at most n distinct instances of the **class-expression** via the slot relation.
- **min-cardinality** and, as a shortcut, **cardinality**.

A slot definition **(slot-def**) associates a slot name with a slot description. A slot description specifies global constraints that apply to the slot relation, for example that it is a transitive relation. A **slot-def** consists of the following main components:

- **subslot-of** A list of one or more **slots**. The slot being defined in this **slot-def** must be a sub-slot of each of the slots in the list. For example,
  slot-def *daughter*
  subslot-of *child*
  defines a slot *daughter* that is a subslot of child, i.e., every pair of individuals that is an instance of *daughter* must also be an instance of child.
- **domain** A list of one or more **class-expressions**. If the pair (x; y) is an instance of the slot relation, then x must be an instance of each **class-expression** in the list.
- **range** A list of one or more **class-expressions**. If the pair (x; y) is an instance of the slot relation, then y must be an instance of each **class-expression** in the list.
- **inverse** The name of a slot *S* that is the inverse of the slot being defined. If the pair (x; y) is an instance of the slot *S*, then (y; x) must be an instance of the slot being defined.
- **properties** A list of one or more properties of the slot. Valid properties are: **transitive, symmetric,** and **reflexive**.

The syntax of OIL is oriented towards XML and RDF. [Horrocks et al., to appear] defines a DTD, a

XML schema definition, and a definition of OIL in RDF.

3 UPML

Knowledge-based systems are computer systems that deal with complex problems by making use of knowledge. Making knowledge on how to solve problems efficiently explicit is the rationale that underlies *problem-solving methods (PSMs)* (cf. [Stefik, 1995], [Benjamins & Fensel, 1998], [Benjamins & Shadbolt, 1998], [Fensel, to appear (a)]). Problem-solving methods refine generic inference engines to allow a more direct control of the reasoning process. Problem-solving methods describe this control knowledge independent from the application domain thus enabling 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. PSMs decompose the reasoning task of a knowledge-based system in a number of subtasks and inference actions that are connected by knowledge roles. Therefore PSMs are a special type of software architectures ([Shaw & Garlan, 1996]): *software architectures* for describing the *reasoning* part of knowledge-based systems.

The IBROW project [Benjamins et al., 1999], [Fensel & Benjamins, 1998] has been set up 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 human-understandable high-level descriptions with underpinned formal means to allow automated support by the broker. Therefore, we developed the *Unified Problem-Solving Method description Language UPML* (cf. [Fensel et al., 1999], [Fensel et al., to appear (a)]). UPML is an architectural description language specialized for a specific type of systems providing *components*, *adapters* and a configuration of how the components should be connected using the adapters (called *architectural constraints*).

The UPML architecture for describing a knowledge-based system consists of six different elements: 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** 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. UPML provides two types of adapters: **bridges** and **refiners**. Bridges explicitly model the relationships between two distinguished 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, to appear]). Generic problem-solving methods and tasks can be refined to 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 methods.

In the following we provide a short example providing a task definition together with its ontology. 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 is illustrated in Figure 2 which is used to provide the elements for defining a diagnostic problem. The ontology introduces two elementary sorts *Finding* and *Hypothesis* that later will be grounded in a domain model. The former describes phenomenon and the latter describe 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 < 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 should be solved by the knowledge-based system. In distinction with most approaches in software engineering this

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
      $explain: Hypotheses \rightarrow Findings$
    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. 2 A task ontology for diagnostic problems.

```
task complete and parsimonious diagnoses
   pragmatics
      The task asks for a complete and minimal diagnoses;
      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
         $observations \neq \emptyset$
      assumptions
         If we receive input there must be a complete hypothesis.
            $observations \neq \emptyset \rightarrow \exists H\ complete(H, observations)$;
         Nonreflexivity of <.
            $\neg (H < H)$;
         Transitivity of <.
            $(H < H') \wedge (H' < H'') \rightarrow (H < H'')$;
         Finiteness of H.
            $Finite(H)$
```

Fig. 3 The task specification of a diagnostic task.

problem definition is kept domain independent, which enables the reuse of generic problem definitions for different applications. A second particular feature is the distinction between preconditions on input and assumptions about knowledge. In an abstract sense, both can be viewed as input. However, distinguishing case data, that are processed (i.e., input), from knowledge, that is used to define the goal, reflect a particular 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 the 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 legal 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). A task definition may import ontologies and other tasks. The latter enable hierarchical structuring of task specifications. For example, parametric design can be defined as a refinement of design (cf. [Fensel & Motta, to appear]).

An example of a task specification is given in Fig. 3. 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 input (which is non-empty). 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.

Another important aspect of UPML are architectural constraints that ensure well-defined components and composed systems in UPML. The conceptual model of UPML decomposes the overall specification and verification tasks into subtasks of smaller grainsize and clearer focus. 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, domain models, and PSMs. The constraints for composition are introduced by constraints that apply to bridges. As an example, we provide the constraints for well-defined task definitions. For a task specification we require consistency of a task specification, i.e:

T_1 *ontology axioms* ∪ *preconditions* ∪ *assumptions* must have a model.

Otherwise we would define an inconsistent task specification which would be unsolvable. In addition, it must hold:

T_2 Each model of *ontology axioms* ∪ *preconditions* ∪ *assumptions*
must be an elementary substructure of at least one model of *goal*[6]

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.

These are the two architectural constraints UPML imposes to guarantee well-defined task specifications. A third optional constraint ensures minimality of assumptions and preconditions and therefore maximizes the reusability of the task specification. It prevents overspecifity of assumptions and preconditions. Otherwise they would disallow to apply 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 described in [Fensel & Schönegge, 1998]). A large number of further constraints are described in [Fensel et al., to appear (a)].

6. 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*.

4 The relations between OIL and UPML

OIL is designed for defining ontologies, i.e., static information sources. UPML is designed for describing dynamic information sources. The Web blows the differences between these two types of information sources. Originally, web pages were static objects. Pages may be active and created as a result of a user query. Many software agent communicate with human users during their web browsing. Therefore it is quite natural to compare languages developed for static and dynamic information sources. In the introduction we identified four meaningful ways of relating OIL and UPML.

- OIL can be used as a meta-language to define UPML.
- OIL can be used as a language for writing down UPML specifications.
- OIL can be used as an object language for UPML.
- UPML be used as a language for writing down ontologies in OIL.

4.1 OIL as a meta-language for UPML

The Meta Object Facility (MOF) standard is a proposal of the OMG's group for expressing various modeling frameworks in a joined representation. Expressing the various modeling frameworks in a joined language (where the various modeling primitives are concepts and relations of the same “meta”-language) facilitates information exchange and reuse of software specifications expressed within different modeling frameworks. Therefore, in this section we will examine how useful OIL is for such a purpose taking UPML as an example. That is, we take OIL as the “meta” language and examine how well a modeling framework like UPML can be expressed in it.

[Fensel et al., to appear (a)] developed a *meta* ontology of UPML used to define its modeling constructs. This ontology starts with concepts, binary relationships, and restricted binary relationships. All three entities may have attributes (Figure 4 shows some of its parts). The main concept of UPML that are defined with this basic ontology are *Library*, *Ontology*, *Domain Model*, *PSM*, and *Task*. Besides *uses*, all attributes model *part-of* relationship. Sub concepts (*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*. Restricted Binary Relations connect two components of the same types. The root restricted binary relation of UPML is *Refiner*.

Concept and *Binary* **Relation** have not to be modeled explicitly in OIL because they correspond to the two main language primitives in OIL: classes and slots. OIL does not provide a generic element entity that would reify both. Also OIL fails to express **Restricted Binary Relation** because of its lacking meta-language features. We can model a specific slot in OIL that has the same specific concept as domain and range restriction. But we cannot express in a generic way a slot that must have the same concept as domain and range restriction without specifying an actual class (i.e., we cannot *parameterize* this definition because we do not have variables for class names).

Most of the components of UPML can be straight-forwardly modeled in OIL. Some examples are

Entity
attribute → type
Concept < Entity
Binary Relation < Entity
argument$_1$ → Concept$_1$
argument$_2$ → Concept$_2$
Restricted Binary Relation < Binary Relation
in = argument$_1$ → Concept$_1$
out = argument$_2$ → Concept$_2$
with Concept$_1$ = Concept$_2$
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
domain refiner → Domain 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
Primitive PSM < PSM
knowledge roles→ Role
assumptions → Formula
Complex PSM < PSM
subtasks → Task
operational description → Operational Description

Bridge < 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 < Bridge
argument$_1$ → PSM
argument$_2$ → Task
uses → PSM-Task Bridge
Task-Domain Bridge < Bridge
argument$_1$ → Task
argument$_2$ → Domain
uses → Task-Domain Bridge
Refiner < Restricted Binary Relation
pragmatics → Pragmatics
ontologies → Ontology
in → Concept
out → Concept
Domain Refiner < Refiner
...
Ontology Refiner < Refiner
in → Ontology
out → Ontology
signature → Signature
theorems → Formula
axioms → Formula
renaming → Renaming
Task Refiner[< Refiner
in → Task
out → Task
goal → Formula
input roles → Role
output roles → Role
precondition → Formula
assumptions → Formula
axioms → Formula
renaming → Renaming
PSM Refiner < Refiner
...
CPSM Refiner < PSM Refiner
...
PPSM Refiner < PSM Refiner
...
Pragmatics < Concept
explanation → STRING
author→ STRING
last date of modification → STRING
reference → STRING
URL → STRING
where & when be used → STRING
evaluation → STRING

Fig. 4 Part of the Meta-ontology of UPML.

```
class-def Library
    slot-constraint pragmatics
        value-type Pragmatics
    slot-constraint ontology
        value-type Ontology
    slot-constraint domain model
        value-type Domain Model
    slot-constraint complex PSM
        value-type Complex PSM
    slot-constraint primitive PSM
        value-type Primitive PSM
    slot-constraint task
        value-type Task
    slot-constraint ontology refiner
        value-type Ontology Refiner
    slot-constraint cpsm refiner
        value-type CPSM Refiner
    slot-constraint ppsm refiner
        value-type PPSM Refiner
    slot-constraint task refiner
        value-type Task Refiner Refiner
    slot-constraint psm-domain bridge
        value-type PSM-Domain Bridge
    slot-constraint psm-task bridge
        value-type PSM-Task Bridge
    slot-constraint task-domain bridge
        value-type Task-Domain Bridge

class-def Ontology
    slot-constraint uses
        value-type Ontology
    slot-constraint pragmatics
        value-type Pragmatics
    slot-constraint signature
        value-type Signature
    slot-constraint theorems
        value-type Formula
    slot-constraint axioms
        value-type Formula

class-def Pragmatics
    slot-constraint explanation
        value-type STRING
    slot-constraint author→
        value-type STRING
    slot-constraint last date of modification
        value-type STRING
    slot-constraint reference
        value-type STRING
    slot-constraint URL
        value-type STRING
    slot-constraint where & when be used
        value-type STRING
    slot-constraint evaluation
        value-type STRING
```

Fig. 5 Parts of the meta-ontology of UPML in OIL.

provided in Figure 5. However, it also makes an additional shortcoming of OIL obvious. The classes *Pragmatics* and *Ontology* refer to classes like *Formula* and *String*. OIL does not provide any axiomatic language and even in the case it will provide such a language it will not be accessible via a class definition. That is, the definition of formulas is provided in the definition of the language and cannot be accessed explicitly as a class. The class *String* points to another shortcoming of OIL. At the moment, OIL does not support concrete domains (e.g., integers, strings, etc.). However, this may change in the near future (cf. [Horrocks et al., to appear]) using the Datatype definitions of the XML schema language (cf. [Biron & Malhotra, 1999]) as a pattern for extending OIL accordingly.

The situation gets even worse when trying to model bridges and refiners with OIL. Binary relations (i.e., slots) do not have attributes in OIL. Therefore, OIL fails completely for modeling the adapter components of UPML.

Finally, an important aspect of the meta model of UPML are the constraints that ensure well-defined components and well-defined combination of components. However, none of these constraints can be expressed in OIL.

In consequence one has to encounter that OIL provides very restricted modeling primitives that fail in many aspects as a meta language for expressing the modeling primitives of UPML. That is, OIL cannot be used to express the ontology that describe the specification elements of reasoning components. If OIL should be of any use for ontology interchange it must provide powerful language elements for expressing these ontologies. *Spoken in a nutshell, OIL must be at least as expressive enough to express OIL, an ontology for ontology specification.*

4.2 OIL as a language for UPML

In many respects, OIL fails as a meta language for UPML. Lets see whether it provides more usability for directly expressing UPML specifications with it. At this level, a component specification of an ontology or a task (see Figure 2 and Figure 3) corresponds to an ontology in OIL. We tried to model a task ontology and a task specification in OIL. The OIL model of the task ontology is provided in Figure 6. The following observations have been made:

- The ontology container of OIL provides an excellent and standardized way to provide meta data of an ontology. The pragmatics slot of UPML looks rather ad hoc and one should expect that UPML will incorporate DublinCore Meta data in its next version, too.
- OIL can only express the first but not the second axiom of the ontology. The first axiom is expressed via subclass relationships and the second axiom is written down in the rule base that has currently no semantics.
- OIL does not provide the means to specify functional slots. We did this by defining a cardinality constraint but this is not yet part of the language definition for slots.
- Finally and most serious, the ontology speaks about sets of sets. An instance of *findings* is a set of instances of the class *finding* (and an instance of *hypotheses* is a set of instances of *hypothesis*). Therefore, we included a powerset operator in our specification but it is not part of the language definition and it may cause serious problems for its semantic. However, without this operator we

```
ontology-container
    title diagnoses
    creator Dieter Fensel
    subject
        The task ontology defines diagnoses for a
        set of observations.
    description.release 1.01.
    publisher
        D. Fensel: Understanding, Developing and
        Reusing Problem-Solving Methods.
        Habilitation, Faculty of Economic Science,
        University of Karlsruhe, 1998
    date May 2, 1998.
    type ontology
    format text/pdf.
    language OIL
    language UPML

ontology-definitions
    rule-base
        A hypothesis is parsimonious iff there is no
        smaller hypothesis with larger or equal
        explanatory power.
        parsimonious(H) ↔
        ¬∃H' (H' < H ∧ explain(H) ⊆ explain(H'))

slot-def <
    domain Hypotheses
    range Hypotheses
slot-def complete
    subslot-of explain
    domain Hypotheses
    range Findings
slot-def explain
    subslot-of complete
    domain Hypotheses
    range Findings
    cardinality 1
class-def Finding
class-def Hypothesis
    subclass-of NOT Finding
class-def Hypotheses
    subclass-of POWERSET Hypothesis
class-def Findings
    subclass-of POWERSET Finding
class-def parsimonious
    subclass-of Hypotheses
class-def observations
    subclass-of Findings
class-def diagnosis
    subclass-of Hypotheses
```

Fig. 6 A task ontology specified with OIL.

ontology-container
- **title** complete and parsimonious diagnoses
- **creator** Dieter Fensel
- **subject**
  - The task asks for a complete and minimal diagnoses
- **description.release** 1.01.
- **publisher**
  - D. Fensel: Understanding, Developing and Reusing Problem-Solving Methods. Habilitation, Faculty of Economic Science, University of Karlsruhe, 1998
- **date** May 2, 1998.
- **type** task
- **format** `text/pdf`.
- **language** `OIL`
- **language** `UPML`
- **relation**
  - **hasPart** It uses the ontology diagnosis

ontology-definitions
- **import** Ontology diagnosis
- **rule-base**
  - If we receive input there must be a complete and finite hypothesis.
  - $observations \neq \emptyset \rightarrow$
    - $\exists H . complete(H, observations) \wedge Finite(H)$
- **slot-def** <
  - **properties** *non-reflexive*, transitive
- **slot-def** task
  - **subslot-of** complete
  - **domain** observations
  - **range** parsimonious
- **class-def** observations
  - **documentation** input role
  - ***cardinality* >0**
- **class-def** diagnosis
  - **documentation** output role

Fig. 7 A task specified with OIL.

fail to capture the essence of this small and simple ontology.

Besides applying OIL directly to the ontology component of OIL we also tried to use it to model a task specification (still one of the most simplest components of UPML). The result is provided in Figure 7. We encounter similar problems as already reported:

- An important axiom cannot directly be expressed.
- We extend OIL with a property non-reflexive and cardinality constraints for classes.

A problem when using OIL at this level is that the structure of the specification units of UPML gets lost. We keep things like what is an input role or an output role only as natural language comment in the **documentation** slot. We will discuss the mismatch of architectural structures of OIL and UPML in the following subsections.

4.3 OIL is an object language for UPML

Using OIL as a logical language to define semantics for the elementary slots of UPML was the first way we thought to combine OIL and UPML. However, there are two problems with this approach. First, OIL is already more than just a logical language. It already comes along with an architecture comparable to a refined version of the ontology component in UPML (see Figure 8). Therefore, it does not make much sense to provide an architectural specification of each elementary slot of UPML. Second, OIL does not provide adequate expressive power for many of the axiomatic parts of UPML specifications. The first problem indicates that OIL is more appropriate at the level discussed in Section 4.2. It will require some work to synchronize the slightly different component models of OIL and UPML but then it should be possible to express a component of UPML as an ontology in OIL. The second problem is more principal. OIL fails at any level (i.e., as a meta-language, language, and object-language) to express important aspects of UPML. Extending the expressive power of OIL seems absolutely necessary for making it usable in this context.

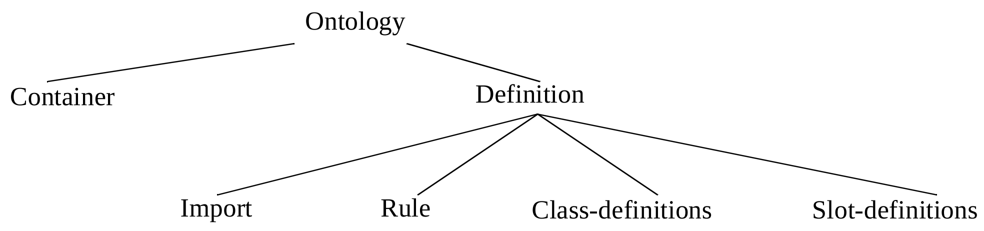

Fig. 8 The architecture of OIL.

4.4 UPML as a language (i.e., architecture) for OIL

Up to now we have asked what OIL can do for UPML. Now we will deal with the reverse question: Can UPML provide any help to OIL? Yes it can! OIL provides a very simple construction to modularize ontologies. In fact, this mechanism is identical to the namespace mechanism in XML. It amounts to a textual include of the imported module, where name-clashes are avoided by prefixing every imported symbol with a unique prefix indicating its original location. However, much more elaborated mechanism are required for a structured representation of large ontologies. Renaming, restructuring, and redefinition means must be applicable to imported ontologies. Here, we can make use of the adapter concept of UPML. UPML provides refiners and bridges to modify components. These adapter components of UPML can be used to integrate the need of ontology structuring in an existing architecture. When combining UPML and OIL in this way we are also able to specialize the generic adapter concept of UPML for the fixed set of language primitives of OIL like [Gennari et al., 1994], [Park et al., 1997] did for the fixed set of language primitives of Protégé [Grosso et al., 1999] (i.e., OKBC [Chaudhri et al., 1998]).

5 Conclusions

In this paper we try to relate two standardization efforts:

- OIL provides a standard language for expressing and interchanging ontologies, i.e., static information sources.
- UPML provides a standard language for specifying and reusing problem-solving methods, i.e., dynamic information sources.

Currently, the web blows the distinction between static and dynamic information sources. There is a continuum of static pages, dynamic generated pages, query-answering services and complex software services on it. Therefore it looks quite reasonable to try to bring these languages together forming a coherent framework for describing services on the WWW. In principle this can also be done in a fruitful way for both approaches because they focus currently at different levels. On the one side, OIL provides a specification language with well defined semantics and efficient reasoning support. The overall architecture of OIL specifications is rather simple–not going beyond an import statement. On the other side, UPML provides a full-fledged architecture for describing various aspects of a reasoning service. However, no formal language has yet been defined for it. Therefore OIL and UPML fit nicely together compensating the weaknesses of each other. However, in order to make this really true, the language OIL needs to provide more expressive power. Currently we fail for any example of a UPML specification to express its main aspects in OIL. In a nutshell, Description Logics seems too restricted for the purpose of functional specification of software components.

The message of the paper in a nutshell is: *Synchronizing the architectures of OIL and UPML* and *extending the expressive power of OIL* are two urgent things to do quickly. Providing a unified language for content and reasoning description is an essential step in the direction of a knowledeable web where the difference between both aspects should be transparent for the user.

References

[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.

[Biron & Malhotra, 1999] P. V. Biron and A. Malhotra: XML schema part 2: Datatypes. http://www.w3.org/TR/1999/WD-xmlschema-2-19991217/, 1999. W3C Working draft.

[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.

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

[Fensel, to appear (b)] D. Fensel. *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce*. Springer-Verlag, to appear. http://www.cs.vu.nl/~dieter/ftp/spool/silverbullet.pdf.

[Fensel & Benjamins, 1998] 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 & 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 et al., 1998] 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., 1999] 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., to appear (a)] D. Fensel, E. Motta, V. R. Benjamins, M. Crubezy, S. Decker, M. Gaspari, R. Groenboom, W. Grosso, F. van Harmelen, M. Musen, E. Plaza, G. Schreiber, R. Studer, and B. Wielinga, The Unified Problem-solving Method Development Language UPML, to appear. http://www.cs.vu.nl/~dieter/ftp/spool/upml.journal.pdf.

[Fensel et al., to appear (b)] D. Fensel, I. Horrocks, F. Van Harmelen, S. Decker, M. Erdmann, and M. Klein:

OILin a nutshell, to appear. http://www.ontoknowledge.com/oil.

[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).* http://www.cs.vu.nl/~dieter/ftp/spool/enrico.pdf.

[Gennari et al., 1994] J. H. Gennari, S. W. Tu, T. E. Rothenfluh, and M. A. Musen: Mapping Domains to Methods in Support of Reuse, *International Journal of Human-Computer Studies (IJHCS)*, 41:399–424, 1994.

[Gomez Perez & Benjamins, 1999] A. Gomez Perez and V. R. Benjamins: Applications of ontologies and problem-solving methods. *AI-Magazin*e, 20(1):119–122, 1999.

[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.

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

[van Heijst et al., 1997] G. van Heijst, A. Th. Schreiber, and B. J. Wielinga: Using explicit ontologies in KBS development. *International Journal of Human-Computer Studie*s, 46(2/3):183–292, 1997.

[Horrocks et al., to appear] I. Horrocks, D. Fensel, J. Broekstra, S. Decker, M. Erdmann, C. Goble, F. Van Harmelen, M. Klein, S. Staab, and R. Studer: OIL: The Ontology Inference Layer, to appear. http://www.ontoknowledge.com/oil.

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

[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.

[OMG, 1997] Object Management Group (OMG): Meta Object Facility (MOF) Specification, 1997.

[Park et al., 1997] J. Y. Park, J. H. Gennari, and M. A. Musen: Mappings for Reuse in Knowledge-based Systems. SMI Technical Report 97-0697, 1997.

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

[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.

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

[Uschold & Grüninger, 1996] M. Uschold and M. Grüninger: Ontologies: Principles, methods and applications. *Knowledge Engineering Revie*w, 11(2), 1996.