Ontology matching for semi-automatic and type-safe adaptation of Java programs[*]

Davide Ancona and Viviana Mascardi

DISI, Univ. of Genova, Italy {davide,mascardi}@disi.unige.it

Abstract. This paper proposes a solution to the problem of semi-automatic porting of Java programs. In particular, our work aims at the design of tools able to aid users to adapt Java code in a type-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 semantics of the program to be adapted.

1 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'. In [9], 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

[*] 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.

using subtyping instead of type isomorphism, when translating a program into another.

The limitation of their work, that we want to overcome by introducing ontologies in our system, is that they abstract from the names of classes, methods and attributes and just consider safe matching between types. There may be a large number of type correspondences $< \tau, \tau' >$ that preserve type-safety, making their identification of little help to a user that wants to semi-automatize the program translation process and needs correspondences between methods and attributes names, and not just between types. We claim that re-introducing names of classes, methods and attributes into the algorithm that matches libraries' elements will help in removing those correspondences that, even if type safe, are not "semantic-safe". It will also allow the user to obtain a set of correspondences between names of methods and attributes. These correspondences are 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 simple 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 [9] 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 semantics of their names. And it is here that ontologies come into play.

According to T. Gruber, [12],

> *In the context of computer and information sciences, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application.*

Assuming that an "ontology matching algorithm" can devise the correspondences between ontology elements (classes, properties, relationships, individuals) that better respect their intended semantics, and assuming that from a Java library, an ontology carrying the intended semantics 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.

The output of the type and ontology matching algorithms will be combined in order to produce a type- and semantic- safe matching relation. A human user will disambiguate multiple possible matchings in order to identify a matching *function* which will finally be used 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 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 2 describes the architecture of our system and Sections from 3 to 7 describe its components in detail. Section 8 concludes and highlights future directions of work.

2 Architecture

Our system, depicted in Figure 1, takes libraries l, l' as input and returns a *match* function between their elements as output, if possible. The *match* function can in turn be given in input to the translation module which, if fed with a program p, returns 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'.

An ontology matching algorithm is run on o and o' to get the alignment (namely, the set of correspondences) a, and a type matching algorithm is run

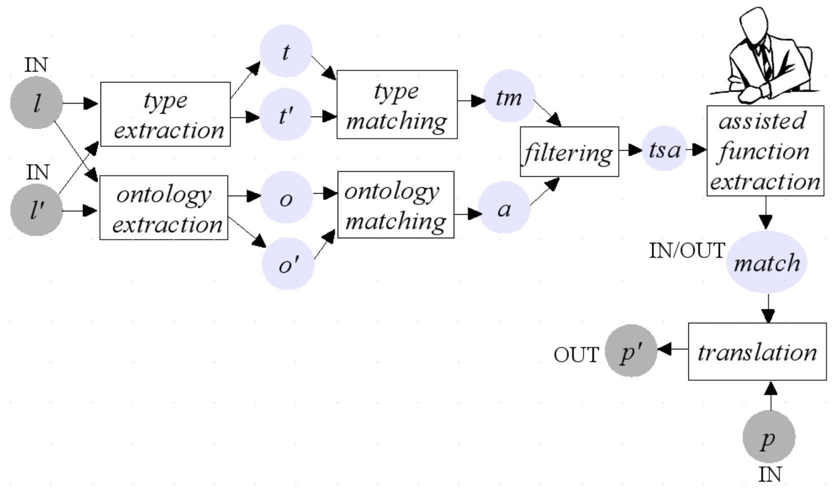

Fig. 1. The architecture of our system.

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 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 semantic-safe.

The user is involved in the loop for making the relation output by the filtering module turn out into a *match* function: if more 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 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 semantic-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 automatic translation module, the last component of our system.

Of course, it might also happen that the output of the filtering module cannot become a function because there are some elements in l for which no corresponding element in l' has been found. The user will be involved even in this case: he/she will be informed that no type and semantic-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 module 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 take place leading to the desired output, namely program p'.

The system consists of seven modules implementing the activities sketched above: ontology and type extraction, ontology and type matching, filtering, assisted extraction of a *match* function, and translation of p into p' guided by *match*.

In the sequel of this section, each activity is shortly presented.

Ontology extraction

The ontology extractor module 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. The ontology extractor, described in Section 3, is run on both l and l' in order to obtain o and o' respectively.

Type extraction

The type extractor module takes one Java library as input and returns a collection of types following S. Jha, J. Palsberg and T. Zhao's proposal [17, 14]. 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 extractor is run on both l and l' in order to extract the corresponding collections of types, t and t' respectively. We shortly discuss it in Section 5.

Ontology matching

The ontology matching module will return an alignment (namely, a set of correspondences between elements) of the two ontologies taken in input. The component devoted to ontology matching will be responsible for the "semantic-safety" of the matching between elements of l and elements of l'; it will be fed with the ontologies o and o' extracted from l and l' respectively and will return an ontology alignment a between them. As we will discuss in Section 4, many ontology matching algorithms and tools exists: we will integrate in our system the most suitable one for our purposes, adapting it if needed.

Type matching

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 [9] and is briefly summarized in Section 5. It ensures the type-safety of the matching.

Filtering

In order to find a matching between the elements of l and those of l' that is both type-safe and that takes the semantics 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.

Extraction of the "match" function (assisted by the user)

In the general case the output of the filtering algorithm, 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 semantics 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 user must enter in the loop 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 system, 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.

Section 6 provides some details on both filtering and user-driven extraction of the match function.

Translation

In case a the *match* function has successfully been extracted, the translator takes a function *match* and a program p and returns a program p' following the rules defined in Section 7. The program p to migrate is given in input to the very last component of the system. 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 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.

3 Extraction of an OWL ontology from a Java library

This section describes an algorithm for automatically extracting an OWL ontology from a Java library. It first introduces the elements of the OWL language that we use for representing ontologies (Section 3.1) and then describes the extraction mechanism (Section 3.2).

3.1 OWL and the Semantic Web

OWL stands at the topmost layer of the standardized technologies in the Semantic Web Stack (see Figure 2). The Semantic Web Stack was introduced for the first time by T. Berners-Lee, Director of the World Wide Web Consortium (W3C), in a keynote session at XML 2000 [2]. It represents the hierarchy of Semantic Web languages, where each layer grounds on the layers below.

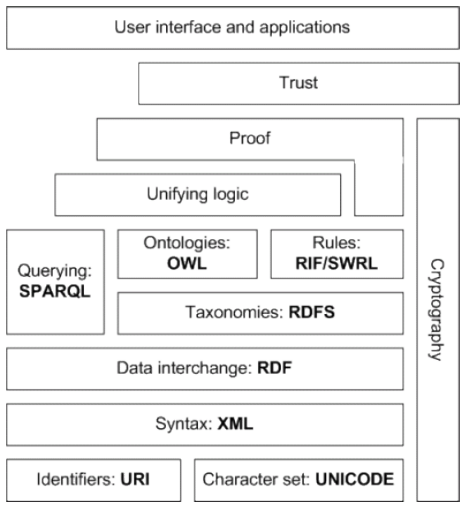

Fig. 2. The Semantic Web Stack [2].

The bottom layers contain hypertext web technologies: Internationalized Resource Identifier (IRI) is a generalization of URI and provides means for uniquely identifying semantic web resources; Unicode serves to represent and manipulate text in many languages; XML is a markup language that enables creation of documents composed of structured data [22]; XML Namespaces provide a way to use markups from more sources [23].

Middle layers contain technologies standardized by W3C to enable building semantic web applications.

- Resource Description Framework (RDF, [26]) allows to create statements about resources in form of triples.
- RDF Schema (RDFS, [25]) provides basic vocabulary for RDF and allows, for example, to create hierarchies of classes and properties.
- Web Ontology Language (OWL, [24]) extends RDFS by adding more advanced constructs to describe semantics of RDF statements; since it is based on description logic [1], it empowers the semantic web with reasoning capabilities.

  OWL provides three increasingly expressive sublanguages: OWL Lite, OWL DL, and OWL Full.
  - OWL Lite supports those users primarily needing a classification hierarchy and simple constraints. It has has a lower formal complexity than OWL DL.
  - OWL DL gives the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs, but they can be used only under certain restrictions (for example, a class cannot be an instance of another class). The "OWL DL" name is due to its correspondence with description logics.
  - OWL Full gives the maximum expressiveness and the syntactic freedom of RDF with no computational guarantees.

  Each of these sublanguages is an extension of its simpler predecessor, both in what can be legally expressed and in what can be validly concluded.
- SPARQL is a RDF query language and can be used to query any RDF-based data, including statements involving RDFS and OWL.

Top layers contain technologies that are not yet standardized or contain just ideas what should be implemented in order to realize Semantic Web. They are out of the scope of this paper.

In order to explain how our 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.

Data Types. Data Types used in OWL ontologies are those defined by the XML Schema specification[1]:

- *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

[1] `http://www.w3.org/TR/xmlschema-2/`

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. Instead, they 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.

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.

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.

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.

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`.

3.2 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 al 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 not all the implemented matching algorithms 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 1).

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 2).

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

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

Table 1. 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 2. Java class *sc* that extends class *c*.

domain is c, and whose range is the XML data type that corresponds to t (Table 3).

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 3. Attribute with a basic type.

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 4).

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 5).

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 4. Attribute with type *c*.

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 5. Methods.

4 Ontology matching algorithms and tools

The ontology matching module of our system will integrate available solutions: as it will become clear in the sequel of this section, implementing a new ontology matching algorithm makes no sense since many proposals are around.

However, we will have to carefully choose the algorithm/tool to integrate, since different algorithms lead to very different results. As an example, some algorithms do not take the structure of the ontology into account, whereas others do.

The ontologies that we match have been built in order to have a meaningful structure: subclasses in the Java library are modeled by subclasses in the ontology; membership of methods and attributes to a Java class *c* is modeled by setting the range of the properties that represent these methods and attributes in OWL equal to the OWL class corresponding to *c*; inheritance of Java methods and attributes from superclasses comes for free by having defined methods and attributes as OWL properties. Thus, selecting an algorithm that performs a matching that takes the ontology structure into account would be the most reasonable solution.

The content of this section is based on the recent book "Ontology Matching" by J. Euzenat and P. Shvaiko, 2007 [11].

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 [13], n-gram distance [5], Levenshtein distance [16], SMOA measure [20].
- *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.

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 [11], as well as in previous works by the same authors [19, 18] and by other researchers [8].

Just to cite some very recent systems, HMatch [7, 6] 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 [3, 4] 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 [10] 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.

Finally, 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 individual ontology mapping methods. Towards this goal, AUTOMS-F provides a highly extensible and customizable application programming interface. AUTOMS [15] is a case study ontology mapping tool that has been implemented using the AUTOMS-F framework.

5 Type extraction and matching

In two papers [17, 14] Jha, Palsberg and Zhao observe that a more expressive notion of structural type equivalence between Java classes and interfaces can be defined by abstracting record types with products and by allowing isomorphism of associative and commutative types.

By following this approach, interfaces and classes can be represented as recursive types using n-ary products and arrow types. For instance, the following class declarations

```
class C1 {
 float  m1(C1 a){...}
 int m2 (C2 a){...}
}
```

and

```
class C2 {
 C1 m3(float a){...}
 C2 m4(float a){...}
}
```

can be abstracted by the mutually recursive types $C1 = (C1 \rightarrow float) \times (C2 \rightarrow int)$ and $C2 = (float \rightarrow C1) \times (float \rightarrow C2)$, where method names are disregarded. Our type extraction module performs the extraction of mutually recursive types in a way similar as shown by the above example. However, product types corresponding to collection of methods and fields (that is, classes and interfaces) are distinguished from product types corresponding to parameters of methods, for the technical reason explained below.

Type isomorphism based on commutativity and associativity of products can be extended with the usual notion of structural subtyping, as done by Di Cosmo [9], yielding the more permissive notion of AC-equality, which allows matching of a class with another having more methods and fields.

To filter the outcome of the ontology matching algorithm, in order to guarantee the type safety of the translation, the general algorithm proposed by Di Cosmo et al. can be reused. It exhibits a reasonable time complexity and can be easily adapted to our notion of subtyping.

As already mentioned, we need to distinguish between two different kinds of products, one for encoding classes and the other for parameters. Subtyping for classes is the usual width and depth subtyping, whereas subtyping for parameters is non standard. To see this, let us consider the usual rules for subtyping between arrow types: $\tau_1 \rightarrow \tau_2$ is a subtype of $\tau_1' \rightarrow \tau_2'$ iff τ_1' is a subtype of τ_1 and τ_2 is a subtype of τ_2'. However, the translation defined in Section 7 works properly only if a method of type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$ is mapped into a method of type $\tau_1' \times \ldots \times \tau_m' \rightarrow \tau'$ s.t. τ' is a subtype of τ, and there exists an injection π from $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$ (hence $m \leq n$) s.t. for all $i = 1, \ldots, m$ τ_i' is a subtype of $\tau_{\pi(i)}$.

The output of the type matching algorithm is a set of couples of classes $< c, c' >$ such that $c \in l$, $c' \in l$, and the type of c' is a subtype of the type of c according to the definition above.

6 Filtering and extraction of the "match" function

The output of the ontology matching algorithm is an alignment a, namely, according to the definition given in Section 4, a set of a 5-tuples $< id, e, e', R, conf >$. Since the semantic relation we are mainly interested in is equivalence, and since we do not need to identify tuples in a unique way, we may rework the alignment in order to discard 5-tuples where R is different from *equivalence*, and we may transform the tuples obtained in this way into triples by discarding id and R. Also, we may discard those triples whose confidence is lower than a certain threshold th decided by the user. Let us name $ref(a)$ the refined alignment obtained in this way.

Elements e and e' appearing in triples of $ref(a)$ belong to o and o' respectively, and, because of the way o and o' have been built, they may be either class names, or attribute names, or method names of elements of l and l', respectively.

On the other hand, the output tm of the type matching algorithm is a set of couples $< c, c' >$ such that $c \in l$ and $c' \in l'$, and $c' \leq c$ according to the AC-subtyping relation defined in [9] and shortly summarized in Section 5.

In order to filter the triples $< e, e', conf > \in ref(a)$ to discard those not type-safe, the filtering module implements the following algorithm:

```
**Filtering algorithm**

$tsa = \emptyset$

begin
for each $< e, e', conf > \in ref(a)$
if $< e, e' > \in tm$ (that is, $e$, $e'$ are classes) then $tsa = tsa \cup < e, e', conf >$
else if $e$, $e'$ are names of attributes
    begin
    retrieve the type $\tau$ of $e$
    retrieve the type $\tau'$ of $e'$
    if $\tau' \leq \tau$ then $tsa = tsa \cup < e, e', conf >$
    end
else if $e$, $e'$ are names of methods
    begin
    retrieve the type of $e$, $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$
    retrieve the type of $e'$, $\tau'_1 \times \ldots \times \tau'_m \rightarrow \tau'$
    if $\tau' \leq \tau$ and there exists an injection $\pi$:$\{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$ s.t. for
    all $i = 1, \ldots, m$ $\tau'_i \leq \tau_{\pi(i)}$ then $tsa = tsa \cup < e, e', conf >$
    end
end
```

After the filtering algorithm has completed its execution, *tsa* contains a type-safe subset of the correspondences initially contained in a.

Two situations may take place: either the domain of *tsa* (namely, the set of e such that $< e, e', conf > \in tsa$) contains all the elements of l, or not.

In the first case it is possible to extract a function $match : elements(l) \rightarrow elements(l')$ from *tsa*, whereas in the second case this is not possible. We have just discussed this second case in Section 2: the user must perform the translation from p to p' by hand exploiting the information contained in *tsa* if useful, but our system cannot help him/her any longer.

In the first case, instead, the user is required to define all the elements of the *match* that will make the automatic translation from p to p' possible; the first activity of the algorithm consists in asking the user to make some choices on elements of *tsa*:

Extraction of the "match" function

```
$match = \emptyset$
**if** element $e \in elements(l)$ appears only in one triple $< e, e', conf > \in tsa$,
no choice is required and $match = match \cup < e, e' >$, else
**for each** $e \in elements(l)$ such that there are more $e' \in elements(l')$ such
that $< e, e', conf > \in tsa$,
    **begin**
    reorder those $< e, e', conf >$ in such a way that triples with higher
    confidence come first
    show them to the user following this order and ask the user to choose
    one
    in case the choice involves an $e'$ that already appears in the range of
    *match*, inform the user that this choice would lead to the loss of injec-
    tivity of *match*
    if $< e, e', conf >$ is the final choice of the user, then $match = match \cup <
    e, e' >$,
**end**
```

The second activity of the algorithm consists in defining the injections that *match* should use for making parameters on m and m' compliant. If class c is declared in l and has a method with n parameters, whereas $m' = match(m)$ has k parameters, the user must choose the right $\pi_{c,m}$ that denotes the injection from $\{1, \ldots, k\}$ to $\{1, \ldots, n\}$ (hence $k \leq n$), specifying how the parameters of method m declared in c are matched with those of method m'. The same applies for constructors. Such injections will be integrated into the *match* function.

Here, we do not go into the details on how this part of the algorithm may be implemented. While checking type-safety, the algorithm of Di Cosmo et al. that we integrate in the type matching module must find at least one such injection (they name it σ) for any couple of methods m and m' that are type-compliant. We may recording all such $\sigma_{(m,m')}$s and propose them to the user as the first possible choice. However, since usually more than one $\sigma_{(m,m')}$ will be acceptable for matching m to m' we must extend the algorithm in order to give more choices to the user, if the $\sigma_{(m,m')}$ found by the type matching algorithm should not satisfy his/her needs.

7 Translation

In this section we formalize a translation of programs in a simple language very similar to FeatherWeight Java. The translation is driven by the *match* function computed by the algorithm presented in the previous section. In the sequel, for presentation reasons, we will indicate $match(e)$ with $[\![e]\!]^{mtch}$ with subscripts if needed. User-chosen injections from tuples of parameters to tuples of parameters are part of the *match* function too, and are denoted with π in the sequel.

7.1 Language definition

$$\begin{array}{rl} cds & ::= cd_1 \ldots cd_n \\ cd & ::= \texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ fd_1 \ldots fd_n\ kd\ md_1 \ldots md_m\ \} \quad (c_1 \neq \texttt{Object}) \\ fd & ::= c\ f; \\ kd & ::= c(c_1\ x_1, \ldots, c_n\ x_n)\{\texttt{super}(e_1, \ldots, e_m); f_1 = e'_1; \ldots f_k = e'_k;\} \\ md & ::= c_0\ m(c_1\ x_1, \ldots, c_n\ x_n)\ \{e\} \\ e & ::= x \mid \texttt{new}\ c(e_1, \ldots, e_n) \mid e.f \mid e_0.m(e_1, .., e_n) \mid (c)\ e \end{array}$$

Assumptions: $n, m, k \geq 0$, inheritance is not cyclic, names of declared classes in a program, methods and fields in a class, and parameters in a method are distinct.

Fig. 3. Syntax of the language

7.2 Definition of the translation

The translation which allows a program to be ported from library cds_{lib} to cds'_{lib} is induced by a matching from cds_{lib} to cds'_{lib}. Such a matching is specified as follows:

- Every class c declared in cds_{lib} is matched by a class $[\![c]\!]^{mtch}$ declared in cds'_{lib}.
- For every class c declared in cds_{lib}, there is an injection mapping every field f of c in a field $[\![f]\!]^{mtch}_c$ of $[\![c]\!]^{mtch}$.
- For every class c declared in cds_{lib}, there is an injection mapping every method m of c in a method $[\![m]\!]^{mtch}_c$ of $[\![c]\!]^{mtch}$.
- If class c is declared in cds_{lib} and has a constructor with n parameters, whereas $[\![c]\!]^{mtch}$ has a constructor with k parameters, then π_c denotes the injection from $\{1, \ldots, k\}$ to $\{1, \ldots, n\}$ (hence $k \leq n$), specifying how the parameters of the constructor of c are matched with those of the constructor of $[\![c]\!]^{mtch}$.
  We denote with $\pi_c[\![(e_1, \ldots, e_n)]\!]$ the k-tuple $(e_{\pi_c(1)}, \ldots, e_{\pi_c(k)})$.

– If class c is declared in cds_{lib} and has a method with n parameters, whereas $[\![m]\!]_c^{mtch}$ has k parameters, then $\pi_{c,m}$ denotes the injection from $\{1,\ldots,k\}$ to $\{1,\ldots,n\}$ (hence $k \leq n$), specifying how the parameters of method m declared in c are matched with those of method $[\![m]\!]_c^{mtch}$. The notation $\pi_{c,m}[\![(e_1,\ldots,e_n)]\!]$ is analogous to that used for constructors.

Note that in principle a translation preserving typings could also be induced by a matching where functions mapping fields, methods, and parameters are not injective. However, injectivity has been imposed for the following two reasons:

– A more efficient algorithm can be implemented for finding all possible candidate classes in cds'_{lib} matching a class in cds_{lib}.
– It is very unlikely that a matching based on non injective maps could be a good choice, since it would imply that library cds_{lib} contains redundant fields or methods, or methods with redundant parameters.

$$[\![\texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ fd_1\ldots fd_n\ kd\ md_1\ldots md_m\ \}]\!]^{tr} = \\ \texttt{class}\ c_1\ \texttt{extends}\ [\![c_2]\!]^{tr}\ \{\ [\![fd_1]\!]^{tr}\ldots[\![fd_n]\!]^{tr}\ [\![kd]\!]^{tr}\ [\![md_1]\!]^{tr}\ldots[\![md_m]\!]^{tr}\ \}$$

$$[\![c(c_1\ x_1,\ldots,c_n\ x_n)\{\texttt{super}(e_1,\ldots,e_m); f_1 = e'_1;\ldots f_k = e'_k;\}]\!]^{tr} = \\ c([\![c_1]\!]^{tr}\ x_1,\ldots,[\![c_n]\!]^{tr}\ x_n)\{\texttt{super}([\![e_1]\!]^{tr},\ldots,[\![e_m]\!]^{tr}); f_1 = [\![e'_1]\!]^{tr};\ldots f_k = [\![e'_k]\!]^{tr};\}$$

$$\begin{aligned}
[\![c\ f;]\!]^{tr} &= [\![c]\!]^{tr}\ f; \\
[\![c_0\ m(c_1\ x_1,\ldots,c_n\ x_n)\ \{e\}]\!]^{tr} &= [\![c_0]\!]^{tr}\ m([\![c_1]\!]^{tr}\ x_1,\ldots,[\![c_n]\!]^{tr}\ x_n)\ \{[\![e]\!]^{tr}\} \\
[\![x]\!]^{tr} &= x \\
[\![\texttt{new}\ c(e_1,\ldots,e_n)]\!]^{tr} &= \texttt{new}\ [\![c]\!]^{mtch}\ \pi_c[\![([\![e_1]\!]^{tr},\ldots,[\![e_n]\!]^{tr})]\!] \\
[\![e{:}c.f]\!]^{tr} &= [\![e]\!]^{tr}.[\![f]\!]_c^{mtch} \\
[\![e_0{:}c.m(e_1,..,e_n)]\!]^{tr} &= [\![e_0]\!]^{tr}.[\![m]\!]_c^{mtch}\pi_{c,m}[\![([\![e_1]\!]^{tr},\ldots,[\![e_n]\!]^{tr})]\!] \\
[\![(c)\ e]\!]^{tr} &= ([\![c]\!]^{mtch})\ [\![e]\!]^{tr}
\end{aligned}$$

7.3 Preservation of typings

We assume that the matching computed from cds_{lib} to cds'_{lib} satisfies the following properties:

1. If `class` c_1 `extends` c_2 $\{\ldots\}$ is in cds_{lib}, then $[\![c_1]\!]^{mtch}$ is declared in cds'_{lib}, and either $c_2 =$ `Object`, or $[\![c_2]\!]^{mtch}$ is declared in cds'_{lib} and is an ancestor of $[\![c_1]\!]^{mtch}$.
2. If c_1 is declared in cds_{lib} and declares field f of type c_2, then $[\![c_1]\!]^{mtch}$ either declares or inherits field $[\![f]\!]_{c_1}^{mtch}$ of type $[\![c_2]\!]^{mtch}$.
3. If c is declared in cds_{lib} and declares constructor $c(c_1\ldots c_n)$, then $[\![c]\!]^{mtch}$ declares constructor $[\![c]\!]^{mtch}(c'_1,\ldots,c'_m)$ s.t. for all $i = 1,\ldots,m$ $c'_i = [\![c_{\pi_c(i)}]\!]^{mtch}$.
4. If c is declared in cds_{lib} and declares method $c_0\ m(c_1\ldots c_n)$, then $[\![c]\!]^{mtch}$ either declares or inherits method $[\![c_0]\!]^{mtch}\ [\![m]\!]_c^{mtch}(c'_1,\ldots,c'_m)$ s.t. for all $i = 1,\ldots,m$ $c'_i = [\![c_{\pi_{c,m}(i)}]\!]^{mtch}$.
5. If c is declared in cds_{lib} and declares method m which correctly overrides method m declared in superclass c', then either $[\![m]\!]_c^{mtch}$ correctly overrides $[\![m]\!]_{c'}^{mtch}$, or the conversely.

Lemma 1. *If* $\vdash \Delta_{lib}, \Delta\diamond$*, then* $\vdash \Delta'_{lib}, [\![\Delta]\!]^{tr}\diamond$.

Theorem 1. *If* $\Delta_{lib}, \Delta; \Gamma \vdash e{:}c$*, then* $\Delta'_{lib}, [\![\Delta]\!]^{tr}; [\![\Gamma]\!]^{tr} \vdash [\![e]\!]^{tr}{:}[\![c]\!]^{tr}$.

Theorem 2. *If* $\Delta_{lib}, \Delta \vdash cds_{lib}\ \ cds\diamond$ *then* $\Delta'_{lib}, [\![\Delta]\!]^{tr} \vdash cds'_{lib}\ \ [\![cds]\!]^{tr}\diamond$

8 Conclusion and future work

In this paper we have described a system that 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 "semantic-safe" way. To the best of our knowledge, no previous attempts of exploiting ontologies for facing porting and migration problems exist.

The contribution of this paper is twofold. On the one hand, we have designed the system's architecture; on the other hand, we have either identified existing algorithms to integrate in the system's modules when possible, or designed new ones (the ontology extraction algorithm, the filtering and user-assisted activities, and the translation algorithm are original contributions of this paper).

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. Besides our algorithms for ontology extraction, alignment filtering, user involvement and program translation discussed in Sections 3, 6, 7, also the type extraction and matching algorithms are, to the best of our knowledge, still to be implemented. In parallel to the implementation of these algorithms, the choice of the most suitable solution for ontology matching 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.

The work to carry out in order to obtain a running prototype is heavy and we will not be able to obtain experimental results in a short time. However we foresee that, by complementing existing proposals for type-safe program translation with the integration of ontology-based semantics of the library elements, our system will provide a valid help to the user in his/her program porting activities.

References

1. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.
2. T. Berners-Lee. Semantic Web - XML2000, 2000. `http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html`.
3. P. Bouquet, B. Magnini, L. Serafini, and S. Zanobini. A SAT-based algorithm for context matching. In P. Blackburn, C. Ghidini, R. M. Turner, and F. Giunchiglia, editors, *Modeling and Using Context, 4th International and Interdisciplinary Conference, CONTEXT 2003, 2003, Proceedings*, volume 2680 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2003.

4. P. Bouquet, L. Serafini, S. Zanobini, and S. Sceffer. Bootstrapping semantics on the web: meaning elicitation from schemas. In L. Carr, D. De Roure, A. Iyengar, C. A. Goble, and M. Dahlin, editors, *15th International Conference on World Wide Web, WWW 2006, Proceedings*, pages 505–512. ACM, 2006.
5. E. Brill, S. Dumais, and M. Banko. An analysis of the askmsr question-answering system. In *Conference on Empirical Methods in Natural Language Processing, EMNLP 2002, Proceedings*, 2002.
6. S. Castano, A. Ferrara, and G. Messa. ISLab HMatch Results for OAEI 2006. In *International Workshop on Ontology Matching, collocated with the 5th International Semantic Web Conference, ISWC-2006, Proceedings*, 2006.
7. S. Castano, A. Ferrara, and S. Montanelli. Matching ontologies in open networked systems: Techniques and applications. *J. Data Semantics V*, pages 25–63, 2006.
8. N. Choi, I.-Y. Song, and H. Han. A survey on ontology mapping. *SIGMOD Record*, 35(3):34–41, 2006.
9. R. Di Cosmo, F. Pottier, and D. Rémy. Subtyping recursive types modulo associative commutative products. In P. Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2005.
10. J. Euzenat and et al. Alignment API and Alignment Server, 2008.
11. J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer, 2007.
12. T. Gruber. Definition of ontology. In *Encyclopedia of Database Systems*. Springer, 2008. To appear.
13. M. Jaro. UNIMATCH: A record linkage system: User's manual. Technical report, U.S. Bureau of the Census, Washington (DC US), 1976.
14. S. Jha, J. Palsberg, and T. Zhao. Efficient type matching. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 187–204. Springer, 2002.
15. K. Kotis, A. G. Valarakos, and G. A. Vouros. AUTOMS: Automated ontology mapping through synthesis of methods. In P. Shvaiko, J. Euzenat, N. F. Noy, H. Stuckenschmidt, V. R. Benjamins, and M. Uschold, editors, *1st International Workshop on Ontology Matching, OM-2006, Collocated with the 5th International Semantic Web Conference, ISWC-2006, Proceedings*, volume 225 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.
16. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady akademii nauk SSSR*, 163(4):845–848, 1965. In Russian. English Translation in Soviet Physics Doklady 10(8), 707-710, 1966.
17. J. Palsberg and T. Zhao. Efficient and flexible matching of recursive types. In *15th Annual IEEE Symposium on Logic in Computer Science, LICS 2000, Proceedings*, pages 388–398. IEEE Computer Society, 2000.
18. P. Shvaiko. Iterative schema-based semantic matching. Technical Report DIT-06-102, DIT - University of Trento, 2006. Ph.D. Thesis.
19. P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *J. Data Semantics IV*, 3730:146–171, 2005.
20. G. Stoilos, G. B. Stamou, and S. D. Kollias. A string metric for ontology alignment. In Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, editors, *4th International Semantic Web Conference, ISWC 2005, Proceedings*, volume 3729 of *Lecture Notes in Computer Science*, pages 624–637. Springer, 2005.

21. A. Valarakos, V. Spiliopoulos, K. Kotis, and G. Vouros. AUTOMS-F: A java framework for synthesizing ontology mapping methods. In *I-KNOW 2007 Special Track on Knowledge Organization and Semantic Technologies 2007, KOST '07, Proceedings*, 2007.
22. W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition) – W3C Recommendation 26 November 2008, 2004.
23. W3C. Namespaces in XML 1.0 (Second Edition) – W3C Recommendation 16 August 2006, 2004.
24. W3C. OWL Web Ontology Language Overview – W3C Recommendation 10 February 2004, 2004.
25. W3C. RDF Vocabulary Description Language 1.0: RDF Schema – W3C Recommendation 10 February 2004, 2004.
26. W3C. RDF/XML Syntax Specification (Revised) – W3C Recommendation 10 February 2004, 2004.