FIPARSE - A GENERIC PARSER FOR FIPA-COMPLIANT AGENT COMMUNICATION

Markus B. Söllner
Distributed and Mobile Systems Group
University of Bamberg
Feldkirchenstraße 22
96052 Bamberg, Germany
email: markus.soellner@web.de

Sven Kaffille and Guido Wirtz
Distributed and Mobile Systems Group
University of Bamberg
Feldkirchenstraße 22
96052 Bamberg, Germany
email: sven.kaffille@wiai.uni-bamberg.de

ABSTRACT
Software Agents communicate by exchanging messages formulated in an Agent Communication Language. In order to facilitate communication and understanding of each other, the domain of discourse of heterogeneous agents should be described in an explicit ontology. Both, the Agent Communication Language and the ontology definition, are independent of the implementation of agents. Furthermore the content of messages should be formulated in a standardized content language. So the content of messages and the representations of ontologies have to be transformed into a format that can be interpreted by an agent. Today many agents are implemented using Java technology. This paper presents a generic parser called *fiParse* that enables developers of Java-based agents to create representations of ontologies, namely Java classes and standardised message content that can be interpreted by their agents and is compliant to the FIPA standards. This works

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134

with ontologies defined up-front as well as on-the-fly by adding ontologies when agents are already up and running. Thus *fiParse* facilitates a seamless standardized integration between ontology and agent development.

KEY WORDS
Software Agents, Agent Communication, Ontology, FIPA Content Languages, Parser

1 Introduction

For working with software agents different standardisation approaches have been made during the last years. One of those approaches is the specification work of the FIPA concerning agent management and agent communication (http://www.fipa.org/). These formal specifications (more precisely parts of the specifications) have been realised as Java implemented agent platforms like, e.g., FIPA-OS [1], JADE [2], and Grasshopper [3].

In the FIPA context, Communication between agents takes place by means of exchanging messages based on the theory of speech acts [4]. These messages are formulated in a so called Agent Communication Language (ACL). One standardised and widely used Agent Communication Language is FIPA-ACL [5]. The data contained in a FIPA-ACL message can be divided into five categories: the communicative act [6], the participants in communication, control of conversation, content and description of content. The content of an ACL message is formulated in a content language (CL) and usually has to be interpreted regarding the interdependencies between the expressions as defined in an ontology [5]. While each agent platform includes a parser to convert ACL messages to fields and values in a Java object, the platforms lack a parser to parse the content in different CLs depending on different ontologies. So it is often necessary to implement a new, specific parser for the CL and ontology used with each special agent system.

fiParse is meant to close this gap. It has been implemented in Java as the most important agent platforms have also been implemented in Java. So *fiParse* can be used as plug-in for agent systems implemented on Java-based agent platforms. *fiParse* is able to read ontologies and to recognise the words and objects used. In order to do so, the ontology is mapped to hierarchies of Java classes and their properties. The data transferred as content of ACL messages is then written to instances of those classes in order to enable the agents to process them. *fiParse* has been designed to use ontology definitions formulated in different ontology languages like, e.g., RDFS [7], DAML+OIL [8], OWL [9] or FIPA Meta Ontology [10], in order to facilitate the usage of

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134

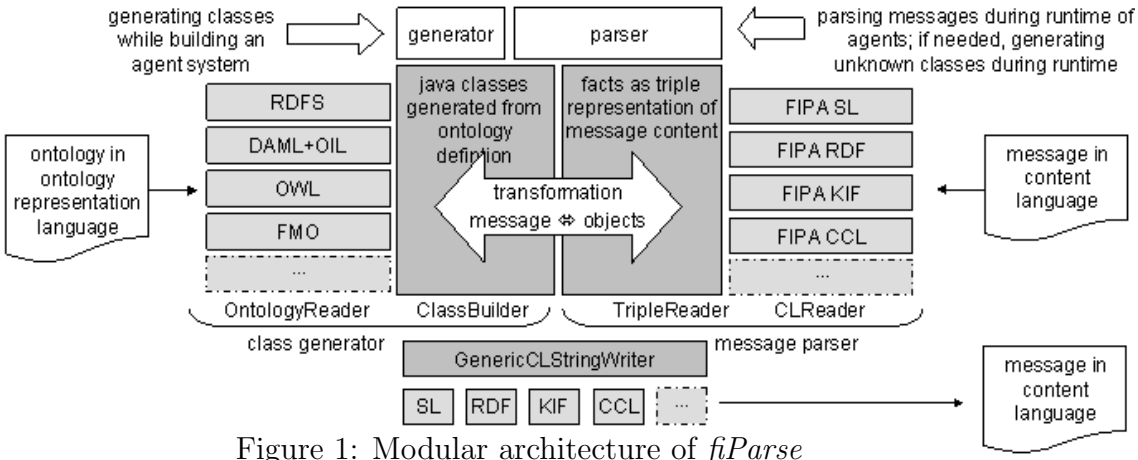

Figure 1: Modular architecture of *fiParse*

many existing and upcoming ontology definitions from many different vendors. Hence, it is able to deal with the heterogenity of ontologies in a multi agent environment.

The rest of this paper is structured as follows. The next section contains a summary of the design criteria the system is finally based on as well as a description of the architecture and the modules of the *fiParse* parser system implemented in Java. A short tutorial, how to use *fiParse* in design and usage of an agent system is given in section 3. An overview of related and future work in sections 4 and 5 concludes this paper.

2 Design and Implementation

This section first discusses the requirements of generic parser for agent platforms. Afterwards, a description of the overall architecture and the most important modules that constitute *fiParse* in order to meet these requirements are presented.

The first group of requirements are deduced directly from the FIPA specifications for agent communication as well as the structure of ACL messages. The basic functionality has to include:

- Reading of content and description parameters (CL, ontology) from ACL messages, not restricted to the message format of one specific platform implementation.
- Support for the three types of transported information in a FIPA ACL message: objects, propositions and actions [11].
- Definition of the words and expressions used in an explicit ontology known to all agents participating in conversations.

The second group of requirements is caused by the envisioned generics of *fiParse*:

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134

- Creation of Java objects and properties according to an *explicit* ontology written in an ontology representation language, as explicit ontologies ease distribution and sharing of domain knowledge.
- Ability to use different ontology representation languages with the parser system (RDFS, DAML+OIL, OWL, FIPA Meta Ontology). This requirement follows directly from the first one to facilitate the distribution and sharing of ontologies independent from agent implementations.
- Exact definition of configurable rules for transformation and mapping of definitions from ontologies to classes, primitive data types, etc. to facilitate the adaptation of *fiParse* to new ontology definition languages.
- Transformation of Java objects derived from an explicit ontology definition to message content using a string representation in different FIPA compliant CLs, e.g. RDF or SL [12], to be able to send them in an implementation independent format to other agents understanding FIPA compliant CLs and the ontology used.
- Retransformation of ACL message content written in different CLs to Java objects, derived from an explicit ontology definition. This enables Java-based agents to interpret messages received from other FIPA compliant agents.

Figure 1 presents the modular architecture of the *fiParse* parser system derived from the requirements described above. The figure shows the two main functions of *fiParse*. The modules on the left side (the class generator) are used to create Java objects according to ontology definitions in different ontology representation languages. The modules on the right side (the message parser) offer functions to transform and retransform message strings and objects. The two darker shaded modules are independent from a specific ontology or CL and hold commonly used functionality. On both sides, language specific modules for ontology representation languages and content languages can be added. To ensure the successful operation of the modular system and the extendibility of *fiParse* with new language specific modules the modules interactvia well defined interfaces. The different modules and their interaction with other parts of the *fiParse* system are sketched in the rest of this section.

OntologyReader modules: The OntologyReader interface has to be implemented for each ontology representation language that should be used with *fiParse*. The implementation must come up with the functionality to read ontology descriptions and to convert it to Java sources of classes and fields with get- and set-methods. The module returns the filenames of the source files created. To simplify the step of generating classes, the first implemented

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134

OntologyReader module (for RDFS as ontology representation language) provides the class SourceGenerator with the functionality of creating source code from an input DOM document containing class and property definitions in the node structure equal to the node structure of RDFS class hierarchies. So this module can be used without changes with other RDFS-based languages like DAML+OIL or OWL as well. SourceGenerator also sufficiently supports other frame-based knowledge representation forms like FIPA Meta Ontology. In case of using SourceGenerator with new OntologyReader implementations for non-RDFS-based languages, the only work that has to be done is parsing the input data to DOM and to restructure some nodes in the DOM document in order to suit the structure of a RDFS document.

Generic SourceGenerator: The generic SourceGenerator performs an analysis of the given DOM structure read from an ontology definition for class hierarchies and properties as well as the creation of matching Java source code. Each ontology class is mapped to a Java class, each property to a private field with appropriate public methods to manipulate the values of the object during runtime. SourceGenerator is able to process the following constructs of ontology representation languages:

- class definitions
- class hierarchies
- property definitions
- multiple domains
- multiple ranges
- cardinality restrictions
- functional properties
- data typing (boolean, integer, strings)
- container

To function properly, SourceGenerator needs to be configured for new ontology representation languages. A configuration file holding all necessary information to identify the nodes containing specific information has to be written. Configuration files for RDFS and OWL and an editor to create other configuration files are part of the *fiParse* implementation. The SourceGenerator returns the filenames of the created source files, so that the files can be processed by the ClassBuilder implementation.

ClassBuilder module: The ontology language independent ClassBuilder implementation processes the source code written by OntologyReader and SourceGenerator. It adds constructors and information about the ontology to the source files and finally compiles the Java classes. If the option addToClStringMethodToEachObject(see table 1) is set to true, a method toCLString() is added to each class, containing serialization definitions for all

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134

CLs stated on clStringLanguages. If no such method is added, the objects are serialized by an external CLStringWriter. The call to this CLStringWriter is accomplished by the common super class of all created classes (OntoObject) that provides common functions of all objects created from ontologies.
Super class OntoObject: The class OntoObject adds some important functions to all inheriting classes:

- The field CLASSID must be set by each inheriting class and must state a unique ID to identify the class.
- The field objectID ensures secure identification of each instance of a class.
- The method toCLString(String cl) is used to serialize an object to a string in the given CL.
- The field usedObjects is filled with all other OntoObjects depending from an OntoObject to ensure that depending data is written to an ACL message as well.
- The methods equals() and hashCode() from java.lang.Object are overridden to compare OntoObjects using their unique ID.
- The static method getSuperClass(Class clazz) can be used to check whether a class is sub class of OntoObject.

Container class OntoVector: For dealing with multiple values of one property and container definitions in ontologies, the container class OntoVector is used. It is an extension of java.util.Vector with additional functionality to select objects in an OntoVector by their type.
Individual Settings: For each ontology, Java classes are built for, the user of *fiParse* may set individual parameters by writing a configuration file using the editor mentioned above or by using the set-methods of the class FiParseSettings. All parameters are shown in Table 1. After successfully creating all classes, the settings of each ontology are saved in configuration files and used by the message parser when reading or writing messages using the objects of that ontology.
FIPA specific objects: In addition to the objects created from ontologies, some FIPA specific objects are needed to ensure that all types of messages used in FIPA communicative acts can be represented. Those objects, i.e., Action, Statement, Proposition, Code and Rule, are created according to the RDF Schemas in the FIPA RDF Content Language Specification. The two classes UnaryOperator and BinaryOperator can be used to represent logical operators.
CLReader modules: The CLReader interface has to be implemented for each CL that should be used with *fiParse*. The content string has to be

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134

Parameter	Description
OntologyName	name of the ontology; also used as parameter ontology in ACL messages
OntologyLanguage	ontology representation language; used to identify the correct OntologyReader implementation
OntologyFile	file containing the ontology description in the language set in OntologyLanguage
createDatabase	boolean flag indicating, whether a database for living objects should be created; if true every object is written to type-safe containers in a database to allow operations on all living objects of a specified class
fullURL	boolean flag when using URIs or URLs as CLASSID and objectID; if true, the full URIs/URLs are used as class names and field names, if false only the leading part including the # is cut
packages	package the created classes are put in
setMethodPrefix	prefix for set-methods; standard is set
addMethodPrefix	prefix for add-methods; standard is add
getMethodPrefix	prefix for get-methods; standard is get
containsMethodPrefix	prefix for contains-methods; standard is contains
isEmptyMethodPrefix	prefix for isEmpty-methods; standard is isEmpty
nextMethodPrefix	prefix for next-methods; standard is Next
allMethodPrefix	prefix for all-methods; standard is All
byTypeMethodSuffix	suffix for byType-methods; standard is ByType
noCardinalityMeansMax1	boolean flag, indicating whether no explicitly stated cardinality for a property means maxCardinality = 1 (true) or cardinality 0,* (false)
sourcesPath	relative path to the folder, where the Java sources are to be put, standard is src
addToClStringMethodToEachObject	boolean flag, which version of serializing objects should be used; if true, a method toCLString() is added to created classes for all CLs states in clString Languages; if false, the external CLStringWriter is used
clStringLanguages	CLs to be added to the toCLString() methods
writeBehaviour	behaviour when writing source files; BACKUP: backup the old sources, OVERRIDE: write the new sources without backupping the old ones, IGNORE: do not create new sources
printCLStringInOneLine	global boolean flag for output of CLStrings in log files; whole string in one line (true) or formatted (false); standard is false;
DS	directory seperator; automatically set to on windows, to / on unix systems

Table 1: *fiParse* settings.

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134

parsed and the included facts are represented as triples of a specific format. These triples are returned to the TripleReader module in a vector. For each object definition in the string, a type-triple of the form (objectID, instanceOf, CLASSID) must be added to the triple vector. All properties of objects are represented by fact-triples of the form (objectID, property name, value). If a property has more than one value, for each value a triple is added.

TripleReader module: The TripleReader module is the generic part of the message parser. It uses the type-triples received from the CLReader to instantiate FIPA objects like Action and Proposition and objects as defined by the ontology specified in the ontology parameter of the ACL message. Each object is provided with its objectID stated in the triple. When all objects are instantiated, the values for properties are set according to the fact-triples by using the set- and add-methods of the created objects.

CLStringWriter modules: To be able to write object data to message strings, the interface CLStringWriter has to be implemented for each content language, a CLReader implementation is added to *fiParse*. The two kinds to serializing objects in *fiParse* are implemented in CLStringWriter by means of two methods: the method writeCLStringMethod() is used to create Java code for serialization of created objects and the method getCLString() directly returns a content string for a given OntoObject. The required data are extracted from classes using reflection. The developers of new CLStringWriter implementations do not have to implement reflection for each module, as two methods doing that part of the work are included in GenericCLStringWriter.

GenericCLStringWriter module: Besides the reflection-based methods to extract data from classes, the GenericCLStringWriter module is used to link the CLStringWriter implementations for different CLs to the *fiParse* system. A call to methods in GenericCLStringWriter chooses the proper CLStringWriter module for the CL and forwards the call.

FiParser front-end: The users of *fiParse*, i.e., the developer of agent systems and agents, may access *fiParse* using the methods of the class FiParser. There are two possibilities to generate classes from an ontology definition. One way is to create a settings file with the *fiParse*-editor and using the main method of FiParser with the ontology name as parameter. The call of FiParser.main() loads the settings of the settings file and calls the method createClasses(). A direct call of this method after setting all parameters in the class FiParseSettings is the second way to generate classes. This way can be chosen during runtime of an agent to create unknown classes on the fly. FiParser uses a factory to choose the suitable OntologyReader for the ontology representation language used and starts ClassBuilder and OntologyReader functions.

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134

Parsing messages follows the same procedure; after a factory has been chosen, the proper CLReader, TripleReader and CLReader are called to parse the message. An agent can initiate parsing by calling the method parseMessage() in FiParser with either an ACL object or Strings containing content, ontology and CL as parameters.
Implemented modules: At the current state, the following modules of *fiParse* are implemented and running:

- all generic modules (FiParser, ClassBuilder, TripleReader, GenericCLStringWriter)
- OntologyReader modules for RDFS and OWL that may be utilised for all other RDFS-based knowledge representation languages using a configuration file
- generic SourceGenerator for DOM representation of ontologies to ease implementation of further OntologyReader modules
- CLReader and CLStringWriter for the FIPA RDF Content Language

The development of CLReader and CLStringWriter for the FIPA SL Content Language is under way.

3 Using *fiParse* - An example

The scenario used is a simple agent system that shall support information exchange about time-tables of a railroad company.

The first step of system design is to create and agree upon a suitable ontology. A very simple ontology for the railroad example defines the following facts:

- A train has an identifier.
- A train has a number of seats with a status, i.e. free/reserved, each.
- A train departs and arrives in railway stations at particular times.
- Each railway station has a unique name.

For the further work with the ontology it may be appropriate to create a graphical representation of the ontology, e.g., an UML class diagram.

The second step is to formalize the facts of the ontology in an ontology representation format understood by *fiParse*. Ontology definitions in RDFS or OWL can be created using ontology editors in a relative simple way. The ontology editors Protege 2000 [13] and OilEd [14] are freeware and both use a GUI to allow the creation of class hierarchies and allocation of properties. A part of the railroad ontology formalized in OWL is shown in Listing 1.

Listing 1: Example railroad ontology.

```
<?xml version="1.0" encoding="UTF-8" ?>
```

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax
   -ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns="http://lspi.wiai.uniba.de/fiparse#">
<owl:Ontology rdf:about="railroad"/>
<owl:Class rdf:ID="Train"/>
<owl:Class rdf:ID="Station"/>
<owl:Class rdf:ID="Seat"/>
...
<owl:ObjectProperty rdf:ID="hasSeat">
 <rdfs:domain rdf:resource="#Train"/>
 <rdfs:range rdf:resource="#Seat"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="departs">
 <rdfs:domain rdf:resource="#train"/>
 <rdfs:range rdf:resource="#Station"/>
 <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
    FunctionalProperty"/>
</owl:ObjectProperty>
...
<owl:DatatypeProperty rdf:ID="train-id">
 <rdfs:domain rdf:resource="#train"/>
 <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema
    #string"/>
 <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
    FunctionalProperty"/>
</owl:DatatypeProperty>
...
<owl:DatatypeProperty rdf:ID="status">
 <rdfs:domain rdf:resource="#Seat"/>
 <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema
    #boolean"/>
 <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
    FunctionalProperty"/>
</owl:DatatypeProperty>
</rdf:RDF>
```

This formal representation of the ontology can be processed by *fiParse* to create matching Java classes. The *fiParse*-editor is used to create a settings-file. After starting the editor, necessary parameters are set. All missing values are set to standard values:

```
packages railroaddemo.data
ontologyFile /ontologies/train.owl
ontologyLanguage owl
```

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134

After saving the settings in a file named railroad.ocfg, *fiParse* can start to generate the classes. The filename without the suffix '.ocfg' must be the name used to identify the ontology in the agent system later. The Java classes are generated by calling the main method of FiParser with the name of the ontology as parameter. *fiParse* places the classes in the specified package, so that they can be used by the implementation of the agent system.
To serialize an object, e.g. an Action, to be content of an ACL message, the method toCLString() is used:

```
Action a = new Action();
...
ACL acl = new ACL();
acl.setContent(a.toCLString(FiParser.RDF));
```

To parse a message, the method parseMessage() in FiParser must be called. It returns an OntoVector holding all objects included in the message. The get-methods of OntoVector are used to extract the desired objects:

```
ACL acl = conv.getLatestMessageIndex();
OntoVector obj = FiParser.parseMessage(acl);
Action a =
(Action) obj.getNextByType(Action.class);
```

Information about CL and the ontology is read from the ACL object. If no information is included, *fiParse* tries to use the ontology and CL most recently used as a default.

4 Related work

There is some work closely related to *fiParse* that has been useful developing *fiParse* or can be used with it. First *OntoJava* [15] has to be mentioned. This is a cross compiler that translates ontologies written with Protege and Rules written in RuleML into a unified Java object database and rule engine. Some classes of *fiParse* are based on OntoJava.
Furthermore, there is the Java Compiler Compiler (JavaCC) (tm) [16], which is a useful tool to implement parsers in Java. JavaCC could be used to write parsers for agent applications instead of using *fiParse*, but JavaCC is not intended as a tool to generate classes from ontology definitions and therefore is no replacement for *fiParse*. JavaCC has to be seen as a complementary tool to *fiParse* that may be used to create parsers for ontology definitions to provide new OntologyReader modules and to implement parsers for CLs to provide new CLReader modules to *fiParse*.
Protege 2000 is an Ontology editor for RDFS and OWL and a knowledge acquisition tool that can be used to provide Ontology definitions to *fiParse*

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134

for the creation of Java classes. The same holds for *OilEd*in the context of DAML+OIL.
Other useful software that *fiParse* can be used with are Java-based FIPA compliant agent platforms. To mention only two of them, examples are *FIPA-OS* [1] and *JADE* [2], which are both free FIPA compliant agent platforms.

5 Conclusion and Future Work

The usage of *fiParse* as a plug-in adds a set of advantages to agent platforms. It simplifies the development of software agents because it is no longer necessary to implement special parsers for each application. The definition of a formal ontology is an essential step during the development process of most agent systems. *fiParse* uses this formal ontology definition, so that almost no additional work has to be done to obtain a working parser. Additionally, *fiParse* automatically creates Java classes for all needed data objects according to the ontology definition. Without the help of *fiParse*, that task would be an additional burden for the developer.

Agents able to "learn" new ontologies during runtime can use *fiParse* to generate Java classes for the newly learned expressions and words. Such a procedure is a bit more sophisticated, but possible and intended to be used with software agents.

Future work on *fiParse* is concerned with adding more pre-defined language modules to allow the usage of more CLs and ontology representation formats. The current implementation of the parser system serves as a framework and facilitates these development efforts.

Nevertheless, with the modules yet implemented and in work, *fiParse* is fully operational and ready for practical use.

References

[1] emorphia, *FIPA-OS FIPA Open Source Agent Platform*, www.emorphia.com/research/about.htm, (Harlow: 2002)

[2] Giovanni Rimassa, *Runtime Support for Distributed Multi-Agents Systems*, http://jade.tilab.com/papers/Rimassa-PhD.pdf, (Gennaio: 2003).

[3] IKV++ Technologies AG, *Grasshopper 2 - The Agent Platform*, http://www.grasshopper.de/, (Berlin: 2003)

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134

[4] J.R. Searle, *Speech Acts* (Cambridge: University Press, 1969).

[5] Foundation for Intelligent Physical Agents, *FIPA ACL Message Structure Specification, Document No. 00061*, (Geneva: FIPA, 2002).

[6] Foundation for Intelligent Physical Agents, *FIPA Communicative Act Library Specification, Document No. 00037*, (Geneva: FIPA, 2002).

[7] World Wide Web Consortium, *RDF Vocabulary Description Language 1.0: RDF Schema*, http://www.w3.org/TR/rdf-schema/, (2004)

[8] Defense Advanced Research Projects Agency, *DAML+OIL (March 2001)*, http://www.daml.org/2001/03/daml+oil-index.html, (2001).

[9] World Wide Web Consortium, *OWL Web Ontology Language Overview*, http://www.w3.org/TR/owl-features/, (2004).

[10] Foundation for Intelligent Physical Agents, *FIPA Ontology Service Specification, Document No. 00086*, (Geneva: FIPA, 2001).

[11] Foundation for Intelligent Physical Agents, *FIPA RDF Content Language Specification, Document No. 00011*, (Geneva: FIPA, 2001).

[12] Foundation for Intelligent Physical Agents, *FIPA SL Content Language Specification, Document No. 00008*, (Geneva: FIPA, 2002).

[13] N.F. Noy et al., *Creating Semantic Web Contents with Protege-2000*, http://smi-web.stanford.edu/pubs/SMI_Abstracts/SMI-2001-0872.html, (Stanford: IEEE Intelligent Systems, 2001).

[14] Sean Bechhofer et al., *OilEd: a Reason-able Ontology Editor for the Semantic Web*, 14th Intern. WS on Description Logics, http://oiled.man.ac.uk/publications.shtml, (Stanford: 2001).

[15] Dr. Andreas Eberhart, *OntoJava*, www.aifb.uni-karlsruhe.de/WBS/aeb/ontojava, (Karlsruhe: 2001).

[16] Sun Microsystems Inc., *Java Compiler Compiler (tm) - The Java Parser Generator*, https://javacc.dev.java.net/, (2003).

This is a draft of the paper published in: P. Kokol (Editor), *Proceedings of the International Conference on Software Engineering*, 2005
http://www.actapress.com/Abstract.aspx?paperId=19134