Exploiting Agents and Ontologies for Type- and Meaning-Safe Adaptation of Java Programs

Davide Ancona and Viviana Mascardi
DISI, University of Genova,
Via Dodecaneso 35, 16146, Genova, Italy
{davide,mascardi}@disi.unige.it

***Abstract*—This paper discusses an application of intelligent software agents and ontologies to solve the problem of semi-automatic porting of Java programs.**

We have designed a system for aiding users to adapt Java code in a type- and meaning-safe way, when an application has to migrate to new libraries which are not fully compatible with the legacy ones.

To achieve this, we propose an approach based on an integration of the two type-theoretic notions of subtyping and type isomorphism with ontology matching. While the former notions are needed to ensure flexible adaptation in the presence of type-safety, the latter supports the user to preserve the meaning of names that appear in the program to be adapted.

Intelligent agents control the different components of the system and interact with other agents in order to provide the final user with the semi-automatic porting service he/she required.

I. Introduction

Migrating a Java program p that uses library l into a corresponding program p' that uses library l' in a semi-automatic way is an open problem for which *no satisfying solution has been found yet*.

One aspect that must be considered while facing this problem, and that makes it hard to solve, is that migration must be type-safe. Replacing method m defined by l and used in program p by m' defined in l', thus leading to a new program p', is a legitimate operation only if no type inconsistencies are raised by this replacement. If the functionality of m and m' is the same no type problems will arise. But what should it happen in case of a difference in the type returned by m and m', or in the type of some of their parameters, or in their number and order? The most conservative approach would be to give up, and to consider the migration possible only if elements of l used by p have corresponding elements in l' whose type is identical or isomorphic.

However, this is a very restrictive choice with little motivation: type identity or isomorphism between elements of l and the corresponding elements of l' may be relaxed by requiring that the type τ' of e' in l' is a subtype of the type τ of e in l, for a suitable definition of the subtype relation. This requirement allows a type-safe replacement of e in p with e' in p'.

For example, R. Di Cosmo, F. Pottier and D. Rémy propose an efficient decision algorithm for subtyping recursive types modulo associative commutative products that demonstrates the feasibility of using subtyping instead of type isomorphism, when translating a program into another [1].

The limitation of their work, that we want to overcome by exploiting intelligent agents and ontologies in our system, is that they abstract from the names of classes, methods and attributes and just consider safe matching between types. Since there may be a large number of type correspondences $< \tau, \tau' >$ that preserve type-safety, re-introducing names of classes, methods and attributes into the algorithm that matches libraries' elements may help in removing those correspondences that, even if type safe, are not "meaning-safe". Correspondences between names of methods and attributes are also needed during the translation process where type correspondences are not enough.

Assume that we would like to port p from l to l'. For simplicity, the problem can be reduced to the following example scenario: p is the program

```
AttributeList atts;
String name = atts.getName(0);
```

and l is defined as follows:

```
class AttributeList extends Object {
 String getName(int i){...}
}
```

where `Object` and `String` are the usual predefined classes defined in the standard package `java.lang`.

The library l' to which p has to be ported contains the following class declarations:

```
class Attributes extends Object {
 int    getLength(){...}
 String getLocalName(int index){...}
 String getAttributeType(int index){...}
}
```

The approach discussed in [1] would tell us that the structural types of `AttributeList` and `Attributes` are compliant because of a combination of isomorphism and subtyping. Or, in other words, would tell us that the correspondence <`AttributeList`, `Attributes`> is type safe. This is a useful information, but it does not help us in automatically translating p into p' in order to use l'.

What we would like to have, instead, is the set of correspondences {<`AttributeList`, `Attributes`>, <`getName`, `getLocalName`>}. This set cannot be obtained by just checking the type compliance of `String getName(`**int**`)` with **int** `getLength()`, `String getLocalName(`**int**`)`, and `String getAttributeType(`**int**`)`.

In fact, while `getLength` is not type compliant with `getName`, both `getLocalName` and `getAttributeType` are. However, we expect that the right correspondence is that between `getName` and `getLocalName`, due to the intended meaning of their names.

It is here that **ontologies** come into play: assuming that an "ontology matching algorithm" can devise the correspondences between ontology elements (classes, properties, relationships, individuals) that better respect their intended meaning, and assuming that from a Java library, an ontology carrying the intended meaning of the library elements can be extracted, we propose to extract ontologies o and o' from l and l', and to run a matching algorithm on them.

And it is here that **agents** come into play: the system that we have designed consists of complex components that must provide different kinds of services (type and ontology extraction, type and ontology matching, filtering of the matching results, assisted extraction of the translation function, actual translation) either to the final user or to other system's components. In order to make our system as flexible as possible, we associate an intelligent agent with each component. The agent controls the component and interacts both with other agents and with the user.

The output of the type and ontology matching algorithms, controlled by a *Type Matching Agent* and by an *Ontology Matching Agent* respectively, will be combined by a *Filtering Agent* in order to produce a type- and meaning- safe matching relation. A human user assisted by a *Function Extraction Assistant Agent* will disambiguate multiple possible matchings in order to identify a $match$ function which will finally be used by a *Translation Agent* to translate p into p'.

Continuing the example above, p' would be

```
Attributes atts;
String name = atts.getLocalName(0);
```

where `Attributes = match(AttributeList)` and `getLocalName = match(getName)`. Thanks to the $match$ function, the translation from p to p' can be fully automatized.

The aim of this paper is to discuss a multiagent system that exploits type and ontology matching techniques to make automatic migration of Java programs possible. The paper is organized in the following way: Section II describes the architecture of our multiagent system and Sections III and IV describe the *Ontology Extraction* and *Ontology Matching* agents in detail. Section V concludes and highlights future directions of work.

II. Architecture

The purpose of our multiagent system, depicted in Figure 1, is to provide the service of computing a $match$ function between the elements of two Java libraries l, l' given in input either by a human user or by any other software application, by exploiting interactions among the different agents belonging to it[1]. If the user (agent, software application) wants the additional service of performing the translation of a Java program p that uses library l into a Java program p' that uses l', the $match$ function can in turn be given in input to the Translation Agent which computes a translation p' of p driven by $match$.

The $match$ function is obtained in the following way: ontologies o and o' are extracted from libraries l and l' respectively. In a similar way, collections of types t and t' are extracted from l and l'.

The Ontology Matching Agent interacts with a set of Simple Ontology Matching agents (SOM$_i$ in Figure 1), each in charge of running one specific ontology matching algorithm chosen from a pool of existing ones (see Section IV, last paragraphs). The Ontology Matching Agent may decide to demand the ontology matching service to the SOM agent that has the lowest workload, to the one that seems more suitable to correctly match ontologies o and o' according to quality of service criteria or efficiency needs, or to any other SOM agent according to some policy including running all the available ontology matching algorithms and either merging the obtained results or selecting one of them based on ex-post analysis[2]. At the end, the Ontology Matching Agent obtains from one or more SOMs the alignments (namely, the sets of correspondences) a_1, a_2, ..., a_n between o and o' and merges them or selects the most preferred alignment among them if it is the case. The Type Matching Agents behaves in the same way, controlling a set of Simple Type Matching agents (STM$_j$ in Figure 1) each in charge of running a specific type matching algorithm on t and t' to get tm. The type match tm is used for selecting only those correspondences in a that are type safe. We name this activity "filtering".

Filtering, whose responsibility is given to the Filtering Agent, still does not ensure that we obtain a set of correspondences that is a function: it might still be a relation, because more than one correspondence involving $e \in l$ is both type- and meaning-safe.

The user is involved in the loop for making the relation output by the Filtering Agent turn out into a $match$ function: if many correspondences are possible for an element $e \in l$, the user will be asked to make his/her choice among them. Another information must be integrated into the $match$ function, namely, for any method $m \in l$, which injection must be applied on its parameters $p_1, ..., p_n$ in order to obtain

[1]Currently, some agents belonging to the MAS such as type matching and filtering agents have little decisional power and autonomy, so they could be collected into a single sequential process, simplifying the system design. However, we expect that these agents may be equipped with a higher degree of intelligence in a future version of the system. Hence, we model them as agents even if, in the current version, they are just service providers.

[2]Alignments can be compared according to their precision and recall. Unfortunately, computing precision and recall of an alignment between o and o' is only possible if a *reference alignment* for o and o' has already been developed by hand. In fact, precision is defined as the number of correctly found correspondences with respect to a reference alignment divided by the total number of found correspondences and recall is defined as the number of correctly found correspondences divided by the total number of expected correspondences. The higher the precision and recall, the better. If no reference alignment exists, only quantitative features of the alignment such as dimension, number of correspondences with the same first element, etc, can be considered to decide whether one alignment is "better" than another one.

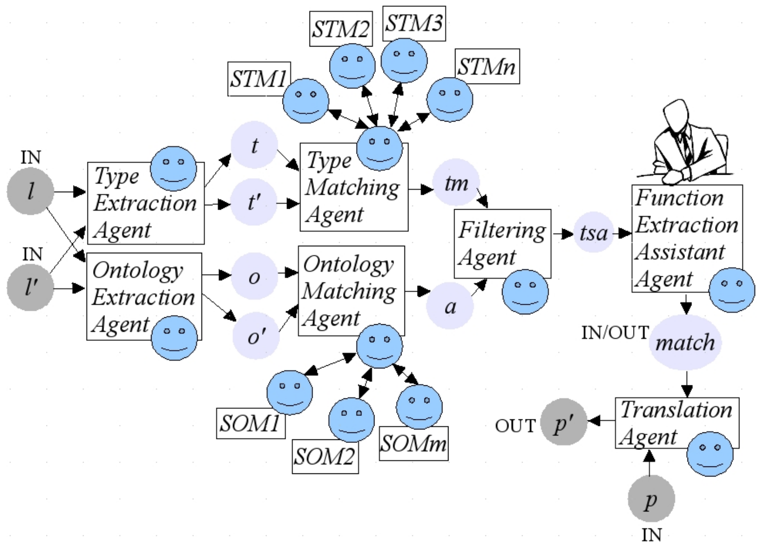

Fig. 1. The architecture of our multiagent system.

the tuple $p_1, ..., p_k$, $k \leq n$ whose ordered elements can be used as parameters for $m' \in l'$, where $m' = match(m)$. Also in this case, the user may be required to make a choice if more injections are possible. For example method `m1(c1,` **int**`, String)` in l might be type- and meaning-safely replaced by `m2(`**int**`, String, c1)` in l, but a permutation of its parameters is required when actually translating p that uses m into p' that uses m'.

The $match$ function (which is indeed a family of functions working either on elements of l, or on tuples of elements of l) is needed by the Translation Agent.

Of course, it might also happen that the Filtering Agent cannot achieve its goal because there are some elements in l for which no corresponding element in l' has been found and thus no $match$ function from l to l' can be computed. The user will be involved in this case too: the Filtering Agent will inform him/her that no type and meaning-safe matching was possible for some elements, and the result of the filtering stage will be shown to him/her. Even if no automatic translation of p will be possible due to the impossibility to generate a $match$ function, the user might find the result of the Filtering Agent useful for driving his/her hand-made translation.

If, thanks to the human intervention, a $match$ function has been defined, the automatic translation of p into p' can be performed by the Translation Agent, leading to the desired output, namely program p'.

In the sequel of this section, each agent is shortly presented. Agents that deal with ontologies are discussed in more detail in the next sections.

Ontology Extraction Agent

The Ontology Extraction Agent takes one Java library as input and returns an ontology that models the structure of the library in term of its classes, their subclass relationships, their methods and attributes. This agent, described in Section III, must operate on both l and l' in order to obtain o and o' respectively.

Type Extraction Agent

The Type Extraction Agent takes one Java library as input and returns a collection of types following S. Jha, J. Palsberg and T. Zhao's proposal [2], [3]. Since Java classes belonging to a library may mutually refer to one another, types in the collection may be mutually recursive. In our system, the Type Extraction Agent must operate on both l and l' in order to extract the corresponding collections of types, t and t' respectively.

Ontology Matching Agent

The service offered by the Ontology Matching Agents is returning an alignment of the two ontologies taken in input. This agent is responsible for the "meaning-safety" of the matching between elements of l and elements of l'; it will take the ontologies o and o' extracted from l and l' respectively as input and will return an ontology alignment a between them. As we will discuss in Section IV, many ontology matching algorithms and tools exists: we will integrate the most relevant ones into our system by implementing, for each of them, a SOM agent that provides an interface towards the algorithm/tool. The Ontology Matching Agent will coordinate the activity of SOM agents

Type Matching Agent

Once the collections of types induced by l and l' have been extracted, a type-safe matching between them must be computed. The algorithm we will use for this activity is inspired by that proposed by R. Di Cosmo, F. Pottier and D. Rémy in [1] and is briefly described in [4]. It ensures the type-safety of the matching.

Filtering Agent

In order to find a matching between the elements of l and those of l' that is both type-safe and that takes the meaning of names of methods, attributes and classes into account, as well as their structural relationships, we need to filter elements of a by taking the type-safe correspondences contained in tm into account. A Filtering Agent that implements the algorithms described in [4] has been designed to this aim.

Function Extraction Assistant Agent

In the general case the output of the Filtering Agent, tsa (for *type safe alignment*), will not be deterministic enough to be used for translating a program p that uses l into the corresponding program p' that uses l'. There might be elements of l that can be matched to more than one element in l' taking both types and meaning into account, and no algorithm could automatically determine the right choice. Once most of the work has been done and the subset tsa of $elements(l) \times elements(l')$ has been generated, the Function Extraction Assistant Agent comes into play and interacts with the user in order to complete the definition of the $match$ function that will drive the translation from p to p'. The task of the user mainly consists in making choices among a set of possibilities provided by the Filtering Agent, in order to constrain a relation to become a function. The user is also asked to define the right operations to be performed on parameters of $m \in elements(l)$ in order to obtain a tuple of parameters suitable for the corresponding method $m' \in elements(l')$.

Of course there might be elements of l for which no type safe matching into a corresponding element of l' exist, and this would mean that tsa could never become a function, and that the system has nothing left to do. The user can benefit from knowing tsa, but he/she has to perform the translation from p to p' by hand.

Translation Agent

In case a the $match$ function has successfully been extracted, the Translator Agent can provide its translation service by taking a function $match$ and a program p and returning a program p' following the rules defined in Section 7 of [4]. The program p to migrate is given in input only to the Translation Agent. The matching function $match$ only depends on l and l': it can be reused for any p developed for using l which must be updated for using l'. The alternative of considering p from the earliest phases of the process has been taken into consideration because of some advantages it would give. In fact, knowing p since the beginning would allow the multiagent system to limit the extraction and matching activities only to those elements of the library that are actually used by p, as well as those that have some dependency relation with them. This would restrict the search space, but would also cause a loss of generality of the function $match$, which should become a $match_p$ function depending on p and might be used only for translating p and programs that use less elements of l than p. A program $p2$ that uses only one more element from l w.r.t p would require the generation of a new $match_{p2}$ function.

III. Ontology Extraction Agent

This section describes the algorithm for automatically extracting an OWL ontology from a Java library exploited by the Ontology Extraction Agent. In case more ontology extraction algorithms should be implemented, the Ontology Extraction Agent might coordinate interface agents towards all or some of them, in the same way as the Ontology Matching and Type Matching agents do.

In order to explain how the extraction algorithm works, we need to provide some details on the subset of OWL that we will use for representing ontologies corresponding to Java libraries. We have designed the extraction in order to make this subset as small as possible. In particular, it is a proper subset of OWL Lite.

a) Data Types: Data Types used in OWL ontologies are those defined by the XML Schema specification, http://www.w3.org/TR/xmlschema-2/:

- *decimal* represents the subset of the real numbers, which can be represented by decimal numerals; *integer* is derived from decimal by fixing the number of decimal digits to 0, and disallowing the trailing decimal point. This results in the standard mathematical concept of the integer numbers. Neither decimal nor integer have a direct counterpart in Java primitive data types.
- *long* is derived from integer by setting the maximum value to be 9,223,372,036, 854,775,807 and the minimum one to be -9,223,372,036,854,775,808 (both included); it corresponds to the *long* Java primitive data type.
- *int* is derived from long by setting the maximum value to be 2,147,483,647 and the minimum value to be -2,147,483,648 (both included); it corresponds to the *int* Java primitive data type.
- *short* is derived from int by setting the minimum admissible value to -32,768 and the maximum admissible value to 32,767 (both included); it corresponds to the *short* Java primitive data type.
- *byte* is a short ranging between -128 and 127 (both included); it corresponds to the *byte* Java primitive data type.
- *float* is patterned after the IEEE single-precision 32-bit floating point type; it corresponds to the *float* Java primitive data type.
- *double* is patterned after the IEEE double-precision 64-bit floating point type ; it corresponds to the *double* Java primitive data type.
- *boolean* has the value space required to support the mathematical concept of binary-valued logic: {true, false}; it corresponds to the *boolean* Java primitive data type.

OWL primitive data types do not include *char*, which is the only Java primitive data type with no direct correspondence. However, since *char* is a finite-valued type type, it may be easily represented as an OWL class with a finite number of instances, as a set of integers with a maximum cardinality (the *owl:maxCardinality* built-in OWL property may be used to this aim), or in other straightforward ways. Instead, OWL primitive data types include for example *string*, *date*, *time* that correspond to some extent to the *String*, *Date*, *Time* classes provided by `java.lang` and `java.sql` packages, respectively.

Since OWL provides no data type corresponding to *void*, we assume that an OWL class named *Void* is defined in a namespace that we abbreviate with *myns*, and that it corresponds to the *void* type specifier in Java.

b) Namespace: Namespaces are inherited by OWL from XML. XML namespaces provide a simple method for qualifying element and attribute names used in XML documents by associating them with namespaces identified by URI references. A standard initial component of an ontology includes a set of XML namespace declarations that provide a means to unambiguously interpret identifiers and make the rest of the ontology presentation much more readable.

c) Class: A class defines a group of individuals that belong together because they share some common properties. The OWL class element, identified by `owl:Class`, is a subclass of the RDFS class element, `rdfs:Class`. The rationale for having a separate OWL class construct lies in the restrictions on OWL DL (and thus also on OWL Lite), which imply that not all RDFS classes are legal OWL DL classes.

d) Subclass: Class hierarchies may be created by making one or more statements that a class is a subclass of another class. This can be achieved by using the `rdfs:subClassOf` element defined by RDFS.

e) Property: Properties have originally being defined in RDF and can be used to state relationships between individuals (object properties, `owl:ObjectProperty`) or from individuals to data values (data type properties, `owl:DatatypeProperty`). Both object and data type OWL properties are subclasses of the RDF class `rdf:Property`.

A. From a Java library to an OWL ontology

The algorithm that we describe in this section has been designed for working under the assumption that names of methods and attributes of the classes in a class library are all different. The absence of name clashes between classes is given for granted, since a class library cannot include two classes with the same name. Even under the assumption that different classes with no inheritance relation among them define different methods, a preprocessing stage must be performed on the library in order to deal with method overriding. In fact, we cannot prevent subclasses from overriding methods defined in superclasses, but this leads to a violation of our assumption on disjoint names of methods. We deal with this situation by just removing the overridden method from all the subclasses that override it. This gives us two advantages:

1) the assumption under which the algorithm works is respected;
2) we avoid that a method m defined by class c may be matched to m', and the same method m overridden by a subclass of c is matched to $m'' \neq m'$.

The basic ideas underlying the extraction algorithm are:

- The Java library l corresponds to a single OWL ontology lo named after the library name and defined in a namespace lns.
- Java classes belonging to l correspond to OWL classes belonging to lo; the identifier of the OWL class coincides with the name of the Java class it corresponds to.
- If the Java class sc extends c, then the OWL class corresponding to c (that we name $owl(c)$ for our convenience) is defined as a subclass of the OWL class corresponding to sc.
- Since properties of an OWL class are inherited by its subclasses, the Java methods and attributes of class c are translated into OWL properties with identifier identical to their name and domain $owl(c)$. This allows them to be inherited by $owl(c)$' subclasses for free. The range of a property corresponding to a Java attribute is defined as the attribute's type; that of a property corresponding to a method is a pre-defined OWL class named `myns:MethodF`.

Our assumption of absence of clash names is very strong, but it allows us to describe the basic ideas underlying the algorithm in a clear and understandable way, discarding the technical details raised by name clashes. The reason for this assumption is that we translate all the elements (classes, attributes, methods) of the class library into corresponding elements of a unique OWL ontology. Unfortunately, an OWL ontology cannot include properties with the same name, even if their domain and range are different as it should happen with methods, parameters and attributes with the same name but different functionality.

In the real case, where name clashes between methods, parameters, and attributes may occur, two solutions have been devised.

1) Instead of translating the entire Java library into an OWL ontology, each Java class c should be translated into an OWL ontology o defined within a namespace ns created starting from c in a way that ensures its uniqueness. Methods and attributes of class c, as well as the methods' parameters, should be translated into properties of the ontology o within the namespace ns. The usage of different ontologies defined in different namespaces should allow us to identify each element of a Java class in a unique way, and thus to overcome the problem of name clashes (using the same identifier in different namespaces is, of course, admitted). The ontology corresponding to the Java class c should import all the ontologies corresponding to translations of Java classes referenced in c, and thus a pre-processing phase should be added to the extraction algorithm. The Java library l should be translated into an ontology that just imports all the ontologies corresponding to the Java classes belonging to l.
   The main drawback of this approach, besides a much more complex extraction algorithm, is that few implemented matching algorithms that the Simple Ontology Matching Agents, SOMs, should interface take namespaces correctly into account.
2) The Java library should still be translated into a single OWL ontology, but clashing names should be modified during their translation in order to obtain an ontology "clash-free".
   Here, the drawback is that the modification of names would result into poorer performances of the ontology

matching algorithms. If, for example, method m in the library l has been translated into $m14$ in ontology o because of a name clash, and method m in library l' has been translated into $m37$ in ontology o', again because of a name clash, the confidence in the correspondence $< m \in o, m \in o' >$ would turn out to be lower than the confidence in the correspondence $< m14 \in o, m37 \in o' >$ for most matching algorithms, because of the syntactic difference between the two names.

The following paragraphs describe the extraction of the OWL elements starting from the Java library elements and provide examples.

OWL elements corresponding to Java classes

A Java class c that extends no class corresponds to an OWL class c (Table I).

A Java class sc that extends a class c different from Object corresponds to an OWL class sc defined as a subclass of c (Table II).

OWL elements corresponding to attributes of Java classes

An attribute a of class c whose type is a basic type t with a corresponding data type in XML corresponds to an OWL datatype property whose ID is a, whose domain is c, and whose range is the XML data type that corresponds to t (Table III).

An attribute a of class c whose type is the class c' defined in the Java library corresponds to an OWL object property whose ID is a, whose domain is c, and whose range is c' (Table IV).

OWL elements corresponding to methods of Java classes

Since we are not interested in representing the functionality of a method m in the ontology, we treat methods in the same way as attributes with the only difference that their range is always an OWL class defined in our namespace, and named `"myns:MethodF"`. The domain of a method is the OWL class representing the Java class it belongs to (Table V).

IV. Ontology Matching Agent

The Ontology Matching Agent will coordinate Simple Ontology Matching Agents, each interfacing towards some existing algorithm and/or tool (for example those mentioned at the end of this section, but others might be considered).

In the recent past, the second author of this paper together with other colleagues from the University of Genova designed, implemented and tested a FIPA compliant Ontology Agent for JADE [5] that provides Ontology Matching services to a MAS [6]. We plan to extend such an agent by adding intelligence to it in the choice of the right matching algorithm (among existing ones) to use, based either on work-balance issues or on quality of service provided, or on both. The experiments described in [7] demonstrate that better results are achieved by more time-consuming algorithms. According to the user's needs, a faster algorithm might be preferred to a slower one, even if this might cause a degradation of the results' quality. The Ontology Matching Agent will take the user's preferences into account for delivering the best service to each user.

In this section, we shortly review the state of the art of ontology matching systems and algorithms towards which Simple Ontology Matching Agent will interface. We draw inspiration from [8]. Following the terminology proposed there, a correspondence between an entity e belonging to ontology o and an entity e' belonging to ontology o' is a 5-tuple $< id, e, e', R, conf >$ where:

- id is a unique identifier of the correspondence;
- e and e' are the entities (e.g. properties, classes, individuals) of o and o' respectively;
- R is a relation such as "equivalence", "more general", "disjointness", "overlapping", holding between the entities e and e'.
- $conf$ is a confidence measure (typically in the $[0, 1]$ range) holding for the correspondence between the entities e and e';

An alignment of ontologies o and o' is a set of correspondences between entities of o and o', and a matching process is a function f which takes two ontologies o and o', a set of parameters p and a set of oracles and resources r, and returns an alignment A between o and o'.

Two of the dimensions according to which matching techniques can be classified are the level (element vs structure) and the way input information is interpreted (syntactic vs external vs semantic).

Level: element vs structure

Element-level matching techniques compute alignments by analyzing entities in isolation, ignoring their relations with other entities. Structure-level techniques compute alignments by analyzing how entities appear together in a structure.

Element-level techniques include, among others:

- *String-based techniques*, that measure the similarity of two entities just looking at the strings (seen as mere sequences of characters) that label them. They include substring distance, Jaro measure [9], n-gram distance [10], Levenshtein distance [11], SMOA measure [12].
- *Language-based techniques*, that consider entity names as words in some natural language and exploit Natural Language Processing techniques to measure their similarity.
- *Constraint-based techniques*, that deal with the internal constraints being applied to the definitions of entities, such as types, cardinality of attributes, and keys.

Structure-level techniques include:

- *Graph-based techniques* that the input ontology as a labeled graph.
- *Taxonomy-based techniques*, that are also graph algorithms which consider only the specialization relation.
- *Model-based techniques* that handle the input based on its semantic interpretation (e.g., model-theoretic semantics). Examples are propositional satisfiability (SAT) and description logics (DL) reasoning techniques.

`public class Bike`	`<owl:Class rdf:ID="Bike"/>`

TABLE I
JAVA CLASS c THAT EXTENDS NO CLASS.

`public class MountainBike extends Bike`	`<owl:Class rdf:ID="MountainBike">` `<rdfs:subClassOf rdf:resource="Bike"/>` `</owl:Class>`

TABLE II
JAVA CLASS sc THAT EXTENDS CLASS c.

Attribute `cadence` of the class `Bike`: `public int cadence;`	`<owl:DatatypeProperty rdf:ID="cadence">` `<rdfs:domain rdf:resource="Bike"/>` `<rdfs:range rdf:resource="xsd:int"/>` `</owl:DatatypeProperty>`

TABLE III
ATTRIBUTE WITH A BASIC TYPE.

Attribute `ft` of the class `Bike`: `public BikeFeatr ft;`	`<owl:ObjectProperty rdf:ID="ft">` `<rdfs:domain rdf:resource="Bike"/>` `<rdfs:range rdf:resource="BikeFeatr"/>` `</owl:ObjectProperty>`

TABLE IV
ATTRIBUTE WITH TYPE c.

Interpretation of input information: syntactic vs external vs semantic

Syntactic techniques interpret the input in function of its sole structure following some clearly stated algorithm.

External techniques exploit auxiliary (external) resources of a domain and common knowledge in order to interpret the input.

Semantic techniques use some formal semantics (e.g., model-theoretic semantics) to interpret the input and justify their results. In case of a semantic based matching system, a further distinction between exact algorithms (that guarantee a discovery of all the possible correspondences) and approximate algorithms (that tend to be incomplete) may be done.

Implemented matching systems and infrastructures

Many implemented matching systems and algorithms exist. If we just consider those listed in the "Project" section of the Ontology Matching portal, http://www.ontologymatching.org/projects.html, we may count about thirty of them. These systems and infrastructures are very different one from another. Many of them have been carefully analyzed and compared in [8], as well as in previous works by the same authors [13], [14] and by other researchers [15].

Just to cite some very recent systems, HMatch [16], [17] is an automated ontology matching system able to handle ontologies specified in OWL. Given two concepts, HMatch calculates a semantic affinity value as the linear combination of a linguistic affinity value and a contextual affinity value. For the linguistic affinity evaluation, HMatch relies on a thesaurus of terms and terminological relationships automatically extracted from the WordNet lexical system. The contextual affinity function of HMatch provides a measure of similarity by taking into account the contextual features of the ontology concepts.

CtxMatch [18], [19] is a sequential system that translates the ontology matching problem into the logical validity problem and computes logical relations, such as equivalence, subsumption between concepts and properties.

The Alignment API [20] is an API and implementation for expressing and sharing ontology alignments. It operates on ontologies implemented in OWL and uses an RDF-based format for expressing alignments in a uniform way. The Alignment API offers services for storing, finding, and sharing alignments; piping alignment algorithms; manipulating (thresholding and hardening); generating processing output (transformations, axioms, rules); comparing alignments. The last release, Version 3.5, dates back to October, 21th, 2008.

AUTOMS-F [21] is a framework implemented as a Java API which aims to facilitate the rapid development of tools for automatic mapping of ontologies by synthesizing several

Methods `setFeatr` and `getFeatr` of the class `Bike`:

```
public void setFeatr
   (BikeFeatr newFeatr,
    String newOwnerName,
    int newOwnersNum)
        {
            ...
        }

public BikeFeatr getFeatr()
        {
            ...
        }
```

```
<owl:ObjectProperty rdf:ID="setFeatr">
    <rdfs:domain rdf:resource="Bike"/>
    <rdfs:range rdf:resource="myns:MethodF"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="getFeatr">
    <rdfs:domain rdf:resource="Bike" />
    <rdfs:range rdf:resource="myns:MethodF"/>
</owl:ObjectProperty>
```

TABLE V
METHODS.

individual ontology mapping methods. Towards this goal, AUTOMS-F provides a highly extensible and customizable application programming interface. AUTOMS [22] is a case study ontology mapping tool that has been implemented using the AUTOMS-F framework.

Finally, automatic matching techniques that exploit "Upper Ontologies", namely general ontologies that deal with concepts that are the same across different domains, have been implemented and analyzed in [7].

V. CONCLUSION AND FUTURE WORK

In this paper we have described a multiagent system that, once implemented, should allow a user to semi-automatically porting a Java program p that uses library l to a program p' that uses l' in a type-safe and "meaning-safe" way. To the best of our knowledge, no previous attempts of exploiting agents and ontologies for facing porting and migration problems exist. We devise some similarity between our proposal and the Natural Programming Project, http://www.cs.cmu.edu/~NatProg/, working on making programming languages and environments easier to learn, more effective, and less error prone. The report [23] suggests that AI tools such as agents, advice, and reversible debuggers may help users convert their intentions into precise programs. In this paper we do not face the general problem of supporting the user in his/her programming activities: we face the more specific problem of helping the user in a migration problem with respect to the Java language. Nevertheless, our exploitation of intelligent agents for supporting the user in activities related to smart programming is coherent with the purpose of the Natural Programming Project.

The contribution of this paper is twofold. On the one hand, we have designed the multiagent system's architecture; on the other hand, we have either identified existing algorithms to integrate in the agents when possible, or designed new ones (the ontology extraction algorithm described in this paper and the algorithms implemented by the Filtering and the Translation agents described in [4] are all original contributions).

The first activity we will carry out in the very near future is the implementation of the algorithms that, at this stage, are only designed. In parallel to the implementation of these algorithms, the choice of the most suitable algorithms and tools to be accessed by Simple Ontology Matching Agents will be made.

Once all these components will be available and tests will be performed over them, a prototype demonstrating the feasibility of our approach will be created in JADE.

ACKNOWLEDGEMENTS

The authors acknowledge the anonymous reviewers for their thoughtful and constructive suggestions.

This work has been partially supported by MIUR EOS DUE - Extensible Object Systems for Dynamic and Unpredictable Environments, and by the CINI-FINMECCANICA Iniziativa Software project.

REFERENCES

[1] R. D. Cosmo, F. Pottier, and D. Rémy, "Subtyping recursive types modulo associative commutative products," in *TLCA 2005, Proceedings*, ser. LNCS, P. Urzyczyn, Ed., vol. 3461. Springer, 2005, pp. 179–193.

[2] J. Palsberg and T. Zhao, "Efficient and flexible matching of recursive types," in *LICS 2000, Proceedings*. IEEE Computer Society, 2000, pp. 388–398.

[3] S. Jha, J. Palsberg, and T. Zhao, "Efficient type matching," in *FOSSACS 2002, co-located with ETAPS 2002, Proceedings*, ser. LNCS, M. Nielsen and U. Engberg, Eds., vol. 2303. Springer, 2002, pp. 187–204.

[4] D. Ancona and V. Mascardi, "Ontology matching for semi-automatic and type-safe adaptation of Java programs," DISI - University of Genova, Tech. Rep., 2008, ftp://ftp.disi.unige.it/person/AnconaD/AM1208.pdf.

[5] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[6] D. Briola, A. Locoro, and V. Mascardi, "Ontology agents in FIPA-compliant platforms: a survey and a new proposal," in *WOA'08, Proceedings*, M. Baldoni, M. Cossentino, F. D. Paoli, and V. Seidita, Eds. Seneca Edizioni, 2008.

[7] V. Mascardi, A. Locoro, and P. Rosso, "Automatic ontology matching via upper ontologies: A systematic evaluation," 2009, IEEE Trans. Knowl. Data Eng., to appear.

[8] J. Euzenat and P. Shvaiko, *Ontology Matching*. Springer, 2007.

[9] M. Jaro, "UNIMATCH: A record linkage system: User's manual," U.S. Bureau of the Census, Washington (DC US), Tech. Rep., 1976.

[10] E. Brill, S. Dumais, and M. Banko, "An analysis of the askmsr question-answering system," in *EMNLP 2002, Proceedings*, 2002.

[11] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Doklady akademii nauk SSSR*, vol. 163, no. 4, pp. 845–848, 1965, in Russian. English Translation in Soviet Physics Doklady 10(8), 707-710, 1966.

[12] G. Stoilos, G. B. Stamou, and S. D. Kollias, “A string metric for ontology alignment,” in *ISWC 2005, Proceedings*, ser. LNCS, Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, Eds., vol. 3729. Springer, 2005, pp. 624–637.

[13] P. Shvaiko and J. Euzenat, “A survey of schema-based matching approaches,” *J. Data Semantics IV*, vol. 3730, pp. 146–171, 2005.

[14] P. Shvaiko, “Iterative schema-based semantic matching,” DIT - University of Trento, Tech. Rep. DIT-06-102, 2006, ph.D. Thesis.

[15] N. Choi, I.-Y. Song, and H. Han, “A survey on ontology mapping,” *SIGMOD Record*, vol. 35, no. 3, pp. 34–41, 2006.

[16] S. Castano, A. Ferrara, and S. Montanelli, “Matching ontologies in open networked systems: Techniques and applications,” *J. Data Semantics V*, pp. 25–63, 2006.

[17] S. Castano, A. Ferrara, and G. Messa, “ISLab HMatch Results for OAEI 2006,” in *OM-2006, co-located with ISWC-2006, Proceedings*, 2006.

[18] P. Bouquet, B. Magnini, L. Serafini, and S. Zanobini, “A SAT-based algorithm for context matching,” in *CONTEXT 2003, Proceedings*, ser. LNCS, P. Blackburn, C. Ghidini, R. M. Turner, and F. Giunchiglia, Eds., vol. 2680. Springer, 2003, pp. 66–79.

[19] P. Bouquet, L. Serafini, S. Zanobini, and S. Sceffer, “Bootstrapping semantics on the web: meaning elicitation from schemas,” in *WWW 2006, Proceedings*, L. Carr, D. D. Roure, A. Iyengar, C. A. Goble, and M. Dahlin, Eds. ACM, 2006, pp. 505–512.

[20] J. Euzenat and et al., “Alignment API and Alignment Server,” 2008. [Online]. Available: http://alignapi.gforge.inria.fr/

[21] A. Valarakos, V. Spiliopoulos, K. Kotis, and G. Vouros, “AUTOMS-F: A java framework for synthesizing ontology mapping methods,” in *KOST ’07, Proceedings*, 2007.

[22] K. Kotis, A. G. Valarakos, and G. A. Vouros, “AUTOMS: Automated ontology mapping through synthesis of methods,” in *OM-2006, co-located with ISWC-2006, Proceedings*, ser. CEUR Workshop Proceedings, P. Shvaiko, J. Euzenat, N. F. Noy, H. Stuckenschmidt, V. R. Benjamins, and M. Uschold, Eds., vol. 225. CEUR-WS.org, 2006.

[23] H. Goodell, S. Kuhn, D. Maulsby, and C. Traynor, “End user programming/informal programming,” *SIGCHI Bull.*, vol. 31, no. 4, pp. 17–21, 1999. [Online]. Available: http://www.cs.uml.edu/~hgoodell/EndUser/blend/report.html