Semantic and Algorithmic Recognition Support to Porting Software Applications to Cloud

Beniamino Di Martino and Giuseppina Cretella

Second University of Naples - Dept. of Industrial and Information Engineering, Italy
beniamino.dimartino@unina.it, giuseppina.cretella@gmail.com

Abstract. This paper presents a methodology, a technique and an ongoing implementation, aimed at supporting software porting (i.e. to adapt the software to be used in different execution environments), from object oriented domain towards Cloud Computing. The technique is based on semantic representation of Cloud Application Programming Interfaces, and on automated algorithmic concept recognition in source code, integrated by structural based matchmaking techniques. In particular the following techniques are composed and integrated: automatic recognition of the algorithms and algorithmic concepts implemented in the source code and the calls to libraries and APIs performing actions and functionalities relevant to the target environment; comparison through matchmaking of the recognized concepts and APIs with those present in the functional ontology which describes the target API; mapping of the source code excerpts and the source calls to APIs to the target API calls and elements.

Keywords: Semantic Discovery, Cloud APIs, Cloud Resources, Algorithmic Recognition.

1 Introduction

Software porting over different domains is an important issue, mainly in the recent years, where porting operation are needed not only for the reingeenerization of old applications, but mostly to port applications over different technologies, like Cloud Computing.

In the last years Cloud Computing has emerged as a prominent model to provide online access to computational resources, thanks to its characteristics of scalability, elasticity, reduced cost, easiness of use, simple maintenance. The concept of cloud computing is clearly expressed by the NIST definition: "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.[...]"

One of the issues related to the adoption of the cloud computing paradigm is the lackness of a common programming model and open standard interfaces. Many cloud providers offer different cloud services, but each of their offerings is

A. Moschitti and B. Plank (Eds.): EternalS 2013, CCIS 379, pp. 1–15, 2013.

based on proprietary Application Programming Interfaces (APIs). This situation complicates the already challenging task of building up applications from interoperable services provided by different cloud providers; and the specific APIs, kind of resources and services provided by a given cloud provider make future migrations costly and difficult (Cloud Vendor Lock in). Portability of code towards a Cloud providers' environment and among Cloud providers' environments is a severe issue today.

The application of semantic techniques to reverse engineering can enable the automation or automated support of activities such as porting. Understanding the functionalities exposed by software artifacts represents an essential support for a large range of software reengineering activities such as maintenance, debugging, reuse, modernization, and porting.

This paper presents a methodology, a technique and the ongoing implementation, aimed at supporting software porting towards Cloud Computing environments. The technique is based on semantic representation of Cloud Application Programming Interfaces, and on automated Algorithmic Concept Recognition in source code, integrated by structural based matchmaking techniques.

We combine techniques such as graph based source code representation, first-order-logic rules for algorithmic recognition and semantic based algorithmic and codes (APIs) knowledge representation.

Objectives of the described work relate porting of applications towards Cloud through the extraction of knowledge from the Application Programming Interfaces, and the association of semantic description identifying the concepts they implement. The described technique anyway can target a wide range of applications, from code reuse to advanced code searching.

The paper is organized as follows. In Section 2 we will present an overview of works related to reverse engineering solution for program comprehension, with particular attention to the works that use ontologies and graph based representations for code and software artifacts. In Section 3 we will present the porting methodology, based on an automatic analysis and representation of code at higher level of abstraction than the syntactical and structural one. Section 4 presents the main components of the architecture and the workflow of the methodology presented in the previous section. Conclusions and future works are drawn in Section 5.

2 Background and Related Work

Reverse engineering is the process of system analysis to identify the components and their interrelationships and create representations at a higher level of abstraction. Therefore it's an activity that allows getting specific information about the design of a system from its code, through extraction and abstraction of system information. Reverse engineering may require a thorough understanding of systems (white-box approach) or may be limited to only the external interfaces (software reengineering of black-box). The white-box approach supports reverse engineering with a deep understanding of individual modules and

conversion activities. The black-box approach is limited to the study of the external interfaces of systems and activities of encapsulation (wrapping). In reverse engineering various software artifacts can be analysed. A software artifact is any tangible product created during software development. Some artifacts (e.g. use cases, class diagrams and other UML models, requirements and design documents) are useful to describe functions, architecture and software design. Others involve the development process itself, such as project plans, business cases and risk assignment. The code, the released executable and the associated documentation are artifact too. There are two different directions in program comprehension research: the first strives for understanding the cognitive processes that programmers use when they understand programs and use empirical information to produce a variety of theories that provide explanations of how programmers understand programs, and can provide advice on how program comprehension tools and methods may be improved; the second aims at developing semi-automated tool support to improve program comprehension. Some of the more prominent approaches include textual, lexical and syntactic analysis (focus on the source code and its representations), execution and testing (based on observing program behaviour, including actual execution and inspection walkthroughs), graphic methods (including earlier approaches such as graphing the control flow of the program, the data flow of the program, and producing the program dependence graphs), domain knowledge based analysis (focus on recovering the domain semantics of programs by combining domain knowledge representation and source code analysis). The problem of associating concepts to code is not a problem amenable to be solved in its general formulation because the human-oriented concepts are inherently ambiguous, and their recognition is based on a priori knowledge of a particular domain. The problem can instead be solved under specific constraints and limitations, such as limiting the range of recognition at the algorithmic level [10]. A different approach is to understand the code through the analysis of the documentation associated with it, with text mining techniques that capture the concepts described in the documentation and connect them with the appropriate portions of code that implement [2]. The approach in [2] represents various software artifacts, including source code and documents as formal ontologies. The ontological reasoning services allow programmers not only to reason about properties of the software systems, but also to actively acquire and construct new concepts based on their current understanding; and introducing an ontology-based comprehension model and a supporting comprehension methodologies that characterize program comprehension as an iterative process of concept recognition and relationship discovery. Application developers often reuse code already developed for several reasons. The most common situation is accessing libraries of reusable components or putting them in the application framework. Unfortunately, many libraries and frameworks are not very intuitive to use, and libraries often lack a comprehensive API documentation and code examples that illustrate a particular feature or functionality. It is therefore useful to provide advanced tools for code search and suggestion. This issue is addressed generally representing code in a form

suitable to perform computation and reasoning, as shown in [3], where the code is represented through an ontology to perform query that can be used to provide suggestion for library usage. The ontology formalism is used to represent software assets by building a knowledge base that is automatically populated with instances representing source-code artifacts. This approach uses this knowledge base to identify and retrieve relevant code snippets. To add formal semantic annotations, it's necessary to have a formal knowledge description processable and to an appropriate level of abstraction. This is not always available, so it would be useful to have tools that can extract this knowledge automatically or semi automatically from the sources of information. One of the major structured sources of knowledge are the public interfaces of libraries of a specific domain. However, a single API contains only a view of the particular domain and it's not generally sufficient to obtain a complete model of the domain. In addition, the APIs contain a significant amount of noise due to implementation details that combine with the representation of knowledge in the domain interfaces. In order to overcome these problems it's possible to base the extraction of domain knowledge on multiple APIs that cover the same domain. This issue is addressed in [4], where it is proposed an approach to extract domain knowledge capturing the commonalities among multiple API; the extraction is based on the frequency matching of given elements. In [5] an approach to learning domain ontologies from multiple sources associated with the software project, i.e., software code, user guide, and discussion forums is proposed. This technique do not simply deal with these different types of sources, but it goes one step further and exploits the redundancy of information to obtain better results. In [6] and [7] it is proposed a method for domain ontology building by extracting ontological knowledge from UML models of existing systems, by comparing the UML model elements with the OWL ones and derive transformation rules between the corresponding model elements. The aim of the process is to reduce the cost and time for building domain ontologies with the reuse of existing UML models.

3 The Methodology

The porting methodology we are presenting is based on an automatic analysis and representation of code at higher level of abstraction than the syntactical and structural one: namely the algorithmic or functional level.

The methodology assumes that the porting procedure can be realized by restructuring the code to be ported to a target environment (e.g. Cloud) with suitable calls to functionalities of a given *target* Application Programming Interface, implementing all functionalities needed to deploy and run the code on the environment.

The target API is assumed to be (manually) semantically described at the algorithmic and functional level, and annotated, with concepts described by means of an OWL based *functional ontology*.

It is also assumed that the code to be ported includes implementations of algorithms and functionalities included in the functional ontology, and calls or libraries and APIs, which do not (necessarily) correspond to the target API.

The main idea underlying the methodology is the following: to automatically recognize the algorithms and algorithmic concepts implemented in the source code and the calls to libraries and APIs performing actions and functionalities relevant to the target environment, compare through matchmaking the recognized concepts and APIs with those present in the functional ontology which describes the target API and semantically annotates its elements and calls, and by means of this matching, eventually map the source code excerpts and the source calls to APIs to the target API calls and elements.

The methodology represents the following components in a uniform, graph based, representation, the *knowledge base*:

- the *Target API*;
- the *Grounding Ontology*;
- the *Functional Ontology*;
- the source code *Call Graph*;
- the source code *API Graph*;
- the *Candidate API Ontology Graph*;
- the source code *Program Dependence Graph*;
- the source code *Abstract Program Representation Graph*.

The *Target API* is the Application Programming Interface towards which the porting activity is addressed. Examples are the APIs exposed by the Cloud providers, offering Cloud resources and services at Infrastructure, Platform and Application levels. The methodology assumes that this API is (manually) semantically annotated with concepts of the Functional Ontology.

The *Grounding Ontology* is a syntactical representation on an API. It represents a base to build semantic annotations of the *grounding concepts* (the syntactical elements of the API) with the Functional Ontology concepts.

The *Functional Ontology* represents a collection of concepts from the domain of Programming Algorithms and Data Structures [8], general purpose functionalities offered by libraries related to a given domain, such as Cloud Computing, and Design Patterns [9].

The *Call Graph* represents the calling relationships between the source codes procedures.

The *Candidate API Ontology* is an ontology automatically derived from an API by applying a set of graph transformation patterns, as for instance illustrated in [1].

The *Program Dependence Graph* is a structural level representation of a program, which represents dependence relationships (control and data) among the program statements. In our methodology we use a PDG representation slightly augmented with syntactical control and data dependence information.

The *Abstract Program Representation* represents the recognized algorithmic concepts in the source code and their structure, the relationships among them, and groundings within the source code.

The above defined knowledge base components can be grouped in three different levels for both the source and the target porting environments, as sketched in (Fig. 1). In the first level, the *grounding level*, there are the basic information extracted by parsing the source code to port, which are the *Source API Graph* (in the scenario we have an API to map over another API), the *Call Graph* and the *Program Dependence Graph* (in the scenario we have a source code to port) in the source environment and the *Target API Graph* in the target environment. In the *Functional and Algorithmic Concept Level* we have on the source side the *Abstract Program Representation Graph* and the *Candidate API Ontology Graph* witch represent the high level information derived respectively from the Program Dependence Graph and the Source API Graph. On the source side at functional and algorithm level we have the graph representation of the Functional Ontology. In the *Application Level* we have concepts related to the application domain which can be linked with functional and algorithmic concepts.

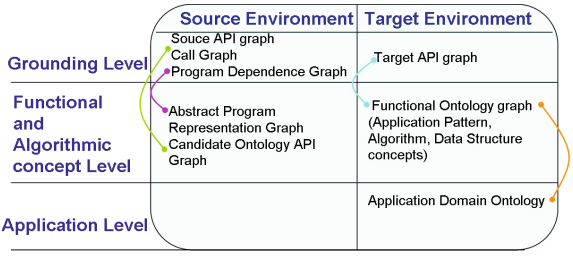

Fig. 1. Knowledge base levels

The source code is represented using two graph structures: the *Program Dependence Graph* suitable for algorithm recognition, discussed in Section 4.1, and another based on the *Call Graph* with one node for each call in the source code.

Given these representations, the methodology tries to find an equivalence of source code components and target API components, through graphical matchmaking of their graph based semantic representations. These are the Abstract Program representation and the Candidate API Ontology of the source code, and the functional ontology with which the target API components are represented and annotated.

The same approach can be used to find equivalent implementations of the same functionalities among different API. If the two APIs (source and target) are both semantically described and annotated with the functional ontology the equivalence is quite straightforward to find because the two annotations will refer to the same functional ontology concept; while if one of the two APIs is not annotated, we can produce the Candidate API ontology Graph and match it with the functional ontology, used to annotate the other API.

4 Design of the Architecture and Ongoing Implementation of the Porting Support Procedure

The architecture implementing the methodology described in the previous section, is illustrated in Fig. 2 with the workflow and interactions among the components, while Fig. 3 illustrates the workflow for the API annotation process.

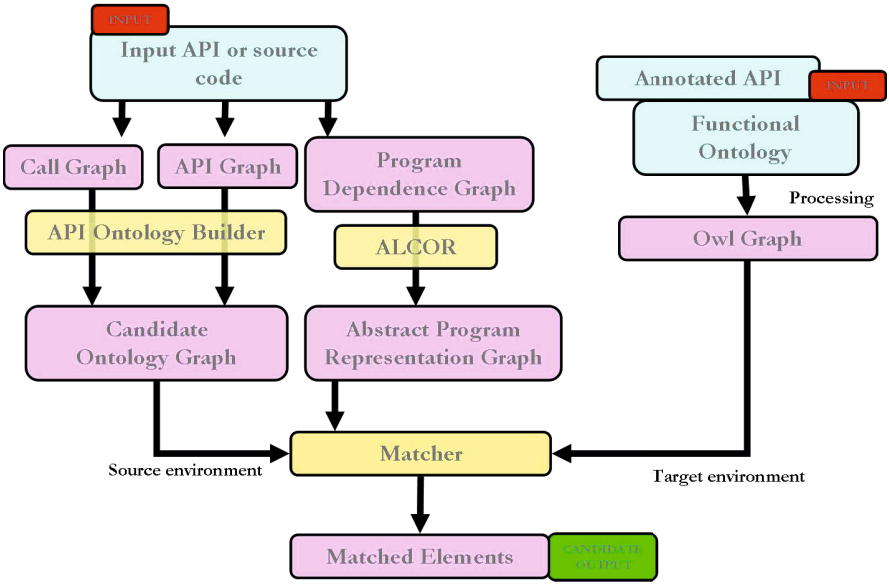

Fig. 2. Workflow of the porting procedure

The architecture is composed of the following four modules.

The *ALCOR (ALgorithmic COncept Recognizer)* module [10] recognizes algorithmic concepts in the code, producing the Abstract Program Representation.

The *API Ontology Builder* extracts the graph by parsing source code and from the graph representing the code produces the *API grounding ontology* which represent the base where to ground the semantic annotations. This ontology enables the annotation of the API elements in a simple way, by adding relation between grounding elements and high level abstraction concept. Additionally the API Ontology Builder produces the *Candidate API ontology* graph applying graph transformation patterns defined for the specific programming language or model.

The *Schema Matcher* module [14] accepts two graph based representations and performs the matching between the two graphs, by applying several algorithm including structural-based algorithms and syntactical ones.

The *Annotator* allows the user to semantically annotate the target API with concepts from the functional ontology.

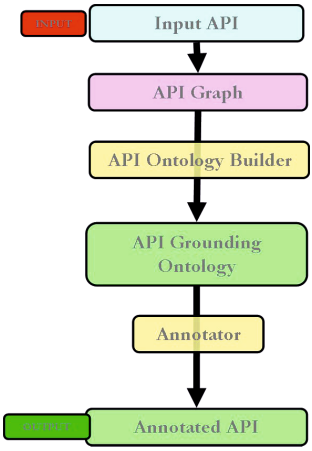

Fig. 3. API Semantic Annotation workflow

As illustrated in Fig. 2, the inputs to the procedure are, on one side the source code to be ported, together with APIs utilized, or directly the APIs; on the other side the target API, semantically described and annotated with the Functional Ontology (expressed in OWL language). The output of the procedure is a mapping between the source API components and source code excerpts, and target API components which are equivalent (functional equivalence) to the source elements and code, and which represent the candidates to replace the source elements during the porting activity.

The input components (source code and APIs) are statically analysed with use of a static code analyser, and the components of uniform the knowledge representation, the Program Dependence Graph, the Call Graph and the API Graph, are produced. On the other hand an OWL parser produces the OWL graph representation of the functional ontology. The ALCOR module detects, from the Program Dependence Graph, the algorithmic concepts implemented within the source code, and produces the Abstract Program Representation, represented as a graph in the uniform knowledge base, which represents the recognized concepts and their hierarchical and control/data dependences, and their grounding (implementation) within the source code. Details on the recognition procedure and the concepts representation are provided in sec. 4.1.

The API Ontology Builder module, on the other hand, analyzes the APIs used by the source code, represented by the API graph, and produces the *Candidate API ontology* graph, by applying graph transformation rule patterns defined for the specific programming language or model. This Ontology represents the semantics of the components of the API under analysis, and their semantic relationships. Details on the transformation rules and on the module implementation are provided in sec. 4.3.

Once produced the uniform Knowledge base with the components described, the Matcher performs the matching between the source and target elements,

producing a set of mapping elements specifying the matching elements together with a similarity value between 0 (strong dissimilarity) and 1 (identity) indicating the plausibility of their correspondence.

The implementation of the porting support procedure is ongoing work, and it is mainly consisting of (a) the development of the API Ontology builder and API annotator; (b) the development of the source code analysis front end (starting from a previous implementation realized within the ROSE compiler construction toolkit; (c) the integration of the already developed modules ALCOR and Schema Matcher; (d) the implementation of a Graphical User Interface, providing the user with the matching results in a form graphically relating the source code excerpts with suggested target API elements, in order to perform a suitable and effective support to the porting activity.

In the following sections we describe in more details the working principles and the ongoing implementation and integration of the Algorithmic Concepts Recognition module (sec. 4.1), of the Schema Matcher module (sec. 4.2) and of the API Ontology Builder module (sec. 4.3).

4.1 Algorithmic Concepts Recognition

The *Algorithmic Concept Recognizer*, previously designed and developed [10,11] implements a technique for automated algorithmic concepts recognition in source code [12], where the definition of *parallelizable algorithmic concept* and the technique to describe and detect the algorithmic concepts by using an attributed grammar were presented, and which is briefly resumed here.

The Algorithmic Concepts Recognition is a Program Comprehension technique to recognize in source code the instances of known algorithms. The recognition strategy is based on a hierarchical parsing of algorithmic concepts. Starting from an intermediate representation of code, *Basic Concepts* are recognized first. Subsequently they become components of *Structured Concepts* in a hierarchical and / or recursive way. This abstraction process, can be modeled as a hierarchical parsing, by using *Concept Recognition Rules* that act on a description of concept instances found in the code.

Basic concepts. The building blocks of the hierarchical abstraction process are the *Basic Concepts*. They are chosen among the elements of the intermediate code representation at the structural level. A slightly modified version of the Program Dependence Graph is used: it is augmented with syntactical information (e.g. trees structures representing expressions for each statement node), control and data dependence information (e.g. control branches, data dependence level, variables, ... are added).

Concept Recognition Rules. The *Concept Recognition Rules* are the production rules of the parsing: they describe the set of characteristics and properties to permit the identification of an algorithmic concept instance in the code.

Each recognition rule related to an algorithmic concept specifies how subconcepts, formed by set of statements and variables linked by a functionality,

are related and organized within a specific abstract control structure. Each rule describes the concept in a recursive way by using:

- A composition hierarchy: this is specified by the set of sub-concepts directly composing the concept and their own composition hierarchies.
- A set of constraints and conditions to be satisfied by the composing sub-concepts, and all the relationships among them and with the sub-concepts of the hierarchy.

A formalism for the specification of the recognition rules is given by *Attributed Grammars* [13] for their expressiveness regarding the specification of the hierarchy, the constraints and relationships, as is well-known for the specification of programming languages.

A production rule of the grammar specifies: a set *sub-concept* of terminal and non-terminal symbols which represent the set of sub-concepts forming the concept represented by the lhs symbol *concept*.

The set *condition* represent the relationships and constraints that must be fulfilled by the sub-concepts forming the concept, in order to be recognized as a valid instance.

The set *AttributionRule* of the production assigns values to the attributes of the recognized concept utilizing the values of attributes of the composing sub-concepts.

The syntax of a production rule is as follows:

Rule =
  rule *concept* →
    composition
      { *subconcept* }
    condition
      [**local** LocalAttributes]
      { Condition }
    attribution
      { AttributionRule }

LocalAttributes =
  attribute : Type { *attribute* : Type }

concept ∈ N
subconcept ∈ $N \cup T$
attribute ∈ A
Condition ∈ C
AttributionRule ∈ R

The Recognition Process. The PDG information, together with syntactical information can be produced as a set of Prolog facts representing the *Abstract Program Representation*. The hierarchical parsing process that do the recognition, is performed by an *Inferential Engine* that applies the production rules of

the parsing (expressed as Prolog clauses) to the set of terminals, non-terminals and relationships of the *Abstract Program Representation*.

An overall *Abstract Program Representation* is generated during the recognition process. An example is illustrated in Fig. 4.

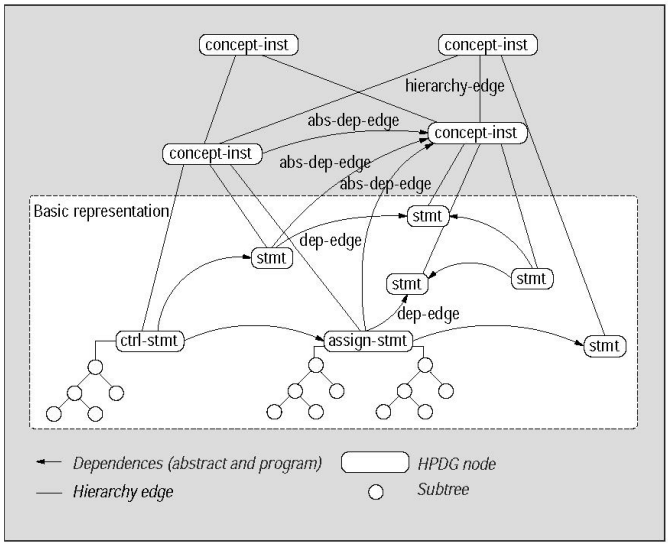

Fig. 4. Abstract Program Representation

It has the structure of a *Hierarchical PDG* (HPDG), reflecting the hierarchical strategy of the recognition process. As long as the parsing process proceeds and more and more abstract concepts are recognized, they are represented as nodes in increasingly higher layers of the HDPG. The nodes of this graph are connected by two kind of edges. The *hierarchy edges* connect each node representing a concept to the lower layer nodes representing its subconcepts. The graph structure determined by this kind of edges represents the hierarchy of abstraction; this structure is generally a tree, excepted in the case of shared concepts, i.e. when a concept instance is subconcept of more than one concept. The *dependence edges* link together nodes that have abstract control and data dependence relationships between them. Note that, during the recognition process, dependence edges for the newly created abstract concept nodes are inherited from those of the composing subconcept nodes in a way that is characteristic of each concept.

4.2 Schema Matcher

A fundamental operation in the manipulation of ontologies is *match*, which takes two ontologies as input and produces a mapping between elements of the two ontologies that correspond semantically. Match plays a central role in numerous

applications, such as web-oriented data integration, electronic commerce, schema integration, schema evolution and migration, application evolution, data warehousing, database design, web site creation and management, and component-based development. A mapping is defined as a set of mapping elements, each of which indicates that certain elements of schema S1 are mapped to certain elements in S2. Furthermore, each mapping element can have a mapping expression which specifies how the S1 and S2 elements are related. The mapping expression may be directional, for example, a certain function from the S1 elements referenced by the mapping element to the S2 elements referenced by the mapping element, or it may be non-directional, that is, a relation between a combination of elements of S1 and S2.

The *Schema Matcher*, previously designed and developed [14], implements a technique based on syntactic and structural schema matching, among two or more input ontologies.

The matching procedure takes as input two schemas and determines a mapping indicating which elements of the input schemas logically correspond to each other. The match result is a set of mapping elements specifying the matching schema elements together with a similarity value between 0 (strong dissimilarity) and 1 (identity) indicating the plausibility of their correspondence. Our matching procedure combines and integrates a number of matching algorithms, adopting two of the above described approaches: the structural approach, based on the application of the following algorithms: Children Matcher [15], Leaves Matcher, Graph and SubGraph Isomorphism [16]; the linguistic or syntactic approach, based on application of: Edit Distance (Levenshtein Distance) [17] and Synonym Matcher (through WordNet [18] synonyms thesaurus).

4.3 API Ontology Builder

The production of the *Candidate API Ontology Graph* from the source code is performed by the *API Ontology Builder* module by applying graph transformation patterns defined for the specific programming language or model. We have defined a series of transformation rules for object oriented model aimed to extract and transform proper language elements in ontological relation. In [1] similar set of rules are described. Some of the defined rules are illustrated in the following:

- (APIClassNodeA) → (OwlNodeA)
  A node representing a class A in the API graph becomes an OWL class A in the candidate ontology graph.
- (APIClassMethodNodeA) → (OwlNodeA)
  A node representing a method A in the API graph becomes a class A in the candidate ontology graph.
- (APIParameterNodeA) → (OwlNodeA)
  A node representing a parameter A in the API graph becomes a class A in the candidate ontology graph.

- (APIClassNodeA ihneritsEdge APIClassNodeB) → (OwlNodeB subclassOf OwlNodeA)
  If a class A inherits a class B in the API graph, the relation becomes a subclassOf relation between the correspondent classes of the OWL graph.
- (APIClassNodeA hasAttributeEdge APIClassNodeB) → (OwlNodeA ObjectProperty: hasProperty OwlNodeB)
  If a class A has an attribute B in the API graph, the relation becomes an Object Property with label "hasProperyy" between the correspondent classes of the OWL graph.
- (APIClassNodeA hasMethodEdge APIClassMethodNodeB) → (OwlNodeA ObjectProperty: isDoer OwlNodeB)
  If a class A has a method B in the API graph, the relation becomes an Object Property with label "isDoer" between the correspondent elements of the OWL graph.
- (APIClassMethodNodeA hasMethod APIClassMethodNodeConstructorB and APIClassMethodNodeConstructorB hasInputParameter APIParameterNodeC) → (OwlNodeClassA ObjectProperty: hasProperty OwlClassC)
  If a class A has a constructor with some input parameters, in the owl graph there are Object Properties with label "hasProperty" between the correspondent elements.
- (APIClassMethodNodeA hasInputParameterEdge APIParameterNodeB) → (OwlNodeA ObjectProperty: actsOn OwlNodeB)
  If a method A has some input parameter, there are Object Properties with label "actsOn" between the correspondent elements on the OWL graph.
- (APIClassMethodNodeA hasReturnTypeEdge APIClassNodeB) → (OwlNodeA ObjectProperty: produce OwlNodeB)
  If a method A has a return type B there is an Object Property with label "produce" between the correspondent elements in the OWL graph.

5 Conclusion

In this paper we have proposed an approach to perform automatically, or with automated support, operations like the alignment and mapping of software which will be useful to perform software modernization and migration. The methodology is based on an automatic analysis and representation of code at higher level of abstraction than the syntactical and structural one that enables automatic recognition of the algorithms and algorithmic concepts implemented in the source code. Based on matchmaking techniques, the concepts recognized are compared with functional concepts represented by the ontologies and the results provide useful information to perform porting of source code excerpts and API calls to the target cloud programming environment. The architecture supporting this methodology is composed of four components: the Algorithmic COncept Recognizer, which recognizes algorithmic concepts in the code, the API Ontology Builder, which extracts the graph by parsing source code and produces an ontology graph applying graph transformation patterns, the Schema Matcher which

performs the matching among graphs and finally the Annotator which allows the user to semantically annotate the target API with concepts from the functional ontology. This work represents a contribute to facilitate software development in cloud scenario, since in cloud computing environment there are many APIs and services offered by different providers and big efforts are needed both to port applications in the cloud and to migrate from one provider to another. Future work planned includes the introduction of reasoning to extract additional knowledge based on inferential rules running on the acquired knowledge base and on optimization of the adopted graph matching algorithms for the specific graph representations of API components. Natural Language Processing techniques for ontology extraction, already developed by one of the authors [19], [20] are planned to be integrated, in order to deal with entire software artifacts which include natural language components (specification requirements, documentation, etc.).

Acknowledgements. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n 256910 (mOSAIC Project), and by the Italian Ministry of University and Research, PRIN programme (project Cloud@Home). We would like to thank Manuela Serrao (Second University of Naples) who has implemented part of the Matchmaking algorithms.

References

1. Ratiu, D., Feilkas, M., Jurjens, J.: Extracting Domain Ontologies from Domain Specific APIs. In: Proc. of the 12th European Conf. on Software Maintenance and Reengineering, pp. 203–212. IEEE Computer Society (2008)
2. Zhang, Y., Rilling, J., Haarslev, V.: An Ontology-Based Approach to Software Comprehension - Reasoning about Security Concerns. In: 30th Annual International Computer Software and Applications Conference, COMPSAC 2006, vol. 1, pp. 333–342 (2006)
3. Alnusair, A., Zhao, T., Bodden, E.: Effective API navigation and reuse. In: Information Reuse and Integration, IEEE IRI, pp. 7–12 (2010)
4. Eberhart, A., Argawal, S.: SmartAPI - Associating Ontologies and APIs for Rapid Application Development. In: Ontologien in der und für die Softwaretechnik Workshop Anlsslich der Modellierung 2004. Marburg/Lahn (2004)
5. Bontcheva, K., Sabou, M.: Learning Ontologies from Software Artifacts: Exploring and Combining Multiple Sources. In: Workshop on Semantic Web Enabled Software Engineering, GA, USA (2006)
6. Na, H.-S., Choi, O.-H., Lim, J.-E.: A Metamodel-Based Approach for Extracting Ontological Semantics from UML Models. In: Aberer, K., Peng, Z., Rundensteiner, E.A., Zhang, Y., Li, X. (eds.) WISE 2006. LNCS, vol. 4255, pp. 411–422. Springer, Heidelberg (2006)
7. Na, H.S., Choi, O.H., Lim, J.E.: A Method for Building Domain Ontologies based on the Transformation of UML Models. In: Fourth International Conference on Software Engineering Research, Management and Applications, August 9-11, pp. 332–338 (2006)

8. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: Data Structures and Algorithms. Addison-Wesley (1983)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Reading (1995)
10. Di Martino, B.: Algorithmic Concept Recognition to support High Performance Code Reengineering. Special Issue on Hardware/Software Support for High Performance Scientific and Engineering Computing of IEICE Transaction on Information and Systems E87-D(7), 1743–1750 (2004)
11. Di Martino, B., Kessler, C.W.: Two Program Comprehension Tools for Automatic Parallelization. IEEE Concurrency 8(1), 37–47 (2000)
12. Di Martino, B., Zima, H.P.: Support of Automatic Parallelization With Concept Comprehension. Journal of Systems Architecture 45(6-7), 427–439 (1999)
13. Knuth, D.E.: Semantics of context-free languages. Math. Syst. Theory 2(2), 127–145 (1968)
14. Di Martino, B.: Semantic Web Services Discovery based on Structural Ontology Matching. International Journal of Web and Grid Services (IJWGS) 5(1), 46–65 (2009)
15. Do, H.H., Rahm, E.: COMA: System for Flexible Combination of Schema Matching Approach. In: VLDB (2002)
16. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: Performance evaluation of the VF graph matching algorithm. In: Proc. of the 10th ICIAP, pp. 1172–1177. IEEE Computer Society Press (1999)
17. Gilleland, M.: Levenshtein Distance algorithm, Merriam Park Software (2000), `http://www.merriampark.com/ld.html`
18. Princeton University. Wordnet a lexical database for the English language (2006), `http://wordnet.princeton.edu`
19. Di Martino, B.: An Approach to Semantic Information Retrieval based on Natural Language Query Understanding. In: Daniel, F., Facca, F.M. (eds.) ICWE 2010. LNCS, vol. 6385, pp. 211–222. Springer, Heidelberg (2010)
20. Di Martino, B.: Ontology Querying and Matching for Semantic Based Retrieval of Semantically Annotated Documents. In: Proc. of IADIS International Conference on Applied Computing, Rome, November 19-21, pp. 227–232 (2009) ISBN 978-972-8924-97-3