JAR2ONTOLOGY - A TOOL FOR AUTOMATIC EXTRACTION OF SEMANTIC INFORMATION FROM JAVA OBJECT CODE

Nicolás Marín, Clara Sáez-Árcija* and M. Amparo Vila
Department of Computer Science and Artificial Intelligence, University of Granada, Granada, Spain

Keywords: Object code analysis, Semantics extraction, Java, OWL.

Abstract: We present here a novel approach (and its implementation) for the automatic extraction of semantic knowledge from Java libraries.
We want to match software libraries, so we need to obtain as much information as possible to use it in the matching process. For this purpose, this approach extracts information about the structure of the classes (i.e., name, fields and hierarchy), as well as information about the behavior of the classes (i.e., methods).
In the literature, to our knowledge, it can be only found lightweight approaches to the extraction of this kind of information from Java object code. The approach is implemented in an automatic extraction tool (called *Jar2Ontology*) that has been developed as a plug-in of the Protégé Ontology and Knowledge Acquisition System. Jar2Ontology extracts the semantics from Java libraries and translates it into OWL (Ontology Web Language).

1 INTRODUCTION

Information systems integration is a problem of increasing interest in many areas, as in Business Intelligence, Customer Relationship Management, Enterprise Information Portals, E-Commerce or E-Business. Interested lectors can find remarkable surveys of research in this area in (Kalfoglou and Schorlemmer, 2003; Rahm and Bernstein, 2001; Shvaiko and Euzenat, 2005; Wache et al., 2001).

Often, software libraries are part of the information systems, and for this reason, information systems integration should entail software libraries integration. Nevertheless, there is not any integration system that performs software libraries matching. In this work, we focus on the representation of software libraries with the aim to integrate them.

There are many definitions about information integration, and we can find in (Sáez-Árcija et al., 2009) the next one (it is a compendium of the before mentioned definitions).

Definition 1. Information Integration *is the task that aims at building a global system which provides an unified access to the information from many information sources. Those information systems are distributed (placed in different places), autonomous (independently managed) and heterogeneous (with different software, hardware, data model, etc.).*

This work is devoted to the first step of the information integration process, i.e., the semantics extraction. Our goal is to integrate software libraries, and for this reason, we have to extract the semantics of the libraries and model them.

We present here a novel approach (and its implementation) for the automatic extraction of semantic knowledge from Java object code. We have chosen ontologies to represent the obtained knowledge, because most of the integration systems are based on ontology matching techniques. Java data model has been chosen because Java programming language is one of the most extended general-purpose object-oriented languages.

We want to point out that the proposed approach has been implemented as *Jar2Ontology*, a plug-in for the widely used Protégé Ontology Editor and Knowledge Acquisition System[2]. This tool extracts information directly from Java jar files, that is Java object code, so we do not need the source code to model the semantics of the library.

Most of tools for conceptual modeling allow to express information about the structural component of the concepts, and integration systems use this type of

*Corresponding author

[2]http://protege.stanford.edu

Marín N., Sáez-Árcija C. and Amparo Vila M..
JAR2ONTOLOGY - A TOOL FOR AUTOMATIC EXTRACTION OF SEMANTIC INFORMATION FROM JAVA OBJECT CODE.
DOI: 10.5220/0003510302670276
In *Proceedings of the 13th International Conference on Enterprise Information Systems* (ICEIS-2011), pages 267-276
ISBN: 978-989-8425-53-9

knowledge in the matching process. In our case, the conceptual model is the Java data model, where the concepts are the library classes and its structural component refers to the classes name and fields, as well as to the class hierarchy.

Nevertheless, in some conceptual models (e.g., object-oriented data model), the semantics consists of two types of knowledge: structural and behavioral knowledge, that are related to the structural component and the behavioral component of the model, respectively. In our case, the behavioral component refers to the information about the methods of the classes. The use of this behavioral information can enrich the matching process thanks to additional criteria concerning to the behavioral component.

It is important to point out that, thought our research has been oriented to the data integration context, semantic knowledge extraction from Java libraries can be useful not only for data integration, but also for many more applications, as for example, automatic generation of code documentation, or reverse engineering.

The paper is structured as follows. Initially, section 2 presents a brief state of the art on semantic knowledge extraction and code analysis. In section 3 we focus on how to face the extraction of semantic information from a jar file and how to write it in the form of an OWL (Ontology Web Language) ontology. Next, in section 4 we present the diverse types of ontologies that can be obtained after the semantic extraction process. Section 5 is devoted to introduce some implementation issues of the proposed approach and example experimentation. Finally, in section 6, concluding remarks end the paper.

2 RELATED WORK

Many research efforts have been made on the field of automatic semantic knowledge extraction during last years. There are many works that aim to obtain a formal representation of the semantics that underlies a variety of sources, as for example, plain text (e.g. (Buitelaar et al., 2008; Wimalasuriya and Dou, 2010)), semi-structured documents (e.g. (DuL, ; Thiam et al., 2009)) or relational database schema (e.g. (Curino et al., 2009; Myroshnichenko and Murphy, 2009)).

Yet on the object analysis area, we can find many interesting works. Code analysis provides support for many applications, as program understanding (e.g. (Jakobac et al., 2005)), hardware design (e.g. (Martino et al., 2002)), software metrics (e.g. (Wong and Gokhale, 2005)), security testing (e.g. (Herbold et al., 2009; Hong et al., 2009; Letarte and Merlo, 2009; Spoto et al., 2010)), software design (e.g. (Amey, 2002)) and reengineering (e.g. (Herbold et al., 2009; Kawrykow and Robillard, 2009)). Most code analyzers examine C/C++ (e.g. (Martino et al., 2002; Spinellis, 2010; Wong and Gokhale, 2005)) and Java (e.g. (Jakobac et al., 2005; Kawrykow and Robillard, 2009)) source code, but we have also found works about PHP (e.g. (Letarte and Merlo, 2009)) and SPARK (e.g. (Amey, 2002)). Most of these approaches take *source code* as input data.

Although information extraction from source code is straightforward, in many cases it is not possible, simply because source code is not available. Therefore, if we want to develop a tool as general as possible, we have to face the difficult task of analyzing in detail *object code*.

With respect to object code, we can cite (Hong et al., 2009; Jackson and Waingold, 1999; Spoto et al., 2010) as examples of Java Byte Code analysis. Nevertheless, only one of these approaches (Jackson and Waingold, 1999) is focused on the representation of the semantic knowledge of the analyzed object code, and it only represents the structural component of the model in a UML (Unified Modeling Language) diagram. The main goal of this work is to extract the structural component from the analyzed object code, as well as the behavioral one.

3 SEMANTIC MODEL EXTRACTION

We have carried out the task of developing an approach to semantically model the structure and the behavior of the classes embedded in a jar file. It means going one step beyond the traditional structural conceptual modeling.

We will distinguish two kind of ontologies obtained after the semantic model extraction process, depending on the information that we want to use. The fist kind of ontology (we call it *Data Ontology*) models only structural knowledge from the java library. Classes from the library are modeled as ontology classes and are organized in a class hierarchy that is analogue to the library class hierarchy. The other kind of ontology (we call it *Metadata Ontology*) is more comprehensive, because it models both structural and behavioral knowledge from the java library.

In this section we explain the processes of *Extraction of a Structural Model* and *Extraction of a Comprehensive Model* that obtain as a result a data ontology and a metadata ontology, respectively.

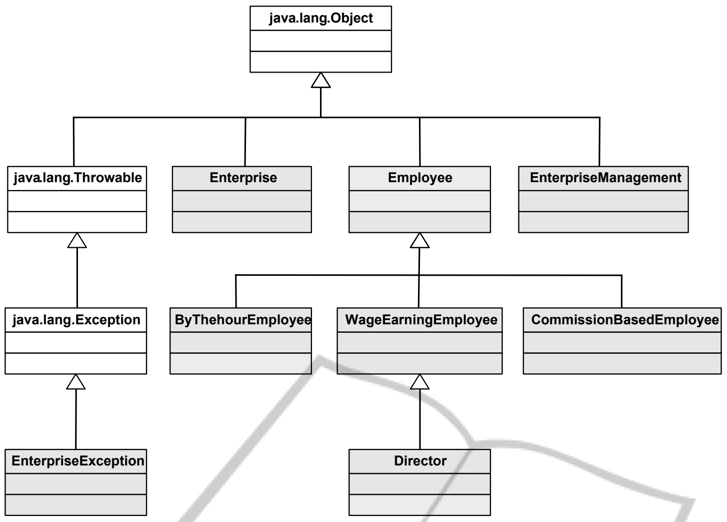


Figure 1: Enterprise management class diagram.

Let us present here an example to illustrate the explained processes of information extraction. It is a little library, that contains 8 class files representing information related to an enterprise and their employees. In figure 1 we can see the class hierarchy embebed in the jar file. Classes in white background are the classes related to the .class files of the library, and classes in dark background are the superclasses of these classes.

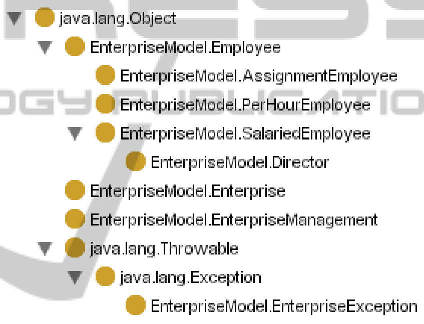
3.1 Extraction of a Structural Model

In this subsection we explain in more detail the process of extraction of the structural model from a Java library and its organization in the form of an OWL ontology. This process provides as a result an *Data Ontology*.

In (Gómez-Pérez et al., 2004) we can find the most extended way to express an object-oriented model by means of ontologies and, specifically, using OWL. On the basis of the ideas exposed in (Gómez-Pérez et al., 2004), we have modeled the library classes as OWL classes, and their fields as properties of these OWL classes. Each field is modeled as an OWL data type property if the type of the field is a datatype, or as an OWL object property, if the field type is a class. In this last case, this class is also modeled in the ontology. Each class is included into the ontology with its superclasses, obtaining thus a class hierarchy.

Figure 2 shows the classes of the ontology that is generated taking into account only the classes of the example (figure 1) that are embebed in the jar file with their superclasses.

We can see in figure 3 the properties of these classes, that are related to the Java classes fields. Properties in dark background are those that represent



```
▼ ● java.lang.Object
    ▼ ● EnterpriseModel.Employee
          ● EnterpriseModel.AssignmentEmployee
          ● EnterpriseModel.PerHourEmployee
        ▼ ● EnterpriseModel.SalariedEmployee
              ● EnterpriseModel.Director
      ● EnterpriseModel.Enterprise
      ● EnterpriseModel.EnterpriseManagement
    ▼ ● java.lang.Throwable
        ▼ ● java.lang.Exception
              ● EnterpriseModel.EnterpriseException
```

Figure 2: Classes in the data ontology that models the example library class hierarchy.

fields whose type is a class (OWL object properties). Properties in white background represent fields whose type is a datatype (OWL datatype properties).

In figure 4 we can see the process to create the ontology. This process is as follows: Initially, a hierarchy class that represents the library classes is created in the ontology. Then, the hierarchy is explored with the aim to add the fields of each class. This step often entails the creation of new classes, because the types of the added fields are classes that do not exist in the hierarchy. Thus, a recursive process is executed. It consists of adding fields to the new classes and adding new classes when it is necessary. At the end of the process, the class hierarchy has remarkably grown, because we add to the ontology the classes of the initial hierarchy fields together with the classes created by the subsequent fields addition.

Let us formalize the above introduced *Structural Model Extraction Process*.

Definition 2. Structural Model Extraction Process *is the transformation of a Java library L to an ontology O that satisfies:*

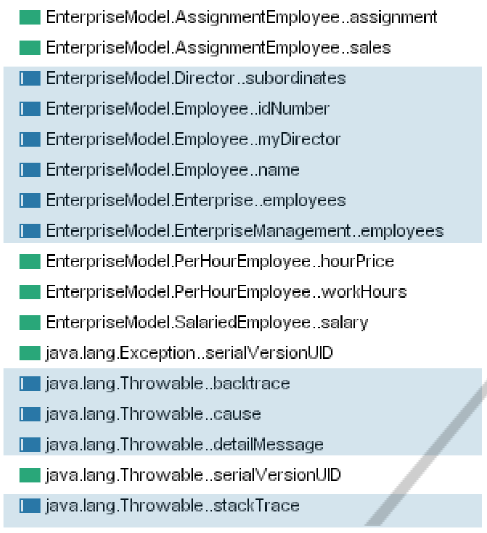

EnterpriseModel.AssignmentEmployee..assignment
EnterpriseModel.AssignmentEmployee..sales
EnterpriseModel.Director..subordinates
EnterpriseModel.Employee..idNumber
EnterpriseModel.Employee..myDirector
EnterpriseModel.Employee..name
EnterpriseModel.Enterprise..employees
EnterpriseModel.EnterpriseManagement..employees
EnterpriseModel.PerHourEmployee..hourPrice
EnterpriseModel.PerHourEmployee..workHours
EnterpriseModel.SalariedEmployee..salary
java.lang.Exception..serialVersionUID
java.lang.Throwable..backtrace
java.lang.Throwable..cause
java.lang.Throwable..detailMessage
java.lang.Throwable..serialVersionUID
java.lang.Throwable..stackTrace

Figure 3: Properties of the classes in the data ontology that models the example library.

- *For each class JC_i in the library L, there exists a corresponding class OC_i in the ontology O.*
- *For each superclass JC_j of a class JC_i, there exists a corresponding class OC_j in O, where OC_j is the superclass of OC_i.*
- *For each field f_j of a class JC_i, there exists a corresponding property p_j of the class OC_i in O.*
- *For each field f_j, if its range is a class JC_k, there exists a corresponding class OC_k in O.*

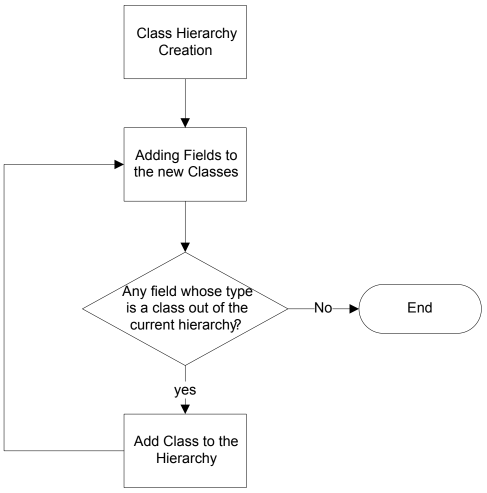

Class Hierarchy Creation
Adding Fields to the new Classes
Any field whose type is a class out of the current hierarchy?
No
End
yes
Add Class to the Hierarchy

Figure 4: Structural model extraction flow diagram.

Figure 2 shows the ontology at the beginning of this process for our example. This ontology has only 11 classes, but when the recursive process ends, the ontology contains 120 classes. These results have been obtained using *Jar2Ontology*, the tool that implements this approach.

We have shown how we can create an ontology that models the structural component of a Java library following the representation of object oriented data models that is most extended in current literature. This approach provides a data ontology whose class hierarchy is analogue to the original Java class hierarchy embedded in the library. However, this data ontology cannot represent all the structural metadata of a Java class. For this reason, we propose here a new approach that solves this problem.

3.2 One Step Beyond: Extraction of a Comprehensive Model

As we have already said, till now we have not taking into account all the semantic information that we can obtain from a Java class. For example, it is not able to represent if a Java class implements a certain interface, if it is public, private, or protected or which its package is. Furthermore, a conceptual model represented by means of an OWL ontology cannot represent information about the behavioral component of the classes (i.e. the set of methods of each Java class).

In this subsection we explain a more comprehensive approach that handles with all the class metadata that we are able to obtain from the object code: first, we do a more exhaustive study of the structure and, additionally, we also incorporate the behavioral information. This approach obtains as a result a *Metadata Ontology*.

To carry out this purpose, we create an OWL ontology that models the metadata of the conceptual model, i.e, a container in OWL to store information from conceptual models. Thus, when we want to represent the semantics of a conceptual model, we create instances of the ontology classes, i.e., we insert semantic information into the container. Let us see this in detail.

3.2.1 A Container in OWL to Store Information from a Conceptual Model

In order to represent all this semantic information in OWL, we have defined 4 classes into the ontology (*Class*, *Method*, *Local Variable*, and *Field*). Later, we will create instances of these classes to capture the extracted information.

In figure 5 we can see these four classes with their properties and their properties types. Let us see them in more detail.

- *ClassClass*: Each instance of this class models a Java class. The set of properties of the ClassClass is the following:

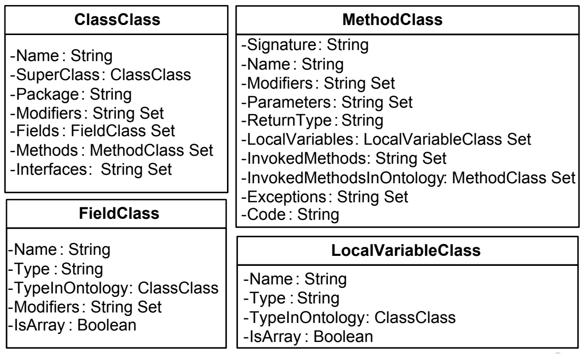


Figure 5: OWL Classes used to model the metadata extracted from Java libraries.

  - *Name*. It is a string which contains the name of the class.
  - *SuperClass*. This property relates the class to its direct superclass.
  - *Package*. This is a string with the name of the package of the class.
  - *Modifiers*. It is a list of the modifiers of the class. These modifiers indicate if the class is abstract, final, private, protected, public, static or strictfp. Furthermore, we can indicate that it is an interface.
  - *Fields*. It is a set of instances of the FieldClass that model the structural component of the class.
  - *Methods*. This property is a set of instances of MethodClass that model the behavior component of the class.
  - *Interfaces*. It is a set of strings with the names of the classes that are implemented by the represented class.

- *FieldClass*: It is used to model class fields. The properties of this class are detailed next.
  - *Name*. This property contains the name of the field in a string.
  - *Type*. It represents the type of the field using a string. If the field is an array, this property indicates the name of the type without the [] symbols.
  - *TypeInOntology*. This property relates the field to its type when the type of the field is a class that is modeled in the ontology.
  - *Modifiers*. It contains the field modifiers, that indicate if the field is final, private, protected, public, static, transient or volatile.
  - *IsArray*. It is a boolean that indicates if the field cardinality is more than 1, i.e., if the field is a set.

- *MethodClass*: Methods are modeled with this class. The set of properties that defines this class is the following.
  - *Name*. This is a string to express the name of the method.
  - *Signature*. This property contains the complete signature of the method. The parts of the signature are also modeled in another properties, as we can see here.
  - *Modifiers*. It is a list with the modifiers of the method. In this case, the modifiers indicate if the method is abstract, final, native, private, protected, public, static, strictfp or synchronized.
  - *Parameters*. This is a list with the types of the metdhod parameters.
  - *ReturnType*. This is a string with the return type of the method.
  - *LocalVariables*. This property relates the method to its local variables, that are modeled as instances of the LocalVariableClass.
  - *InvokedMethods*. It is a set with the names names of the methods that are invoked by the method that is being modeled.
  - *InvokedMethodsInOntology*. This property is very close to the previous one. It is a set of MethodClass instances that represent the invoked methods that are represented into the ontology. The set of invoked methods of the *InvokedMethod* property contains more elements than the set of the *InvokedMethodInOntology* property when the method invokes methods that are not modeled in the ontology. It happens frequently.
  - *Exceptions*. It is a set of strings with the names of the exceptions that the method throws.
  - *Code*. This is a string with the byte code (object code) of the method.

- *LocalVariableClass*: This class is used to represent information about the local variables of the methods. The properties of this class are the same as those of Field Class, except the *Modifiers* property, as we can see.
  - *Name*. It represents the name of the local variable in a string.
  - *Type*. It indicates the type of the local variable in a string. If the local variable is an array, this property do not include the [] symbols.
  - *TypeInOntology*. When the type of the field is a class, if it is represented in the ontology, this property is the instance that represents that class.

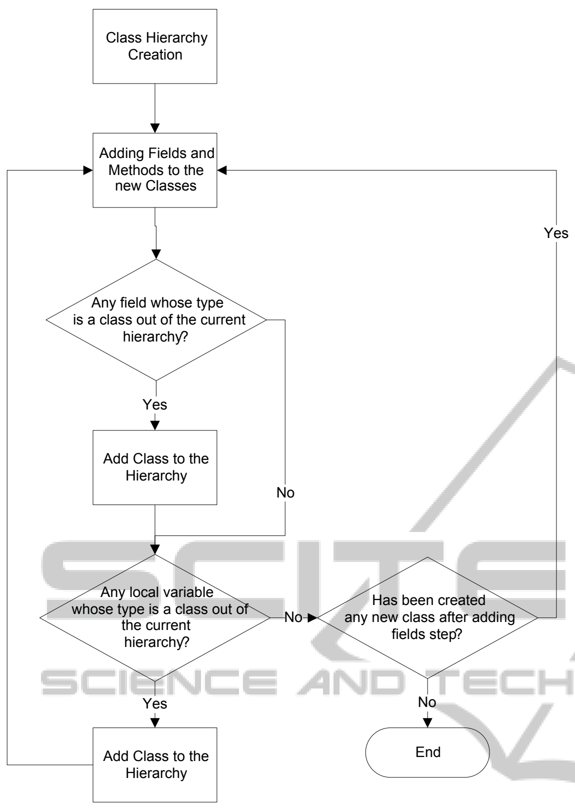

Class Hierarchy Creation
Adding Fields and Methods to the new Classes
Any field whose type is a class out of the current hierarchy?
Yes
No
Add Class to the Hierarchy
Any local variable whose type is a class out of the current hierarchy?
No
Has been created any new class after adding fields step?
Yes
No
Yes
Add Class to the Hierarchy
End

Figure 6: Structural and behavior model extraction flow diagram.

– *IsArray*. It is a boolean that indicates if the field cardinality is more than 1, i.e., if the field is a set.

As we can see, by means of the previous OWL containers, our approach obtains an ontology that represents more semantic information about classes. Each class created in the ontology include all the information obtained by means of the structural model extraction process, plus additional structural and behavior information.

3.2.2 Inserting Semantic Information into the Container

Now, we will explain the process to obtain all that information. It is similar to the structural model extraction process, although it is more complex.

The process, shown in figure 6, is as follows: Initially, we add to the ontology instances of the ClassClass corresponding to the Java classes of the hierarchy embedded in the input library. In this first step, we add all the information about each class, but the fields and the methods. After that, each class of the hierarchy is deeply analyzed focusing on its fields and methods. This step usually entails the creation of new ClassClass instances. It can be caused by two reasons: because there are types of the added fields that are classes that are not yet represented in the OWL ontology or because there are types of the local variables of the added methods that are classes that are not yet represented in the ontology. This process is recursively executed until no new class is added.

At the end of the process, the amount of classes represented in the OWL ontology is usually larger than the amount of classes obtained by the approach explained in the previous subsection, because we now add to the ontology information regarding the classes that appear in the Java methods.

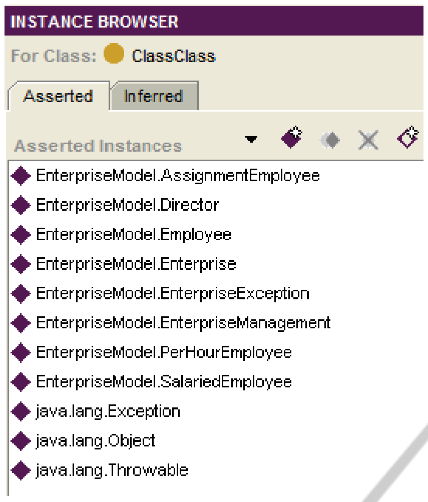
Let us formalize the before introduced *Comprehensive Model Extraction* process.

Definition 3. *Let C be the set of containers shown in figure 5 and represented in an ontology O, we define the* Comprehensive Model Extraction Process *as the transformation of a Java library L in a set of instances of the O ontology classes that satisfies:*

- *For each class JC_i in the library L, there exists a corresponding instance CI_i of the ClassClass in the ontology O.*
- *For each superclass JC_j of a class JC_i, there exists a corresponding instance CI_j of the ClassClass in O, where CI_j represents the superclass of CI_i.*
- *For each field f_j of a class JC_i, there exists a corresponding instance FI_j of the FieldClass in O.*
- *For each method m_k of a class JC_i, there exists a corresponding instance MI_k of the MethodClass in O.*
- *For each local variable lv_l of a method m_k, there exists a corresponding instance LVI_l of the LocalVariableClass in O.*
- *For each field f_j or local variable lv_l, if its range is a class JC_m , there exists a corresponding instance CI_m of the ClassClass in O.*

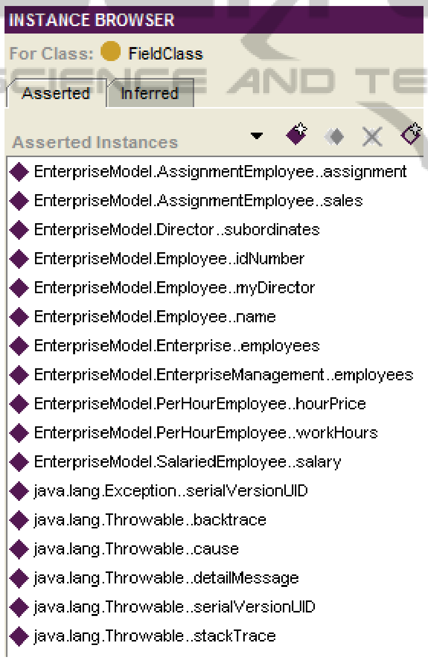
When we use our example library as input of *Jar2Ontology* and we use this comprehensive approach, we obtain an ontology with 4 classes (ClassClass, FieldClass, MethodClass and LocalVariableClass). Their properties are those that we shown in figure 5. This ontology has much information: For each class that we model, we have one instance of the *ClassClass*, some *FielClass* instances related to its fields, some *MethodClass* instances related to its methods, and for each one of its method, some *LocalVariableClass* instances related to its local variables.

Figure 7 shows the instances of the *ClassClass* at the beginning of this process. We can see that the classes are the same that the obtained at the beginning of the structural model extraction process (in figure



EnterpriseModel.AssignmentEmployee
EnterpriseModel.Director
EnterpriseModel.Employee
EnterpriseModel.Enterprise
EnterpriseModel.EnterpriseException
EnterpriseModel.EnterpriseManagement
EnterpriseModel.PerHourEmployee
EnterpriseModel.SalariedEmployee
java.lang.Exception
java.lang.Object
java.lang.Throwable

Figure 7: Classes in the metadata ontology that models the example library class hierarchy.



EnterpriseModel.AssignmentEmployee..assignment
EnterpriseModel.AssignmentEmployee..sales
EnterpriseModel.Director..subordinates
EnterpriseModel.Employee..idNumber
EnterpriseModel.Employee..myDirector
EnterpriseModel.Employee..name
EnterpriseModel.Enterprise..employees
EnterpriseModel.EnterpriseManagement..employees
EnterpriseModel.PerHourEmployee..hourPrice
EnterpriseModel.PerHourEmployee..workHours
EnterpriseModel.SalariedEmployee..salary
java.lang.Exception..serialVersionUID
java.lang.Throwable..backtrace
java.lang.Throwable..cause
java.lang.Throwable..detailMessage
java.lang.Throwable..serialVersionUID
java.lang.Throwable..stackTrace

Figure 8: Properties of the classes in the metadata ontology that models the example library.

2). Figure 8 contains the instances of the *FieldClass* related to the classes shown in figure 7. We can realize that these fields correspond to the OWL properties that we obtained using the structural model extraction process (in figure 3). We do not show the instances of the *MethodClass* neither of the *LocalVariableClass*, because there are too many.

At the beginning of the comprehensive model extraction process, the ontology only has 11 instances of the *ClassClass*, 17 instances of the *FieldClass*, 91 of the *MethodClass* and 188 of the *LocalVariableClass*. When the extraction process has finished, there are 129 *ClassClass* instances, and the number of the other classes instances increases accordingly.

4 THE OBTAINED MODELS

As we have seen in section 3, information extraction process can obtain as a result a big model that represents much more classes than those that are directly embedded in the library. However, this exhaustive recursion is not always necessary (for example, when we want to use these ontologies as input of a matching process). For this reason, we distinguish three kinds of models with different recursion degrees, depending on the desired comprehension of the resulting model:

- *Lite Model*. This model only captures the classes that directly appear in the jar file as well as their superclasses. No additional classes are included in the model. Figures 2, 7 and 8 correspond to this kind of model, for our example.
- *Full Model*. This model contains all the classes that appear in the jar file together with all the classes found during the methods/fields analysis. This type of model stores the highest information amount that can be extracted from the jar file.
- *Pruned Model*. This type of model is a compromise between full and lite models. It contains information regarding to the classes from the jar file class hierarchy and the classes added in the first iteration of the methods/fields analysis. Figure 9 shows the data ontology obtained as pruned model. Classes in darker background are those classes that have been added in the fist iteration of the process.

Next section will show the results of the execution of *Jar2Ontology* with different software libraries. We will see the different amount of classes, fields, methods and local variables represented in each kind of these models.

5 IMPLEMENTATION AND EXPERIMENTATION ISSUES

The Protégé Ontology Editor and Knowledge Acquisition System has been developed in Java, is extensible and provides a programming frame by means of plug-ins. Furthermore, Protégé is backed by a large community of active users and developers.

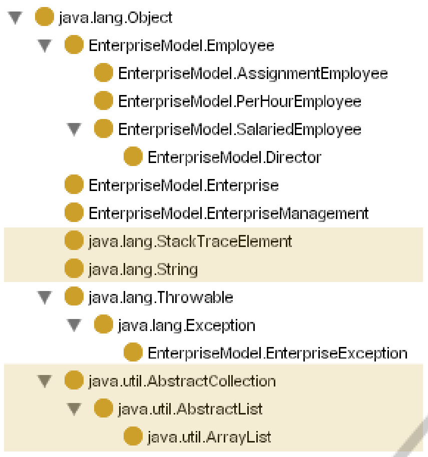

java.lang.Object
EnterpriseModel.Employee
EnterpriseModel.AssignmentEmployee
EnterpriseModel.PerHourEmployee
EnterpriseModel.SalariedEmployee
EnterpriseModel.Director
EnterpriseModel.Enterprise
EnterpriseModel.EnterpriseManagement
java.lang.StackTraceElement
java.lang.String
java.lang.Throwable
java.lang.Exception
EnterpriseModel.EnterpriseException
java.util.AbstractCollection
java.util.AbstractList
java.util.ArrayList

Figure 9: Classes in the data ontology obtained as pruned model of the example library.

We have developed *JarToOntology*, a set of Protégé plug-ins, to implement the before described approaches. As Protégé is developed in Java, we have used some libraries apart from the standard Java ones. Core Protégé API[3] and Protégé-OWL API[4] have been used to develop the plug-ins on top of Protégé and to create and to manipulate OWL ontologies. Additionally, in order to directly deal with Java object code, we have used BCEL API[5](Dahm and Van Zyl, 2006; Dahm, 2001) and Java reflection[6].

JarToOntology has been tested by means of many libraries. Some of them are the following:

- *EnterpriseModel.jar*. It is a little library created with the aim to illustrate the implemented approaches. In figure 1 we can see the class hierarchy that the jar file contains, together with their superclasses.
- *flickrapi.jar*. It is the library of the flickr Java API[7]. This library contains 110 class files.
- *htmlcleaner2_1.jar*. HtmlCleaner is an open-source HTML parser written in Java[8]. This library contains 40 class files.
- *MozillaParser.jar*. It is a Java package that enables you to parse html pages into a Java Document object[9]. This library contains 14 class files.

We want to remind that JarToOntology input is not an UML diagram, neither the source code that is the result of its implementation. The input of JarToOntology is the jar file that contains Java object code files corresponding to the implemented classes of the library.

[3]http://protege.stanford.edu/protege/3.4/docs/api/core
[4]http://protege.stanford.edu/protege/3.4/docs/api/owl
[5]http://jakarta.apache.org/bcel
[6]http://java.sun.com/docs/books/tutorial/reflect
[7]http://www.flickr.com/services/api
[8]http://htmlcleaner.sourceforge.net/
[9]http://mozillaparser.sourceforge.net/

Table 1: Number of classes obtained in the data ontology, i.e., when the structural model extraction approach is used.

Library	Lite Model	Full Model	Pruned Model
EnterpriseModel	11	120	16
flickrapi	121	241	143
HtmlCleaner	47	159	60
MozillaParser	18	143	23

In table 1 we can see the number of classes of the ontologies obtained by means of our extraction tool when we use the structural model extraction approach. The number of classes of the lite model is always a bit higher than the number of class files of the library, because the superclasses of the library classes are included. Thus, we can note that although there are 8 classes in EnterpriseModel library (classes with white background in figure 1), the lite model includes 3 classes more, corresponding to the superclasses of the library classes (java.lang.Object, java.lang.Throwable and java.lang.Exception).

We can see that, as we said when we explained the obtained models in section 4, the full model has much more classes than the initial class hierarchy as well as the pruned model is a compromise solution between the other two.

In table 2 we can see the number of instances of each of the four classes created in the ontology when we apply our comprehensive model extraction approach using the above mentioned example libraries. For the sake of space, we do not show here the obtained ontology instances, but a summary of the number of them.

We obtain the same conclusions from this table than the conclusions obtained before. Furthermore, we can see that the growth of the number of classes represented in the ontologies is higher, because we add classes obtained during the deep analysis of the behavioral component.

6 CONCLUSIONS

In this work we have presented a novel approach for the automatic extraction of semantic knowledge from Java object code. The approach takes into account both the structural and the behavioral knowledge, going one step beyond the traditional structural conceptual modeling.

In addition to this, the proposed approach has been

Table 2: Number of classes (C), fields (F), methods (M) and local variables (LV) obtained in the comprehensive ontology, i.e., when the comprehensive model extraction approach is used.

EnterpriseModel				
Number of...	C	F	M	LV
Lite Model	11	17	91	188
Full Model	129	599	1440	188
Pruned Model	25	65	334	188

flickrapi				
Number of...	C	F	M	LV
Lite Ontology	121	694	236	217
Full Ontology	297	1696	2447	217
Pruned Ontology	163	941	990	217

HtmlCleaner				
Number of...	C	F	M	LV
Lite Ontology	47	177	532	681
Full Ontology	212	976	2609	681
Pruned Ontology	60	217	837	681

MozillaParser				
Number of...	C	F	M	LV
Lite Ontology	18	51	58	76
Full Ontology	144	693	1513	76
Pruned Ontology	23	82	218	76

implemented, obtaining as a result the *Jar2Ontology* tool. It is a set of Protégé plugins that generate OWL models from Java libraries object code. We have tested it with toy examples, as well as with real world libraries.

Our research has been oriented to the data integration context. Nevertheless, semantic knowledge extraction from Java libraries can be useful for many more applications, as for example, automatic generation of code documentation, or reverse engineering.

ACKNOWLEDGEMENTS

Part of the work reported in this paper was supported by the Spain's Science and Innovation Ministry (Ministerio de Ciencia e Innovación) under grant TIN2006-15041-C04-01.

Part of the research reported in this paper is supported by the Andalusian Government (Junta de Andalucía, Consejería de Economía, Innovación y Ciencia) under project P07-TIC-03175 ”Representación y Manipulación de Objetos Imperfectos en Problemas de Integración de Datos: Una Aplicación a los Almacenes de Objetos de Aprendizaje”.

REFERENCES

Amey, P. (2002). Closing the Loop: The Influence of Code Analysis on Design. *Reliable Software Technologies - Ada-Europe 2002. LNCS*, 2361:151–162.

Buitelaar, P., Cimiano, P., Frank, A., Hartung, M., and Raioppa, S. (2008). Ontology-based Information Extraction and Integration from Heterogeneous Data Sources. *International Journal of Human-Computer Studies*, 66(11):759–788.

Curino, C., Orsi, G., Panigati, E., and Tanca, L. (2009). Accessing and Documenting Relational Databases through OWL Ontologies. *Flexible Query Answering Systems (FQAS'09). LNAI*, 5822:431–442.

Dahm, M. (2001). Byte Code Engineering with the BCEL API. Technical report, Freie Universität Berlin. Institut für Informatik.

Dahm, M. and Van Zyl, J. (2006). Byte Code Engineering Library.

Gómez-Pérez, A., Fernández-López, M., and Corcho, O. (2004). *Ontological Engineering: With Examples from the Areas of Knowledge Management, e-Commerce and the Semantic Web*. Springer.

Herbold, S., Grabowski, J., and Neukirchen, H. (2009). Automated Refactoring Suggestions Using the Results of Code Analysis Tools. *First International Conference in System Testing and Validation Lifecycle*, pages 104–109.

Hong, T., Hua, C., Gang, Z., Qiang, L., and Jinjin, Z. (2009). The Vulnerability Analysis Framework for Java Bytecode. *15th International Conference on Parallel and Distributed Systems*, pages 896–901.

Jackson, D. and Waingold, A. (1999). Lightweight extraction of object models from bytecode. *Proceedings of the 21st international conference on Software engineering*, pages 194–202.

Jakobac, V., Egyed, A., and Medvidovic, N. (2005). Improving System Understanding via Interactive, Tailorable, Source Code Analysis. *Fundamental Approaches to Software Engineering (FASE). LCNS*, 3442:253–268.

Kalfoglou, Y. and Schorlemmer, M. (2003). Ontology Mapping: the State of the Art. *The Knowledge Engineering Review Journal (KER)*, 18(1):1–31.

Kawrykow, D. and Robillard, P. (2009). Improving API Usage trhough Automatic Detection of Redundant Code. *IEEE/ACM International Conference on Automated Software Engineering*, pages 111–122.

Letarte, D. and Merlo, E. (2009). Extraction of Interprocedural Simple Role Privilege Models from PHP Code. *16th Working Conference on Reverse Engineering*, pages 187–191.

Martino, B., Mazzocca, N., Saggese, G., and Strollo, A. (2002). A Technique for FPGA Synthesis Driven by Automatic Source Code Analysis and Transformations. *International Conference on Field Programmable Logic and Applications (FLP). LCNS*, 2438:47–58.

Myroshnichenko, I. and Murphy, M. (2009). Mapping ER Schemas to OWL Ontologies. *IEEE International Conference on Semantic Computing*, pages 324–329.

Rahm, E. and Bernstein, P. (2001). A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10:334–350.

Sáez-Árcija, C., Marín, N., and Vila, M. (2009). A Lazy-Typing Based Architecture for a Data Integration System. *Workshop on New Trends on Intelligent Systems and Soft Computing*, 2:1–18.

Shvaiko, P. and Euzenat, J. (2005). A Survey of Schema-based Matching Approaches. *Journal on Data Semantics(JoDS)*.

Spinellis, D. (2010). CScout: A refactoring browser for C. *Science of Computer Programming*, 75:216–231.

Spoto, F., Mesnard, F., and Payet, E. (2010). A Termination Analyzer for Java Bytecode Based on Path-Length. *ACM Transactions on Programming Languages and Systems*, 32(3):8:1–8:70.

Thiam, M., Bennacer, N., Pernelle, N., and Lô, M. (2009). Incremental Ontology-Based Extraction and Alignment in Semi-structured Documents. *20nd International Conference on Database and ExperSystems Applications (DEXA)*. *LCNS*, 5690:611–618.

Wache, H., Vgele, T., Visser, U., Stuckenschmidt, H., Schuster, G., Neumann, H., and Hbner, S. (2001). Ontology-Based Integration of Information - A Survey of Existing Approaches. *Workshop on Ontologies and Information Sharing at the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 108–117.

Wimalasuriya, D. and Dou, D. (2010). Ontology-based Information Extraction: An Introduction and a Survey of Current Approaches. *Journal of Information Science*, 36(3):306–323.

Wong, W. and Gokhale, S. (2005). Static and Dynamic Distance Metrics for Feature-based Code Analysis. *The Journal of Systems and Software*, 74:283–295.