Automatic Creation of Interface Specifications from Ontologies

Iryna Gurevych[†] **Stefan Merten**[‡] **Robert Porzel**[†]

†European Media Lab GmbH
Schloss-Wolfsbrunnenweg 31c
D-69118 Heidelberg, Germany
{gurevych,porzel}@eml.org

‡DFKI GmbH
Erwin-Schrödinger-Str.57
D-67608 Kaiserslautern, Germany
merten@dfki.de

Abstract

The paper presents a system architecture for the automatic generation of interface specifications from ontologies.[1] The ensuing interfaces (XML schema definitions) preserve a significant amount of the knowledge originally encoded in the ontology. The approach is relevant for the engineering of large-scale language technology systems. It has been successfully deployed in a complex multi-modal dialogue system SMARTKOM.

1 Introduction

Interface specifications are an important part of the computational infrastructure in engineering language technology (LT) systems. This is a challenging task, especially for large-scale and multi-domain systems with numerous processing modules. To a great extent, the successful operation of such a system depends on the high-quality domain representations exchanged between individual modules.

Ontologies have traditionally been used to represent domain knowledge and are employed for various linguistic tasks, e. g., semantic interpretation, anaphora, or metonymy resolution. We propose an additional way of employing ontologies, i. e., to use the knowledge modeled therein as the basis for defining the semantics and the content of the information exchanged between the modules of LT systems.

In language technology systems, modules typically exchange messages, e. g., a parser of a dialogue system might get word lattices as input and produce corresponding semantic representations for later processing modules, such as a discourse manager. The increasing employment of XML-based interfaces for agent-based or other multi-blackboard communication systems sets a *de facto* standard for syntax and expressive capabilities of the information that is exchanged amongst modules. The content and structure of the information to be represented is typically defined in corresponding XML schemata (XMLS) or Document Type Definitions (DTD).

Employing the approach introduced below, XMLS and DTDs are created such that they:

- stay logically consistent, as the logical consistency of ontologies can be checked with the help of available tools,
- are easy to manage,
- enable a straight-forward mapping back to the respective knowledge representation for inference,
- allow the handling of language processing tasks immediately on the basis of XMLS.

This paper is organized as follows. In Section 2 we will give an overview of the approaches to knowledge storage as employed in LT systems. Section 3 outlines the task that we address, highlights the problems which arise when mapping knowledge structures from ontologies to XMLS and how these have been solved in our implementation. Section 4 discusses how the approach has been deployed in a real-life dialogue system as well as some of the additional advantages resulting from its application. Related approaches are, then, discussed in Section 5. Concluding remarks are found in Section 6.

2 Approaches to Knowledge Storage

Efforts originating in various W3C and Semantic Web projects brought about several knowledge modeling standards: Resource Description Framework (RDF), DARPA Agent Mark-up Language (DAML), Ontology Interchange Language (OIL), Web Ontology Language (OWL).[2] As for intra-agent or intra-module communication languages, either XMLS or DTDs have become standards for interface specifications, due to the fact that instance documents can be automatically validated during run-time and software has been developed for parsing and marshaling information represented in these formats.[3]

[1] The resulting software is licensed under the GPL and is a free software project. The package and respective documentation can be obtained from http://savannah.nongnu.org/projects/oil2xsd.

[2] See www.w3c.org/RDF, www.ontoknowledge.org/oil, www.daml.org, and www.w3.org/2001/sw/WebOnt for the individual specifications.

[3] See Xerces (xml.apache.org) for parsing and Castor (castor.exolabs.org) for marshaling XML documents.

Current systems often feature both XMLS- or DTD-based communication languages as well as DAML- or RDF-based knowledge stores. However, those are often structurally and terminologically heterogeneous. Mappings from the message content to the ontology are often difficult and costly. Attempts to hand-craft XMLS or DTDs for defining communication between various processing modules show that several problems grow roughly linear to the complexity of the domains to be defined by means of the individual representations. These problems are:

- inconsistencies in modeling choices, e.g. elements versus attributes,
- inconsistencies in the hierarchy, e.g. flat-ness versus depth of individual branches,
- readability and understandability of the schemata.

In a system involving multiple domains it becomes pretty much impossible to manually define suitable XML schemata for the modules that exchange information about the multitude of possible utterances. The ensuing inadequacies of the representations constitute a substantial obstacle for system development and its functionality. Processing modules operating on such schemata can also not apply inferencing algorithms directly on these structures, as they do not represent enough knowledge. This has the effect that individual knowledge stores have to be hand-crafted for specific components, causing the heterogeneity between the communicated and the modeled objects to increase further. Additionally, readability decreases as more complex XMLS structures, such as extension hierarchies or substitution groups, are used and potential links to *semantic web* ontologies are lost or become costly.

3 The Task: From Knowledge to Interfaces

Ontologies are a suitable means for knowledge representation, i.e. for the definition of an explicit and detailed model of the system's domains. That way, they provide a shared domain theory, which can be used for communication. Additionally, they can be employed for deductive reasoning and manipulations of models. The meaning of ontology constructs relies on a translation to some logic. This way, the inference implications of statements, e.g. whether a class can be related to another class via a subclass or some other relation, can be determined from the formal specification of the semantics of the ontology language. However, this does not make any claims about the syntactic appearance of the representations exchanged, e.g. an ordering of the properties of a class.

An interface specification framework, such as XMLS or DTD, constitutes a suitable means for defining constraints on the syntax and structure of XML documents. Ideally, the definition of the content communicated between the components of a language technology system should relate both the syntax and the semantics of the XML documents exchanged. Those can then be seen as instances of the ontology represented as XMLS-based XML documents. However, this requires that the knowledge, originally encoded in the ontology, is represented in the XMLS syntax.

3.1 Ontology to XMLS transformation

The solution presented here states that the knowledge representations to be expressed in XMLS are first modeled in OIL-RDFS or DAML+OIL as an *ontology proper*, using the advantages of ontology engineering systems available, and then transformed into a communication interface automatically with the help of the software developed for that purpose. Before showing how the problems mentioned in Section 2 can be minimized, we will introduce the basic formal properties of the given source and target representations.

Ontology representation languages: Domain knowledge stored in the ontology may be encoded using XML-based semantic mark-up languages, such as OIL, or DAML+OIL. In the work reported here, we used an ontology defined in the OIL-RDFS syntax, but the basic transformation algorithms are as well applicable to OIL or DAML+OIL .

OIL-RDFS is a representation format which allows to express any OIL ontology in RDF syntax. This has the advantage that the ontology is partially understandable for non-OIL aware RDFS applications. Additionally it allows for all the formal semantics and reasoning support available for OIL. A detailed characterization of the formal properties of the OIL language can be found in Fensel et al. (2001).

The semantics of OIL is based on a combination of frame and description logic extended with concrete datatypes. The FACT system[4] can be used as a reasoning engine for OIL ontologies, providing some automated reasoning capabilities, such as class consistency or subsumption checking. The OIL language employs frame semantics and provides most of the modeling primitives commonly used in frame-based knowledge representation systems. Graphical ontology engineering front-ends and visualization tools are available for editing, maintaining, and visualizing the ontology.[5]

XML Schema: XML schemata provide a grammar for prescribing the structure of XML documents, data typing

[4] See also www.cs.man.ac.uk/ horroks/FaCT/.

[5] See OilEd (oiled.man.ac.uk) and for editing and FrodoRDFSViz (www.dfki.uni-kl.de/frodo/RDFSViz) for visualization.

as well as inclusion and derivation mechanisms. XMLS definitions are themselves XML documents, which have the immediate advantage that all tools developed for XML, e.g. validation tools, can be immediately used for XMLS.[6]

3.2 Differences between ontology languages and XMLS

In comparing ontology languages and XML schema it is important to realize, that while both of them provide vocabulary and structure to represent knowledge, the underlying formalisms show different formal properties since their purpose is different.

OIL-RDFS definition is a directed acyclic graph, while XMLS establish a tree structure. Ontology languages, such as OIL provide much richer modeling primitives, i.e. classes, slots. They also incorporate the notion of (multiple) inheritance, which may be either explicitly stated or implied.

XMLS have different modeling primitives, i.e. elements of certain types, which can be either simple or complex types. However, no precise semantic interpretation is assigned to them. There is no inheritance as such, but types can be derived by extension or restriction, i.e. types can share some structures between them. Generally the XMLS language is much richer in terms of its datatyping capabilities and grammar for prescribing the structure and content of the elements.

In contrast to that, ontologies constitute high level domain models. Because of different formal properties of the underlying representation formalisms, a straightforward mapping between them is not always possible. In some cases it may be rather intricate, so that special transformation algorithms are required. These algorithms are responsible for explicating and mapping knowledge structures from the ontology to XMLS.

3.3 Transforming OIL definitions into XMLS definitions

In this section, we provide a description of the algorithms employed by the transformation software. We will show how OIL definitions can be written in XMLS. Let's assume the existence of the ontology shown in Figure 1.

Step 1 Mapping of class definitions: According to this ontology, the class *WatchPerceptualProcess* is a subclass of *PerceptualProcess*, and its instances have an object to be watched, i.e. *AvEntertainment*. A corresponding OIL definition looks as follows:

[6]The specification of the language is given in www.w3c.org/XML/Schema.

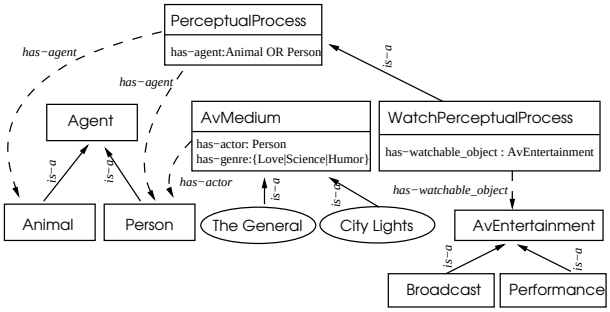

Figure 1: A sample ontology

```
class-def WatchPerceptualProcess
  subclass-of PerceptualProcess
  slot-constraint
   has-watchable_object
   has-value AvEntertainment
```

The translation of the class definition header is done in a straight-forward manner using the XML schema complexType construct to assign a name:

```
  <complexType name="WatchPerceptualProcess>
```

Step 2 Resolving inheritance: The notion of inheritance is important for many language processing algorithms operating either on the basis of an ontology or equivalent XMLS. It allows, for example, underspecification in semantic representations, when a more general class is used in place where specific derived classes can also occur. The generalization hierarchy found in the ontology should be made explicit to an equivalent type extension structure in the XMLS.

The subclass-of statement would thus be translated to XML schema. This corresponds to the type extension[7] and results in a construct:

```
<complexType name="WatchPerceptualProcess>
  <complexContent>
    <extension base="ns:PerceptualProcess"/ >
  </complexContent>
</complexType>
```

However, in more complicated cases, in particular when the class in question has subclasses and consequently shares some structures (slots) with its superclass, as in the example below, a direct mapping to the type extension in XML schema may result in a problem.

[7]As XML schema does not allow to express multiple inheritance, only unary inheritance can be handled. This is a constraint on the modeling side which should be taken into account. Also *ns:* in our examples stands for any namespace, which can be assigned freely.

```
class-def Agent
class-def Animal
  subclass-of Agent
class-def Person
  subclass-of Agent
```

The instance documents would become ambiguous, if some non-abstract types (i.e. types that can be instantiated, e.g. *Agent*) have derived types (e.g. *Person*). As some substructures are shared by the types, in the instance documents it is no longer possible to determine whether a specific element matches the content model of *Agent*, *Person* or possibly another type derived from *Agent* by extension, e.g. *Animal*.

To avoid this sort of ambiguity, a specific algorithm is proposed. The basic idea is to move the information needed to identify the type of the element to additional artificially created complex types.[8] This way, each class in the ontology is translated to a set of three complex types.

Type_abstract: Type_abstract serves to preserve the original class definition in the ontology, but is given an attribute *abstract* in the XML schema, i.e. it may never be instantiated, resulting in:

```
<complexType name="Agent_abstract"
  abstract="true"/>
```

Type_final: Type_final is a direct derivation of the Type_abstract adding no new content, but it is final, i.e. no more type may be derived from it. Type_final is used when elements are given a type. There is no more ambiguity in the instance documents, as Type_final does not have any derived types, e.g.:

```
<complexType name="Agent_final">
  <complexContent>
    <extension base="ns:Agent_abstract"/>
  </complexContent>
</complexType>
```

Type: Type can be used by the elements because it is not derived from anything. It consists of a choice XMLS construct containing an element for each of the possible derivations of the Type_abstract including Type_abstract itself. As a result, any element tag of the choice, which is a derivation of the given base type can be actually used in the instance document, as follows:

[8] An alternative solution is to use xsi:type attributes in the instance documents. This, however, ceases to make inheritance structures visible through the element tags.

```
<complexType name="Agent">
  <choice>
    <element name="agent"
     type="ns:Agent_final"/>
    <element name="animal"
     type="ns:Animal_final"
     minOccurs="0"/>
    <element name="person"
     type="ns:Person_final"
     minOccurs="0"/>
  </choice>
</complexType>
```

Step 3 Mapping of slot constraints: Class descriptions in the ontology may typically contain one or more slot constraints of certain kinds. For each slot constraint associated with a class a sequence element of the corresponding complex type in XMLS is created, like in the example below:

```
class-def WatchPerceptualProcess
  subclass-of PerceptualProcess
  slot-constraint
    has-watchable_object
    has-value AvEntertainment
```

```
<complexType
  name="WatchPerceptualProcess_abstract"
  abstract="true">
  <complexContent>
    <extension
     base="ns:PerceptualProcess_abstract">
     <sequence>
       <element name="watchableObject"
        type="ns:AvEntertainment"
        minOccurs="0"/>
     </sequence>
    </extension>
  </complexContent>
</complexType>
```

The slot constraint *has-watchable_object* of *WatchPerceptualProcess* is mapped to an element *watchableObject*. The cardinality of the slot is transformed to the cardinality of the element accordingly. The type of the element in a sequence is determined from the appropriate slot filler in the ontology. There exist a variety of possibilities to specify the domain of slots in the ontology (see Step 4 for a detailed discussion). As each of them requires special treatment while translating to XMLS, we discuss this in a separate step.

Step 4 Resolving fillers of slot constraints: The domain of the slot in the ontology can be specified by class expressions of different complexity, individual or a set of

those and some elementary data types. However, XMLS require that an element representing a slot in the corresponding complex type definition is always given a single non-ambiguous type. This may be the case when a slot is filled with a single class having no subclasses; in other cases it is not directly expressible in XMLS and a special mechanism is required. We support the transformation of the following slot fillers:

A non-final class definition: A non-final class in the ontology is a class having further subclasses in the generalization hierarchy. The semantics of such a class in the slot definition is that any instance of this class or any of its subclasses can fill the slot. As XMLS lack the notion of implicit semantics, the subclass hierarchy of the respective non-final class must be made explicit. The complex type *Type* containing a choice running across all possible derivations is employed as an unambiguous type of the sequence element within the corresponding complex type definition (cf. Step 3).

```
<complexType name="AvEntertainment">
  <choice>
   <element name="avEntertainment"
    type="ns:AvEntertainment_final"/>
   <element name="broadcast"
    type="ns:Broadcast_final"
    minOccurs="0"/>
   <element name="performance"
    type="ns:Performance_final"
    minOccurs="0"/>
  </choice>
</complexType>
```

Boolean class expressions with OR: In the ontology, slots can be filled with a boolean expression of classes.

```
class-def PerceptualProcess
  slot-constraint
   has-agent has-value Animal or Person
```

The semantics of such an expression as slot-filler is that any instance of the evaluated expression can fill the slot. As XMLS does not support the use of logical operators, placeholder complex types, e.g. *Or_PerceptualProcess_agent* in the example below, should be introduced at this point to XMLS. These artificially created complex types consist of a choice of elements corresponding to each of the operands of the boolean class expression. The elements are given a type of the respective class.

```
<complexType name="PerceptualProcess_abstract"
  abstract="true">
 <sequence>
  <element name="agent"
   type="ns:Or_PerceptualProcess_agent"
   minOccurs="0"/>
 </sequence>
</complexType>
<complexType name="Or_PerceptualProcess_agent">
  <choice>
   <element name="animal"
    type="ns:Animal_final"/>
   <element name="person"
    type="ns:Person_final"/>
  </choice>
</complexType>
```

Similar algorithms exist for resolving AND and NOT operators. The only difference for AND is that elements of a placeholder complex type would not be combined in a choice, but in a sequence. For NOT, the choice of the complex type would contain elements for all classes, except the class which is the operand in the respective expression.

Individuals: In the following ontology definition, the slot-constraint *has-genre* is filled with a set of individual values *love*, *humor*, *science*.

```
class-def AvMedium
  slot-constraint has-genre has-value
  (one-of love humor science)
```

The slot *has-genre* is translated to a sequence element *genre* of the complex type corresponding to the *AvMedium* class definition. In order to give this element an unambiguous type, a placeholder simple type *OneOf_AvMedium_genre* is introduced. The content model of this type is then restricted to the set of enumeration values corresponding to the set of individuals specified in the original slot definition.

```
<complexType name="AvMedium_abstract"
  abstract="true">
 <sequence>
  <element name="genre"
    type="ns:OneOf_AvMedium_genre"
    minOccurs="0"/>
 </sequence>
</complexType>
<simpleType name="OneOf_AvMedium_genre">
 <restriction base="ns:NMTOKEN">
  <enumeration value="humor"/>
  <enumeration value="love"/>
  <enumeration value="science"/>
 </restriction>
</simpleType>
```

Step 5 Mapping of individuals: The last step to be taken in transforming the ontology to XMLS is mapping the individuals which represent instances of specific classes in the ontology, e.g. *The General* and *City Lights* are modeled as instances of the *AvMedium* class.

```
instance-of The General AvMedium
instance-of City Lights AvMedium
```

For instances of individual classes, a simple type with the name of the respective class is created. The content model of this simple type is an enumeration of specific values which correspond to the names of individuals in the ontology.

```
<simpleType name="AvMedium">
 <restriction base="ns:NMTOKEN">
  <enumeration value="The General"/>
  <enumeration value="City Lights"/>
 </restriction>
</simpleType>
```

4 Application in a Dialogue System

Our approach to the automatic creation of interface specifications from an ontology has been successfully tested in SMARTKOM, a complex multi-modal dialogue system (Wahlster et al., 2001). The system comprises a large set of input and output modalities which the most advanced current systems feature, together with an efficient fusion and fission pipeline. SMARTKOM supports speech input with prosodic analysis, gesture input via infrared camera, recognition of facial expressions and their emotional states. On the output side, the system features a gesturing and speaking life-like character together with displayed generated text and multimedia graphical output.

The system currently comprises nearly 50 modules running on a parallel virtual machine-based integration software called *Multiplatform* (Herzog et al., 2003). The modules exchange messages whose content is encoded in XML. The interfaces are defined by a set of XML schemata. The part of them containing the system's knowledge about application domains was obtained via the automatic transformation of an OIL-RDFS ontology (Gurevych et al., 2003b). Thusly, all components of the system operate on a common knowledge store - XML schemata resulting from the ontology transformation, e. g., the parser (Engel, 2002), the dialogue manager (Löckelt et al., 2002).

In this trial, our initial hypothesis that employing ontological knowledge for interface specifications will make them more consistent, better-structured and more readable as compared to manually defined interfaces was fully satisfied. Some additional advantages that were not anticipated originally also resulted from this approach.

Enhancing the OIL-RDFS datatyping capabilities: As previously stated, ontologies are a suitable means for specifying high-level domain knowledge. However, if knowledge represented in the ontology is to become part of the common XML schema based representation exchanged between the modules, it is important to have a mechanism for referencing structures, i. e., datatypes defined elsewhere in a larger XML context.

It should be noted that the datatyping capabilities of the OIL-RDFS ontology *per se* are very limited. Therefore, enabling references to XMLS datatypes within the ontology, or more generally, referencing any datatype defined elsewhere in a larger XML context, is in practice beneficial to the ontology.

We provide a special mechanism which allows to employ *external* datatypes in the ontology, e. g., NmToken (a built-in XML schema datatype) or derived datatypes, such as ns:TimeExpression (as parts of other ontologies, like a time ontology). To make use of this feature, external datatypes have to be modeled as instances in the ontology. Thus, they can be employed, e. g., as slot fillers. Consequently, the ontology converted to XML schema can be embedded in a larger XML context.

Supporting multiple applications with a single ontology: Ontology construction is known to be labor and cost intensive. To reduce the cost of ontology design and maintenance, it is necessary to construct ontologies which are re-usable, i. e., support multiple applications and domains. This may, however, often result in the side effect that an ontology covers more domains than are addressed by the specific system in question. Transforming the ontology to XMLS as is, would then lead to *overloading* the domain model and slowing down the system and development performance.

As a solution to this problem we enabled different (system-dependent) views on a single ontology covering multiple domains. This solution requires that certain

parts of the ontology are marked up as being relevant for a particular system in question. The mark-up is examined automatically and a decision is made which parts of the ontology are irrelevant for the specific system at hand. These parts are, then, skipped in the process of transformation to XMLS. As a result, the XML schema-based domain model contains exactly the knowledge relevant for a particular system and presents one of the possible views on the underlying ontology.

Language processing tasks on XMLS: Resulting from the transformation a maximal amount of the knowledge from the ontology has been preserved and made explicit in the produced XMLS. Various linguistic operations, e. g., anaphora, bridging and metonymy resolution or discourse processing techniques such as *overlay* (Alexandersson and Becker, 2003) can, therefore, work directly on the schemata.

5 Related Work

The relation between ontologies and schema-languages has been addressed previously in the AI and Semantic Web communities. Gil and Ratnakar (2002) carried out a detailed comparison of semantic mark-up languages in the course of looking for a language suitable for developing user-oriented tools for the Semantic Web.

Klein et al. (2000) relate ontologies to the XMLS language definition, applicable in the context of defining the content of on-line information sources. Their conclusion is that both refer to different levels of abstraction and should therefore be used at the different stages of the development of information sources. They also provide a translation procedure from OIL to XMLS which is similar to ours, yet differs in the technical details. In contrast to our work, Klein et al. (2000) do not verify their approach via practical implementation, i. e., while it is stated that most of the steps could be automated, the focus of their work remains on a fairly theoretical level. The approach proposed herein has been implemented and successfully deployed in a language technology system. It is available as a free software project, thus enabling its practical re-use in other systems.

Some research is also underway to explore the reverse direction, i.e. from XML schema to ontology content[9]. The motivation for that is twofold: firstly to enable reasoning about XML content for DAML-enabled software and secondly to create DAML content from XML in a quick and automated fashion. The main objective of our approach is, however, to bring semantics to XML documents, i.e., derive appropriate interface specifications from the given domain model, thereby enabling high-quality reasoning immediately on the XMLS level.

[9]For example, http://www.davincinetbook.com:8080/daml/xmltodaml/xmltodaml.html

6 Discussion

The system architecture presented herein eliminates the necessity of crafting interface specifications manually. Instead, the domain knowledge of LT systems is first modeled as an ontology and then transformed into XMLS automatically. The resulting schemata capture the hierarchical structure and a significant part of the semantics of the ontology. This provides a standard mechanism for the creation of XMLS-based interface specifications storing knowledge about the system's domains.

The ontology employed in our system has been designed as a general purpose component for knowledge-based language processing. It can be accessed automatically, [10] if additional reasoning operations are desired. E.g., it has been successfully applied to the task of scoring sets of concepts in terms of their semantic coherence (Gurevych et al., 2003a). Important to note is that this single ontology can be re-used not only for different tasks, but also for different systems as multiple views and creation of application-specific interface specifications are possible.

A further advantage of our approach is that it combines the power of ontological knowledge representation with the strengths of XMLS as an interface specification framework in a single and consistent knowledge store. Our experience suggests that this is not feasible for large-scale LT systems, if XML schema were defined from scratch or hand-crafted. Successfully deployed in a complex dialogue system, which is just a particular instance of its application, the system architecture constitutes a more general and practical approach to the automatic interface generation in language technology systems.

Acknowledgments

This work has been partially funded by the German Federal Ministry of Research and Technology (BMBF) as part of the SmartKom project under Grant 01 IL 905C/0 and by the Klaus Tschira Foundation. We would like to thank Michael Strube for his helpful comments on the previous versions of this paper.

References

Jan Alexandersson and Tilman Becker. 2003. The Formal Foundations Underlying Overlay. In *Proceedings of the Fifth International Workshop on Computational Semantics (IWCS-5)*, Tilburg, The Netherlands.

Ralf Engel. 2002. SPIN: Language understanding for spoken dialogue systems using a production system approach. In *Proceedings of ICSLP 2002*.

[10]See APIs such as JENA (www.hpl.hp.com/semweb/jena-top.html) and corresponding query languages, such as RDQL.

Dieter Fensel, Frank van Harmelen, Ian Horrocks, Deborah McGuinness, and Peter Patel-Schneider. 2001. OIL: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2).

Yolanda Gil and Varun Ratnakar. 2002. A comparison of (semantic) markup languages. In *Proc.of the 15th Int. FLAIRS Conference*, Florida.

Iryna Gurevych, Rainer Malaka, Robert Porzel, and Hans-Peter Zorn. 2003a. Semantic coherence scoring using an ontology. In *Proceedings of the HLT-NAACL Conference*, Edmonton, Canada.

Iryna Gurevych, Robert Porzel, Elena Slinko, Norbert Pfleger, Jan Alexandersson, and Stefan Merten. 2003b. Less is more: Using a single knowledge representation in dialogue systems. In *Proceedings of the HLT-NAACL'03 Workshop on Text Meaning*, Edmonton, Canada.

G. Herzog, H. Kirchmann, S. Merten, A. Ndiaye, and P. Poller. 2003. Multiplatform testbed: An integration platform for multimodal dialog systems. In *Proceedings of the HLT-NAACL'03 Workshop on the Software Engineering and Architecture of Language Technology Systems (SEALTS)*, Edmonton, Canada.

Michel Klein, Dieter Fensel, Frank van Harmelen, and Ian Horrocks. 2000. The relation between ontologies and schema-languages: translating OIL-specifications in XML-schema. In *Proc. of the Workshop on Application of Ontologies and Problem Solving Methods*, Berlin, Germany.

Markus Löckelt, Tilman Becker, Norbert Pfleger, and Jan Alexandersson. 2002. Making sense of partial. In *Proceedings of the sixth workshop on the semantics and pragmatics of dialogue (EDILOG 2002)*, pages 101–107, Edinburgh, UK.

Wolfgang Wahlster, Norbert Reithinger, and Anselm Blocher. 2001. SmartKom: Multimodal communication with a life-like character. In *Proceedings of the 7th European Conference on Speech Communication and Technology*, pages 1547–1550.