Metacompiling OWL Ontologies

Anders Nilsson[1] and Görel Hedin[2]

[1] Department of Automatic Control
Lund University, Sweden
[2] Department of Computer Science
Lund University, Sweden

Abstract. Ontologies, formal knowledge representation, and reasoning are technologies that have begun to gain substantial interest in recent years. We present a high-level declarative approach to writing application programs for specific ontologies, based on viewing the ontology as a domain-specific language.
Our approach is based on declarative meta-compilation techniques. We have implemented a tool using this approach that allows typed front-ends to be generated for specific ontologies, and to which the desired functionality can be added as separate aspects. Our tool makes use of the JastAdd meta-compilation system which is based on reference attribute grammars. We describe the architecture of our tool and evaluate the approach on applications in industrial robotics.

1 Introduction

The *semantic web* [23] aims at formalizing large portions of knowledge in a form which enhances interoperability of usually distributed systems, and which introduces provisions for a common understanding of basic terms. The term *ontology* is normally used in this context to denote a logical formalization of a particular domain of knowledge, stored in a commonly understood format and accessible via the world wide web or a similar mechanism.

There are already many tools for handling ontologies in different formats. Ontology editors for the well known ontology notation OWL [15, 16], for example Protégé [17] and OntoStudio [5], typically store the information in a knowledge database as RDF triplets (subject, predicate, and object).

Access and manipulation of such knowledge, can be done either at the *generic* level, i.e., in terms of the triplets, or at a *domain-specific level*, where the ontology is used for interpreting the knowledge as a domain-specific programming model. Examples of tools working at the generic level include semantic reasoners, such as FaCT++ [21] or Pellet [11], which can infer new facts from the knowledge base, and the standard query language SPARQL [19], which allows the knowledge base to be queried. The knowledge base can also be directly accessed programmatically, using a generic API, like the Jena Owl API [8].

While the generic level is appropriate for general reasoning over arbitrary ontologies, the domain-specific level is often more appropriate for applications tied to a specific ontology. For example, a hierarchical structure is easy and natural to represent as a tree at the domain-specific level, but needs to be represented as separate triplets of knowledge at the generic level, making generic information retrieval cumbersome and error-prone: Instead of simply traversing a tree structure, each level of nodes must be retrieved using new queries and the hierarchical structure must be maintained outside the knowledge database in the application logic.

There are many tools that can be used to support writing applications at the domain-specific level. These tools typically represent the knowledge as an object-oriented model, introducing classes for the concepts in the ontology, and interpreting predicates to model class and object relationships like inheritance and part-of relations. Typically, this domain-specific model is accessible through an API in an object-oriented programming language, like Java, and which is generated from the ontology schema. Examples of such tools include RDFReactor [22] and Owl2Java [24]. A related approach is that of providing mappings between RDF and object-oriented models, such as EMF, e.g., [7].

However, a plain generated API has its limitations. First, the application programmer might like to enrich the generated semantic model with application-specific computations, for example in the form of additional fields and methods. Second, such application code should be separated from the generated API: we do not want the application programmer to edit generated code. Third, for computing properties of the semantic model, it can be advantageous to use high-level declarative programming. Fourth, the ontology might change, and it is desirable that the application code can be reasonably robust to such changes.

In this paper, we provide a solution that supports these requirements. We note that the problem of writing the application program is similar to writing a compiler: we need to parse information, analyze it, and generate some kind of output as a result. By using an object-oriented language, we can map the triplets for a specific ontology dialect to a typed object-oriented abstract syntax tree that is easy to perform computations on. For example, a subclass triplet, like (PincerGripper, subclassof, Gripper), would be mapped to a subclass relation between the corresponding classes in the object-oriented language. And a composition restriction, like (Gripper, has, OpenSkill), would be mapped to a parent-child relation in the abstract syntax tree.

Implementing the classes for such an abstract syntax tree by hand would be awkward, however, since they will then be sensitive to future changes to the description specification. Even small changes to the structure could imply a lot of work to adjust the compiler to the changes.

As is not uncommon, such problems become easier to solve by moving up to the next abstraction level. By implementing a meta-compiler, a compiler for OWL that, as output, generates a compiler for the description language specified in OWL, the abstraction level is raised. Instead of having to handle the dependencies between description language and tools manually, there is now one single specification for both description language and tool generation. The fact that these description languages are XML-based helps in that the parsing syntax is given beforehand.

We have implemented such a meta-compiler for OWL, called *JastOwl*. JastOwl is implemented using the JastAdd meta-compilation system [3] which supports high-level declarative computations on the abstract syntax tree by means of reference attribute grammars [6], and aspect-oriented modularization using inter-type declarations [10].

The rest of this paper is structured as follows. In section 2 we describe the architecture of JastOwl. Section 3 gives an example application of using JastOwl in the area of industrial robotics, and section 4 evaluates the approach. Related work is discussed in section 5, and section 6 concludes the paper.

2 JastOwl, a Meta-Compiler for OWL

Figure 1 shows the use of the JastOwl meta-compiler. Given an OWL ontology and hand-written application aspects, a dedicated compiler is generated that can parse an OWL knowledge database following the constraints defined by the ontology, and process that information according to the application aspects. For example, the dedicated compiler could generate vendor-specific configuration files for a particular robot, or interface classes for particular sensors and actuators. The dedicated compiler could also be a more advanced interactive application, communicating with an active robot, for example, to employ a skill server database to reason about what tools to attach to a robot to accomplish a specific task. These are just a few examples of possible applications.

In the middle part of the figure we see the generation of the dedicated compiler: JastOwl parses the ontology and generates specifications for the dedicated compiler, namely a parsing grammar, an abstract grammar, and a JastAdd aspect that contains methods for serialization. The parsing grammar is run through a parser generator, JavaCC in our case [13], to produce the parser for the dedicated compiler. The abstract grammar and the generated JastAdd aspects are combined with hand-written application aspects and run through JastAdd to generate the remaining part of the dedicated compiler.

The JastOwl tool is itself generated using JavaCC and JastAdd, as shown in the top part of the figure. The architecture is general, and we could use the same architecture to generate similar tools for other ontology notations than OWL, and for other file formats (there exists a sister tool for XML).

The JastOwl tool analyzes the ontology to find class declarations and restrictions on individuals of these classes in order to generate the JastAdd abstract grammar, see Fig. 2. The abstract grammar corresponds to an object-oriented API with a type hierarchy and traversal methods for abstract syntax trees following the grammar. The generated JastAdd aspect adds OWL/XML serialization methods to this API. The handwritten application aspects use the combined API to generate the desired robotics code for an input knowledge database. Examples of such generated code could be interface classes, communication protocol code, and skill server reasoning code.

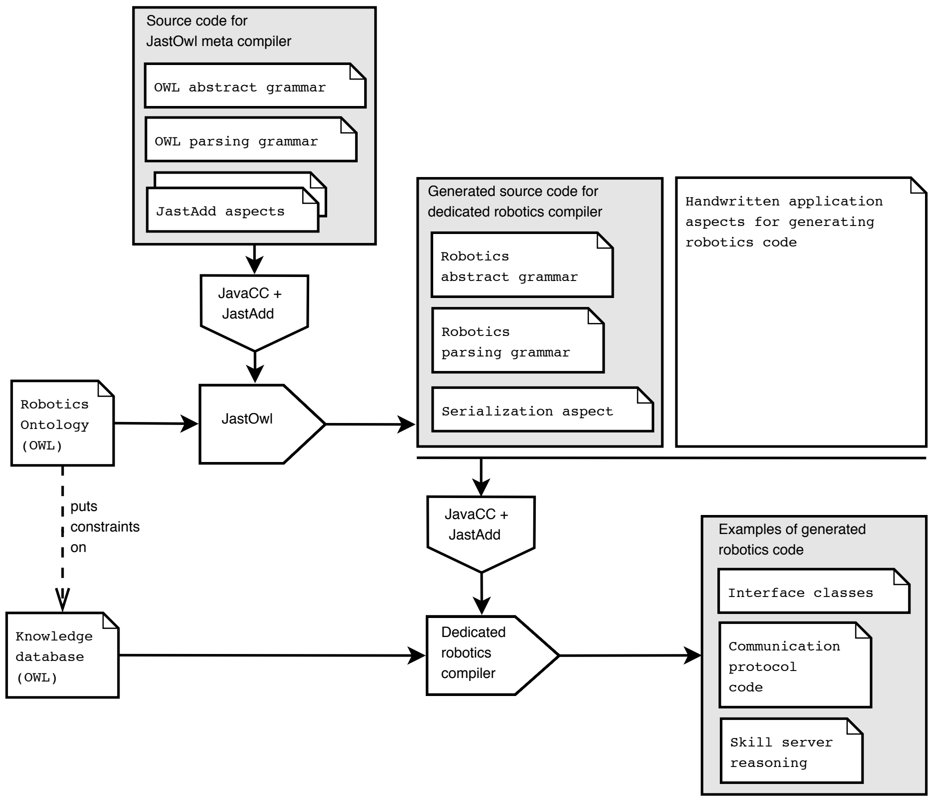

Source code for JastOwl meta compiler
OWL abstract grammar
OWL parsing grammar
JastAdd aspects
JavaCC + JastAdd
Robotics Ontology (OWL)
JastOwl
Generated source code for dedicated robotics compiler
Robotics abstract grammar
Robotics parsing grammar
Serialization aspect
Handwritten application aspects for generating robotics code
puts constraints on
JavaCC + JastAdd
Knowledge database (OWL)
Dedicated robotics compiler
Examples of generated robotics code
Interface classes
Communication protocol code
Skill server reasoning

Fig. 1. The JastOwl meta compiler. JastOwl generates a dedicated compiler description (abstract grammar, parsing grammar, and serialization code) for a given ontology. This description can be extended with application-specific handwritten aspects to generate the dedicated compiler. JastOwl is itself generated using JastAdd and JavaCC.

2.1 Generation Details and Limitations

The conceptual differences between Description Logic (DL) and Object Oriented (OO) systems as well as different ways of bridging the gap have been described in several papers. Kalyanpur et.al. [9] maps OWL classes to Java interfaces and properties to untyped Java lists while for example RDFReactor [22] uses a more elaborate approach where the OWL class hierarchy is being flattened to fit the Java single inheritance model. RDFReactor also handles OWL properties in a typed way in the generated Java code.

The development of JastOWL, on the other hand, has so far not been aimed towards a complete representation of DL in OO or a front-end to existing reasoners. Instead, the original idea was to implement a pragmatic toolkit in order to make it easier to write software that extracts knowledge from an ontological knowledge source and makes something out of it. For example, to generate access code or to generate communication protocol code.

The JastOWL translation of OWL concepts is similar to how RDFReactor does it, but with some limitations: The current version directly translates OWL classes into Java classes limiting us to ontologies where multiple inheritance is not used. OWL properties are handled similarly. For each property, child nodes will be generated for the corresponding domain class in the JastAdd abstract grammar, see Fig. 2. Multiple range properties are not yet supported.

It can be noted that the ontology and knowledge database are often stored in the same OWL file. Both the meta-compiler and the generated compiler will then operate on the same OWL file, but with very different goals. The meta-compiler looks for declarations of classes and restrictions, while the generated compiler is mainly interested in the instances of the aforementioned classes and restrictions.

To evaluate the approach, several prototype applications have been implemented, primarily in the area of industrial robotics.

3 SIARAS Skillserver Example

The example described here was developed as a part of the EU-project SIARAS *Skill-Based Inspection and Assembly for Reconfigurable Automation Systems* (FP6 - 017146) `http://www.siaras.org`. The main goal of the SIARAS project was to facilitate simple dynamic reconfiguration of complex production processes, by introducing the concepts of skill-based manufacturing and structured knowledge.

3.1 Ontology Structure

At the top level, see top left of Fig. 3, the ontology is split into six categories:

ObjectBase Every physical object can be modeled as a simple *Part*, or as an *Assembly* consisting of parts or other assemblies.

Operation The vocabulary needed for talking about operations[3] that are performed by a device.

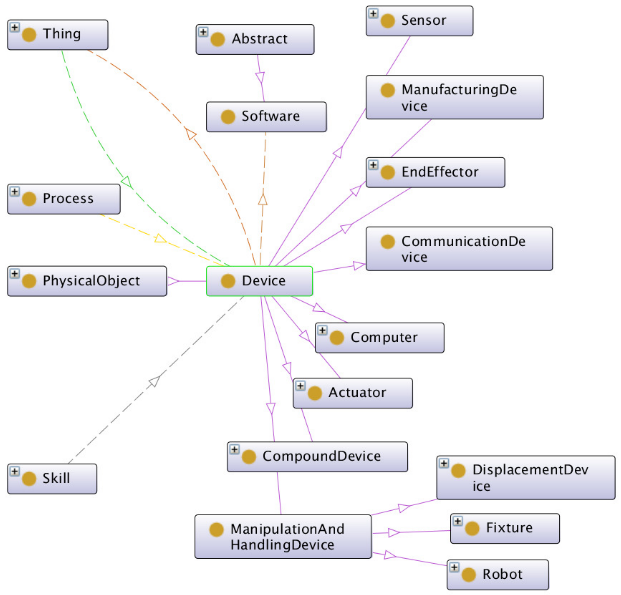
PhysicalObject A work cell consists of *PhysicalObject*s. Some objects, *Devices*, are active and have skills, while other, *Workpieces*, are passive and are being manipulated by the devices.

Property The *Property* hierarchy enumerates those properties of devices and skills which are interesting for the skill server to reason about.

Skill A *Skill* represents an action that might be performed (by a device) in the context of a production process.

Task The definition of a *Task* concept. It is not yet being used, but serves as a placeholder for possible future extension.

[3] An *operation* has been defined earlier as an instantiated skill. It is the basic element of task representation.



```
Start ::= Element*;
abstract Thing : ComplexElement ::=;
abstract Element;
ComplexElement : Element ::=  OwlIdentifier Attribute* Element*;
ValueElement : ComplexElement;
RdfDeclaration : ComplexElement;
abstract SimpleElement : Element ::= &lt;LITERAL&gt;;
Attribute ::= Value;
Value ::= &lt;STRING_LITERAL&gt;;
OwlIdentifier ::= &lt;IDENTIFIER&gt;;

PhysicalObject : Thing ::= hasProperty:Thing*;
Device : PhysicalObject ::= skill:Thing* subDevice:Thing* software:Thing*;
Abstract : Thing ::=;
Software : Abstract ::=;
Skill : Thing ::= hasProperty:Thing* isSkillOf:Thing*;
EndEffector : Device ::=;
Actuator : Device ::=;
CompoundDevice : Device ::=;
Sensor : Device ::=;
ManufacturingDevice : Device ::=;
CommunicationDevice : Device ::=;
Computer : Device ::=;
ManipulationAndHandlingDevice : Device ::=;
DisplacementDevice : ManipulationAndHandlingDevice ::=;
Fixture : ManipulationAndHandlingDevice ::=;
Robot : ManipulationAndHandlingDevice ::=;
```

Fig. 2. An OWL ontology and the corresponding generated abstract grammar. Solid edges indicate an *is superclass of* relation, and dashed edges an *is part of* relation.

Most devices are not useful in isolation in a manufacturing cell, but must be combined with other devices to make a meaningful *compound device*. For exam-

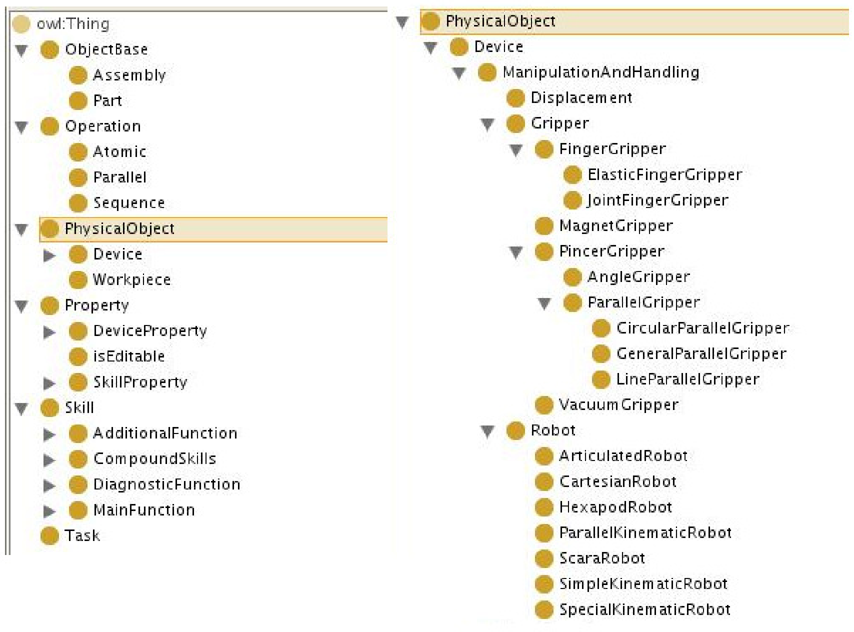


Fig. 3. Parts of the SIARAS robotics ontology.

ple, consider a possible set of devices needed for an industrial robot to perform drilling in workpieces: robot controller, I/O board, robot arm, drilling machine, drill bit. None of these devices is by itself capable of drilling a hole at a specified location; a drilling machine can not position itself at the correct position, nor can it drill a hole without a drill bit attached to it. Only by connecting[4] together all the devices mentioned above may the resulting *compound device* perform a drilling operation.

Since the skill server is supposed to generate configurations for a robot cell, it must also be able to reason about how, and when, devices are connected to each other. We have therefore introduced a device relationship in the ontology, $hasSubDevice \leftrightarrow isSubDeviceOf$, in order to model compound devices. A difference compared to other relations in the ontology is that it is dynamic instead of static. Device instances in a device library will not typically be statically connected to any other device instance. Instead, a task description where a specific device instance is used, must also specify how it is connected to other devices listed in the task description. An example on specifying device relations is shown in Fig. 4.

[4] *Connect* should here not be taken literally but in a logical sense: controls/is controlled by.

```
controller_1: ABB_IRC5
ioboard_1: dig328
robot_1: ABB_IRB-140
clamp_1: AngleGripper
drillmachine_1: Bosch_GBM_10_RE
drillbit_1: DrillBit_HSS_8mm

SubDevice: controller_1,ioboard_1
SubDevice: controller_1,robot_1
SubDevice: drillmachine_1,drillbit_1
SubDevice: robot_1,drillmachine_1
```

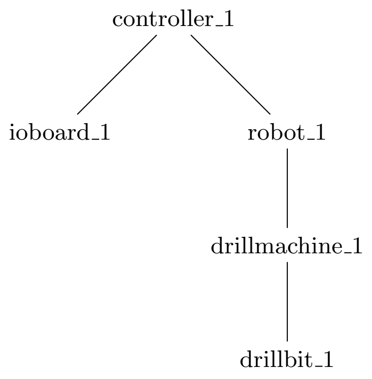


Fig. 4. Device specification from a task description on the left. Corresponding device tree on the right.

We should also keep in mind that device relations may change during execution of a task description, for example by using a tool exchanger. Revisiting the example in Fig. 4 using a tool changer, we get a changing device tree such as one shown in Fig. 5. In the beginning of the task description, there is no device attached to the robot arm (if we do not consider the tool exchanger itself) — the middle tree in the figure. When the robot attaches a drilling machine, the device tree transforms to the left one. Finally replacing the drilling machine with a gripper results in the rightmost version of the device tree.

Yet another aspect of combining devices in compound ones is computing their properties out of the properties of their elements. In come cases this operation is obvious: e.g., a gripper can hold an object, thus a robot equipped with a gripper can also hold an object (simple inheritance). However, the allowed payload for such a compound device will not be inherited, but rather computed in a particular way. For example:

$$\min(\text{payload}(robot) - \text{weight}(gripper), \text{payload}(gripper))$$

There seems to be no obvious way to devise a generic inheritance mechanism for compounds; we currently assume that this will be specified by the user, although other possibilities are investigated.

3.2 Handling Knowledge

A lot of what the skill server is really about, is to transform information (knowledge) between different representations. First, the skill server needs to parse an ontology description, various local ontology extensions and a number of device descriptions from different device libraries, and build an internal representation of how the various parts of a manufacturing cell (devices, other physical objects, software, etc.) are interconnected, which is suitable for performing reasoning and

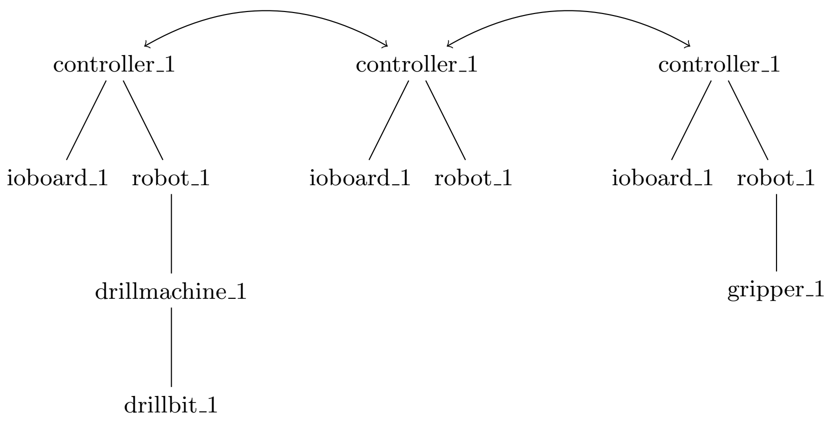


Fig. 5. Changing device tree when using a tool exchanger with the robot.

feasibility analysis. In the other end of the skill server pipeline, it needs to be able to generate configurations for the industrial robot cell. This is actually, at some level of abstraction, quite similar to what is done by any compiler for a programming language.

But, unlike a traditional compiler, the skill server is also used as a knowledge manager, keeping an abstract knowledge representation of the cell. As the manufacturing process executes. the cell state changes. For example, state changes occur when tools are replaced using tool changers or when work pieces are joined (glued, welded, screwed, etc.) together to finally form a product. As the cell state changes, the skill server internal representation of the device configurations must also change. As they are parts of a tree structure, we can simply model manufacturing cell state changes as moving branches of the syntax tree from one position to another.

Data inferred from the knowledge base is described as attributes of syntax tree nodes, declaratively defined by equations in a JastAdd application aspect. The declarative definition allows the information to be automatically updated whenever the syntax tree is changed. Currently, this is done simply by flushing all cached attribute values. New values are then computed on demand as needed. This works well as long as the syntax tree is not very large, and has not been a practical problem for our applications so far.

As an example of the use of JastAdd application aspects, consider the scenario shown in Fig. 5 where a robot is supposed to switch tools from a drilling machine to a gripper in order to be able to fulfill the operations mandated by the robot cell task. The skill server will then first check which ones of the available drilling machines, if any, could be used to perform the upcoming operations. Restrictions are given, for example, by the width and depth of the hole to drill, and by the workpiece material.

As a simplified version of this problem, consider the following grammar.

```
Start ::= Element*;
abstract Element;
Skill : Element ::= <Id> Property*;
Grasp : Skill;
Drill : Skill;
Device : Element ::= <Id> SkillUse*;
SkillUse ::= <Id>;
Property ::= <Value>;
```

To find the devices that can grasp, we need to look in each device, find which skills the device has by matching the `SkillUse` nodes to the appropriate `Skill` nodes, and finding out if one of those `Skills` is a `Grasp` object. This is accomplished by the following attributes and equations.

```
coll Set<Device> Start.devicesWithGrasp()
  [new HashSet<Device>()] with add;

Device contributes this
when canGrasp()
to Start.devicesWithGrasp() for root();

syn boolean Device.canGrasp() {
  for (SkillUse u : getSkillUses()) {
    if (u.decl() != null && u.decl().canGrasp()) return true;
  }
  return false;
}

syn Element SkillUse.decl() = lookup(getId());
inh Element SkillUse.lookup(String id);
eq  Start.getElement(int index).lookup(String id) {
  for (Element e : getElements()) {
    if (e.matches(id)) return e;
  }
  return null;
}

syn boolean Element.canGrasp() = false;
eq  Grasp.canGrasp() = true;

syn boolean Element.matches(String id) = false;
eq  Skill.matches(String id) = id == getId();

inh Start Device.root();
eq  Start.getElement(int index).root() = this;
```

This works as follows. The root of the abstract syntax tree, i.e., the `Start` node, has an attribute `devicesWithGrasp` that is a list of the devices we are looking for. It is defined as a so called *collection* attribute to which so called *contributions* contribute elements. In this case each `Device` contributes itself to this collection if it can grasp things.

To check if a `Device` can grasp things, it checks through its `SkillUses`. These are bound to `Skill` objects through a reference attribute `decl`, which is in turn defined through an inherited attribute `lookup`. The attributes `canGrasp`, `matches`, and `root`, are helper attributes.

Note that more code is needed to implement the actual reasoning, i.e., to match the set of restrictions imposed by the workpiece and operation to be carried out onto the set of properties of the retrieved devices. Based on some (given) optimization criteria the “best” device will be chosen.

4 Evaluation

The current JastOwl prototype, consisting of about 1500 lines of JastAdd code, can analyze a non-trivial OWL document and then generate a JastAdd abstract grammar, as well as a JavaCC parser description, for the description language as described by the OWL document. Regardless of which changes are done in the OWL-based specification, both the abstract and concrete grammars for the description language can be automatically generated.

In order to comprise a fully usable application, in the form of a dedicated compiler, application code, here in the form of JastAdd aspects, is needed. If the ontology changes, there is a possibility that the application code has to be changed as well. However, due to the use of high-level attribution mechanisms, the code is relatively insensitive towards changes in the syntax tree structure and to additions of new ontology classes or relations. In particular, equations for inherited attributes apply to complete subtrees and are therefore relatively insensitive to minor changes in the possible forms of the syntax tree. Likewise for contributions to collection attributes.

Recapitulating the four requirements from Section 1 we find that they are all satisfied. Aspect orientation in the form of static code weaving enables us to add desired functionality to the generated front-end in a modular fashion. We may re-generate the grammars and front-end code while not risking to accidentally delete any of the manually supplied code. Reference attribute grammars, as part of JastAdd, supplies a compact way to implement references to data stored in nodes in different parts of the tree, in effect transforming it to a directed graph.

Also performance-wise the proposed method of automatically generating a JastAdd based dedicated compiler front-end for ontological knowledge seems to be a good choice. The SIARAS skillserver could use any one of two different back-ends to access a library of device knowledge; either a backend based on Protégé with Pellet as reasoner, or a backend based on JastAdd, developed using the JastOwl meta compiler. The task of, for example, returning all grippers capable of lifting at least 0.5kg took around 3 seconds to execute using the JastOwl meta-compiler approach, and more than 20 seconds using the Protégé/Pellet back-end.

5 Related Work

The idea to take some kind of schema representation and generate a dedicated parser, model classes, and serialization code, is used in many other tools. In particular, there are many XML tools that employ this idea. Examples include JAXB [4] which is a part of the Java SE platform. Similar techniques also exist for OWL, for example [9], and is implemented by several tools, including Protege.

Whereas these tools generate high-level classes and APIs for particular schema or ontologies, JastOwl differs by basing the generation on a corresponding abstract grammar, and by supporting the modular addition of application-specific functionality to the generated classes. Furthermore, this added functionality can be specified at a high level, using declarative reference attribute grammars. Because the JastOwl tool is itself generated, it is also possible to add alternative serialization formats easily, that work for any ontology.

There are several other tools that provide high-level processing of schema-based formalisms by making use of grammarware, but that focus on term rewriting rather than analysis and computations on an AST [1, 20].

An early approach to apply attribute grammars for schema-based notations was that of Psaila [18]. In this approach, it was suggested that the DTD schema for a class of XML documents was extended directly with attributes and equations to provide semantics to XML documents.

Cowan [2] presents an interesting way of connecting an OO Java model in the form of Javabeans with an RDF model using Java annotations and runtime reflection/introspection. However, no support for automatically generating Javabeans corresponding to a given RDF model has been found, and the developer is then left with the task of manually coding the needed Javabeans.

6 Conclusions

In this paper we have proposed how meta-compilation based on reference attribute grammars can be used for developing tools for analyzing and manipulating ontological knowledge databases. By implementing a meta compiler, in this case a compiler parsing an ontology description in OWL, producing both abstract and concrete grammars for a dedicated compiler, we can get rid of the often tedious and error prone work of implementing such applications, as well as simplifying the maintenance of them as the ontology changes.

The application code can be modularized as aspects separate from the generated compiler source code, and can be programmed at a declarative high level using attributes. The separation of user submitted code from generated code result in fairly good robustness to changes in the ontology.

The JastOwl meta compiler has so far been used in several ontology related experiments and prototypes. Originally developed within the SIARAS project [12] (`http://www.siaras.org/`), JastOwl has also been used in industrial robotics ontology experiments within the european RoSta project [14] (`http://www.robot-standards.eu/`). Currently there is ongoing work within the ROSETTA project
(`http://www.fp7rosetta.org/`) where we are investigating the possibilities of using ontologies in conjunction with self-describing communication protocols.

Experiences so far indicate that our method of using the JastOwl meta-compiler with the JastAdd toolkit is an efficient way, both in lines of code as well as regarding performance, for analyzing and/or manipulating ontological knowledge.

References

1. Bravenboer, M.: Connecting XML processing and term rewriting with tree grammars. M. Sc. thesis. Utrecht University (November 2003)

2. Cowan, T.: Jenabean: Easily bind javabeans to RDF (April 2008), http://www.ibm.com/developerworks/java/library/j-jenabean/index.html
3. Ekman, T., Hedin, G.: The JastAdd System - modular extensible compiler construction. Science of Computer Programming 69, 14–26 (October 2007)
4. Fialli, J., Vajjhala, S.: The Java Architecture for XML Binding (JAXB). JSR Specification (2003)
5. ontoprise GmBH: OntoStudio semantic modelling environment (2011), *http://www.ontoprise.de/en/products/ontostudio/*
6. Hedin, G.: Reference Attributed Grammars. In: Parigot, D., Mernik, M. (eds.) Proceedings of the 2nd International Workshop on Attribute Grammars and their Applications (WAGA99). pp. 153–172. INRIA Rocquencourt France (March 1999)
7. Hillairet, G., Bertrand, F., Lafaye, J.Y.: Bridging emf applications and rdf data sources. In: Semantic Web Enabled Software Engineering (SWESE 2008). Karlsruhe (Oct 2008)
8. Jena: Jena – a semantic web framework for java. http://jena.sourceforge.net/ (2009), site accessed on November 5, 2009
9. Kalyanpur, A., Pastor, D.J., Battle, S., Padget, J.A.: Automatic mapping of owl ontologies into java. In: Maurer, F., Ruhe, G. (eds.) SEKE. pp. 98–103 (2004)
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. Lecture Notes in Computer Science 2072, 327–355 (2001), citeseer.nj.nec.com/kiczales01overview.html, *http://eclipse.org/aspectj/*
11. LLC, C..P.: Pellet: Owl2 reasoner for Java (2011), *http://clarkparsia.com/pellet*
12. Malec, J., Nilsson, A., Nilsson, K., Nowaczyk, S.: Knowledge-Based Reconfiguration of Automation Systems. In: Automation Science and Engineering, 2007. CASE 2007. IEEE International Conference on. pp. 170–175 (2007)
13. Java-CC Parser Generator, metamata Inc. *http://www.metamata.com*
14. Nilsson, A., Muradore, R., Nilsson, K., Fiorini, P.: Ontology for robotics: A roadmap. In: Advanced Robotics, 2009. ICAR 2009. International Conference on. pp. 1–6 (June 2009)
15. Web ontology language (2004), *http://www.w3.org/2004/OWL/*
16. Web ontology language, version 2 (2009), *http://www.w3.org/TR/owl2-overview*
17. The protégé ontology editor and knowledge acquisition system (2009), http://protege.stanford.edu/
18. Psaila, G., Crespi-Reghizzi, S.: Adding semantics to XML. Workshop on Attribute Grammars, WAGA (1999)
19. SPARQL: SPARQL protocol and RDF query language. *http://www.w3.org/TR/rdf-sparql-query/* (January 2008)
20. Stap, G.: XML document transformation processes using ASF+ SDF. M. Sc. thesis. University of Amsterdam (2007)
21. Tsarkov, D.: FaCT++ (2007), *http://owl.man.ac.uk/factplusplus/*
22. Völkel, M.: Rdfreactor – from ontologies to programatic data access. In: Proc. of the Jena User Conference 2006. HP Bristol (Mai 2006)
23. W3C: Semantic web (2001), *http://www.w3.org/2001/sw*
24. Zimmermann, M.: Knowledge-Based Design Patterns for Detailed Ship Structural Design. Ph.D. thesis, University of Rostock (May 2010)