FAST AND ADVANCED STORYBOARD TOOLS

FP7-ICT-2007-1-216048

http://fast.morfeo-project.eu

Deliverable D2.4.2

Mediation amongst ontologies: Application to the FAST platform

Oszkár Ambrus (NUIG)
Albert Zündorf (UniKassel)
Jörn Friedrich Dreyer (UniKassel)
Sebastián Ortega (UPM)

Date: 27/02/2010

FAST is partially funded by the E.C. (grant code: FP7-ICT-2007-1-216048).

Version History

Rev. No.	Date	Author (Partner)	Change description
1.0	27.02.2009	Oszkár Ambrus (NUIG)	final version (D2.4.1) ready for external review
2.0	27.02.2010	Oszkár Ambrus (NUIG)	final version (D2.4.2) ready for external review

Executive Summary

Ontology mediation is a critical problem in heterogeneous environments, since inter-operating services and agents often use different ontologies, and the differences between them need to be determined and overcome to allow for a seamless data exchange between these parties. This is a difficult task, with which web-based agents must cope.

In D2.4.1 we reviewed current work and literature, discussing the main areas of the topic, that is the core problem of ontology mediation, the types of mismatches that occur between ontologies, and the three main ontology mediation approaches, namely (1) the representation of correspondences between ontologies, (2) the (semi-)automatic process of discovering these correspondences and (3) creation of a new ontology from the source ontologies.

Based on the theoretical foundations laid down in D2.4.1 we present our approach to integrating ontology matching into the FAST platform. We present the roles related to ontology matching in FAST, and the way ontology matching is achieved within each one of those roles.

Document Summary

Code	FP7-ICT-2007-1-216048	**Acronym**	FAST
Full title	Fast and Advanced Storyboard Tools		
URL	`http://fast.morfeo-project.eu`		
Project officer	Annalisa Bogliolo		

Deliverable	**Number**	D2.4.2	**Name**	Mediation amongst ontologies: Application to the FAST platform
Work package	**Number**	2	**Name**	Definition of Conceptual Model

Delivery data	**Due date**	28/02/2010	**Submitted**	27/02/2010
Status			final	
Dissemination Level	Public ☒ / Consortium ☐			
Short description of contents	D2.4.2 is the second iteration of the FAST ontology matching approach. The first iteration contained a general overview of the literature regarding ontology mediation, paving the way for futher analysis of the scenarios arising in FAST that require ontology mediation. Based on that general overview, this current interation presents the approach to implement ontology matching within FAST. We present the different roles within the platform, tasks and challenges related to them, and the way ontology matching is integrated within each role.			
Authors	Oszkár Ambrus (NUIG), Albert Zündorf (UniKassel), Jörn Friedrich Dreyer (UniKassel), Sebastián Ortega (UPM)			
Deliverable Owner (Partner)	Oszkár Ambrus (NUIG)		**email**	oszkar.ambrus@deri.org
			phone	+353 91 495112
Keywords	FAST, ontology mediation, ontology matching, ontology alignment			

Table of contents

1 Introduction

1.1 Goal and Scope.

FAST is a visual programming platform allowing business users to build enterprise-class mashups, employing various underlying services and generating new ones. Resources in FAST are described semantically using different ontologies and vocabularies, and can therefore be combined to what we can call “intelligent gadgets”, that rely on semantically enriched building blocks. In the services used within the FAST environment, the ontologies involved in the creation of gadgets and building blocks come from different parties. These ontologies, though different, will sometimes cover the same domain, making ontology matching necessary as the means to reconcile differences in the various conceptualisations.

We aim to integrate ontology matching into the FAST platform to allow for discrepant systems to intercommunicate. We try to find a solution to these differences which range from simple naming differences to more complex data heterogeneity, behavioural differences, discrepant paradigms and other disparities. The reconciliation of these differences is called ontology matching (often mentioned in the specialised literature as ontology mediation, or ontology alignment, thus we will sometimes use these terms as well).

1.1.1 Changes to Previous Version

The first iteration, dealing with the theoretical foundations of ontology matching, is now contained in the Appendix, and the whole body of this document is newly added content, based on the comments of the last review. As such, it tackles design and development issues for integrating ontology matching within the FAST platform: it presents the overall strategy and deals with specific problems belonging to each phase of ontology matching in FAST.

1.2 Structure of the Document.

This document describes the approach of integrating ontology matching into the FAST environment. Sect. **??** gives the big picture of ontology matching in fast, which the remaining three sections expand. Thus, Sect. **??** describes a tool to automatically discover alignments, a scenario and example ontologies to demonstrate the capabilites of the tool, and the testing procedure with the results. The remaing part of the section deals with a tool to define the discovered mapping rules in a way that is consistent with other similar tasks within the FAST platform, and describes the way in which matching operators are stored within the Semantic Catalogue. Sect. **??** shortly describes the way the matching operator is used when composing screenflows and Sect. **??** describes how the matching operator and the defined rules are converted and used in the final, deployed gadget.

The Appendix contains the first iteration of this document, D2.4.1, laying down the theoretical foundations of ontology matching. We discuss ontologies in general in App. **??**, present the ontology mediation problem in App. **??** , discuss the types of mismatches between ontologies in App. **??** and detail the main approaches to ontology mediation in App. **??**.

1.3 Overall Approach for WP2 and relation to other Work Packages and Deliverables.

Ontology matching will play an important role for the integration of different underlying resources in the user-built gadgets. We present our initial research in the implementation of ontology matching in this second iteration of D2.4, after having laid the theoretical ground work of ontology matching in the first iteration. The final version of the deliverable (i.e., D2.4.3) will complete the implementation of ontology mediation within the FAST platform.

Parts of this work rely on the results of D4.3.2, where a service wrapper tool is described, that will be modified to be used in matching ontologies.

2 Overview

2.1 Ontology Matching

FAST uses ontologies to conceptualise the underlying resources used by the different components. Ontologies embody the fundamental vehicle for conceptualising data on semantic systems; they describe the context and semantic background of data that should be known to all agents using it [**?**]. However, different ontologies are often used to describe the same domain or cover the same scenario. This is also true for FAST, where gadget building blocks can originate from different providers, who might use different ontologies to describe them. To ensure interoperability, the task of ontology matching is therefore critical in FAST.

Given two ontologies O and O' that need to be mapped to each other, we adopt the definition given in [**?**]: an ontology mapping element is a 5-tuple $< id, e, e', n, R >$, where id is a unique identifier, identifying the mapping element, e and e' are entities (formulas, terms, classes, individuals) of the first and second ontology, respectively, n is a confidence measure holding the correspondence value between e and e', R is the correspondence relation holding between e and e' (e.g., `equivalence (=)`, `more general(`$\sqsupseteq$`)` or `disjointness(`$\perp$`)`). The alignment operation determines the mapping M' for a pair of ontologies O and O'. The alignment process can be extended by parameters, such as an input mapping, weights and thresholds and other external resources (dictionaries, thesauri, etc.). Different levels of mappings are defined:

(a) A *level 0* mapping [**?**] is a set of the above mapping elements, when the entities are discreet (defined by URIs). E.g., consider the ontology $O1$ with a class `Person`, and another ontology $O2$ with a class `Human`. For this case a matching algorithm could return the mapping element $< id_{11}, Person, Human, 0.67, =>$, meaning that the `Person` class from the first ontology is found to be equivalent to the `Human` class in the second one with a confidence measure of `0.67`. (b) A *level 1* mapping is a slight refinement of level 0, replacing pairs of elements with pairs of sets of elements. (c) A *level 2* mapping can be more complex and defines correspondences in first order logic. It uses the ontology mapping language described in [**?**]. It can describe complex correspondences, such as the one detailed in Sect. **??**.

2.2 Ontology Matching in FAST

The gadget life cycle in FAST has several phases and roles associated, as detailed in [**?**]. Here, we list the ones relevant for the ontology matching tasks, in decreasing order of the measure in which knowledge about ontologies is required. Note that several roles can be played by the same actor.

(i) The *ontology engineer* creates the ontologies used to annotate services and data. This role also includes the process of ontology matching, either automated or manually, determining if the alignment is feasible and creating so-called *matching operator* building blocks, which are basic elements of the FAST gadget building. The *resource developer* then uses these ontologies to annotate resources created in FAST, which resources will be used by the *screen developer*, who will also specify the resource type at the pre- and postconditions of a screen.

(ii) Ontology matching is needed by the *gadget developer* at the design-time of a screen-flow (gadget). Gadget developers use the GVS (Gadget Visual Storyboard) for building gadgets, where they can use the matching operators to combine screens having input and output conditions specifying resources annotated with different ontologies. No actual matching needs to be performed in this phase, but rather the possibility of matching needs to be determined (i.e., *can* two screens A and B be combined?).

(iii) The *end-user* uses the final deployed gadget at run-time, but is unaware of the underlying resources and ontologies or the matching process. Only at run-time the actual mapping of instance data has to be performed.

3 Ontology Engineering Phase

3.1 Alignment Tool

An *ontology mapping* is a declarative specification of the semantic overlap between two ontologies [**?**]. It is the result of the *ontology alignment process*. This mapping is represented as a set of axioms in a *mapping language*. The mapping process has three main phases: (1) discovering

the mapping (alignment phase), (2) representing the mapping and (3) exploiting the mapping.

To accomplish this we need a tool to assist the ontology engineer in the ontology mapping process. Based on the description given in Sect. **??** we identify the following requirements for an alignment tool, that we will take as the basis for an ontology matching component in FAST: (i) All three phases of the process need to be accessible. (ii) Matching of OWL and RDFS ontologies must be supported in the FAST project. (iii) The tool should be as independent as possible, performing the alignment process with little or no user interference. This is an important requirement, since FAST is end-user oriented. (iv) The tool needs to be open source, allowing it to be integrated into the free and open FAST platform. (v) The code should be suitable for porting to other languages (in particular JavaScript), allowing it to be integrated into the FAST gadget run-time. (vi) It should be well documented.

Based on these requirements, we compared three different tools.

MAFRA *MAFRA* [**?**] supports an interactive and incremental process of ontology mapping. It provides an explicit notion of semantic bridges. This representation is serialisable, portable and independent from the mapped languages. The bridges, however, have been designed to be used within the MAFRA system, and the alignment process needs to be done through the provided GUI.

RDFT *RDFT* [**?**] is a small language originally designed to map between XML and RDF. The results are mappings represented in DAML+OIL, that can be executed in a transformation process. No hints are given to add alignment methods or extending the format and the tool does not longer seem to be available.

Alignment API *Alignment API* [**?**] is the tool best matching our requirements, satisfying all the desired conditions. It is still under active development, provides an API and its implementation, is open source (GPLv2 or above) and written in Java, providing an easy way to embed it into other programs. Alignment API can be extended by other representations and matching algorithms, it can be invoked through the command line interface (thus working without user interference) or one of the two available GUI implementations, or it can be exposed as an HTTP server. The tool allows for testing different alignment methods and can generate evaluation results based on a reference alignment. Alignment API can generate the mapping results in XSLT, therefore providing an easy

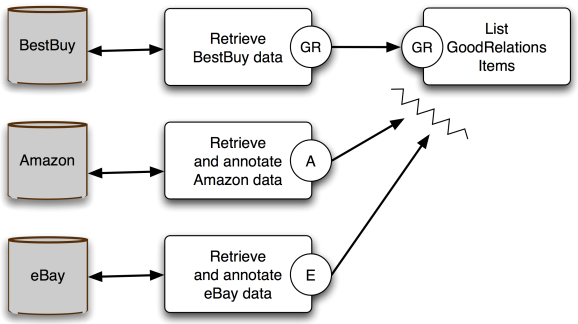

Figure 1: Three gadget components retrieving incompatible data

way to integrate them into other systems.

3.2 Scenario Description

In our evaluation scenario, which is taken from the e-commerce domain, a user needs to build a gadget which combines data from major e-commerce services, allowing to aggregate item lists from all of them in a combined interface. As examples in our scenario, we consider the two most popular online shopping websites[1], Amazon and eBay, along with the BestBuy site. The latter is an interesting case, because it exposes its data in RDF using the GoodRelations (GR) ontology [**?**], which has recently gained a lot of popularity. It is therefore one of the first major e-commerce sites to provide semantic metadata.

Figure **??** illustrates our scenario. There are three retrieval components that wrap the different e-commerce sites and provide data according to three different ontologies: the GR ontology, the Amazon ontology (A) and the eBay ontology (E). Another component displays GR items for display to the user, but not A or E items. If the gadget designer wants to aggregate data from all three services in the display, there will therefore have to be a mapping present between A and E on the one hand, and GR on the other.

[1] `http://alexa.com/topsites/category/Top/Shopping`, checked 01/11/2009

3.3 Ontologies

Of the three ontologies used in our evaluation, only GoodRelations is a real-world, extensive ontology for e-commerce. The other two, i.e., the Amazon and the eBay ontologies were developed for simulation purposes as simplified versions of what would be used in the real-life scenarios. They were designed to showcase particular features of ontology mapping in our scenario.

GoodRelations: This ontology is aimed at annotating so-called "offerings" on the Web, which can be products or services. The ontology features support for ranges of units, measurements, currencies, shipping and payments, common business functions (sell, lease, repair, etc.) and international standards (ISO 4217 or UNSPSC) and codes (e.g., EAN or UPC) in the field. The main class is `Offering`, which represents an announcement by a `BusinessEntity` to provide a `ProductOrService` with a given `BusinessFunction`. It may be constrained in terms of eligible business partner, countries, quantities, and other properties. It is also described by a given `PriceSpecification`. The super-class for all classes describing products or service types is `ProductOrService`. This top-level concept has sub-classes representing actual product instances, product models and dummy product placeholders. A product is described by its title and description, manufacturer, make and model, etc.

Amazon Ontology: We have created a small Amazon ontology based on a subset of the datatypes supported by the web service exposed by Amazon to third-party agents. The ontology describes `Items` based on the `ItemAttributes` description given in the Amazon Product Advertising API documentation[2]. The ontology features three classes for describing a product. Example instance data is given in List. **??**. (1) `Item` represents an Amazon item, defined by a title, a manufacturer, a product group (DVD, Book, etc.), an international EAN code, an ASIN (unique Amazon id), an author (for books) and a `ListPrice`. (2) `Company`, described by a legal name, is used for representing the manufacturer of an `Item`. (3) `ListPrice` has two properties: `hasCurrencyCode`, representing an ISO 4217 currency code (e.g. GBP or EUR), and `hasAmount` representing the price in the given currency.

```
:Item_7590645 a amzn:Item;
   amzn:hasASIN "B0012YA85A";
   amzn:hasManufacturer :Manufacturer_Canon;
   amzn:hasModel "XSI Kit";
```

[2] `http://docs.amazonwebservices.com/AWSECommerceService/latest/DG/`

```
   amzn:hasPrice :Price_7590645_1;
   amzn:hasProductGroup "Electronics";
   amzn:hasTitle "Canon Digital Rebel XSi [...]" .
:Manufacturer_Canon a amzn:Company;
   amzn:hasLegalName "Canon" .
:Price_7590645_1 a amzn:ListPrice;
   amzn:hasAmount "575.55";
   amzn:hasCurrencyCode "GBP" .
```

Listing 1: Simplified Amazon ontology data in N3 notation

eBay Ontology: The eBay ontology was created based on the eBay Shopping API[3] and is supposed to annotate data retrieved through the web service described by the API. The ontology features three basic classes, (1) `SimpleItem` represents an eBay `Item`, that is sold by a `SimpleUser`. It is described by a title, a `CurrentPrice` (specifying the highest bid, or the selling price of fix-priced items), primary category name, manufacturer, model, EAN code, item ID (a unique eBay ID), bid count, end time of bid, country where the item is located, and a product ID (which supports major international product codes — this property is from the Finding API). (2) The `CurrentPrice` features a `hasAmountType` property, specifying the currency code, and a `hasAmount` property, which is the amount of money for a price per unit. (3) `SimpleUser` contains information about eBay users. Users are described by a user ID, about me URL and the seller's positive feedback score. This class will not be used for capturing information on goods for our scenario, but is an essential component of the eBay system, which was the reason for its inclusion in the ontology.

3.4 Testing and Results

We present the approach an ontology engineer has to take to discover and represent ontology mappings, and a means to exploit them after the they have been discovered and appropriately represented.

There is a major paradigm difference between the GoodRelations ontology and the other two ontologies (see Sect. **??** for details). After some initial testing, we concluded that automatic mapping from GR to A/E using the string-based methods employed by the Alignment API tool

[3] `http://developer.ebay.com/DevZone/shopping/docs/CallRef/index.html`

was not feasible. Therefore, the following sections report on *automatic mapping* for the A–E pair — which are similar enough to be suitable for level 0 mapping —, and *manual mapping* for the GR–A/E pairs.

3.4.1 Automatic Mapping

For automatic mapping of level 0 mappings, we used a simple string distance-based algorithm provided by Alignment API [**?**], which computes the string distance between the names of the entities to find correspondences between them. Four methods have been used for computing the distance: (1) equality, which tests whether the names are identical, (2) Levenshtein distance (number of character operations needed), (3) SMOA distance (which is a specialised distance for matching ontology identifiers) and (4) a Wordnet-based [**?**] distance using the JWNL library with Wordnet.

The alignment description derived from these methods is given based on a simple vocabulary, containing a pair of ontologies and a set of correspondences, which express relations between entities of the two ontologies. We used the level 0 mapping representation for representing simple mappings, which map discrete entities of the two ontologies. Thus the representation of the correspondences is given with the five elements described (with the `id` being optional), as shown in List. **??**. Similar mappings were were also used for more complex, manually-created representations (level 2), as detailed in Sect. **??**.

```
<level_0_mapping> a align:Cell;
  align:entity1 amzn:hasCurrencyCode;
  align:entity2 ebay:hasAmountType;
  align:measure "1.0"^^xsd:float;
  align:relation "=" .
```

Listing 2: Level 0 mapping element example

Testing Procedure For the Amazon–eBay pair we set up a reference alignment, against which the results are evaluated. We then ran the matching process for all for methods: (1) equality, (2) Levenshtein distance with a confidence threshold of `0.33` (meaning that any correspondence having a smaller confidence measure will be excluded), (3) SMOA distance with a threshold of

Table 1: Alignment results: Precision, Recall, Fallout and F-Measure

	precision	recall	fallout	f-measure
reference	1.00	1.00	0.00	1.00
equality	1.00	0.38	0.00	0.55
SMOA	0.43	0.75	0.57	0.55
Levenshtein	0.40	0.75	0.60	0.52
JWNL	0.67	0.75	0.33	0.71

`0.5` and (4) Wordnet distance using a threshold of `0.5`[4]. To apply the results, we rendered an XSLT template to transform an example dataset.

Results The results of automatically aligning the Amazon and eBay ontologies were quite favourable. As shown in Tab. **??**, we captured the four main parameters used in information retrieval, as described in [**?**]. These four parameters are used for evaluating the performance of the alignment methods: (1) *Precision*, the fraction of results that are correct — the higher, the better, (2) *Recall*, the ratio of the correct results to the total number of correct correspondences — the higher, the better, (3) *Fallout*, the fraction of incorrect results - the lower the better, and (4) *F-measure*, which measures the overall effectiveness of the retrieval by a harmonic mean of precision and recall — the higher, the better.

The first row (reference) shows the reference alignment, which, naturally, has both perfect precision and recall. We can observe what intuition has predicted, namely that pure string equality (equality) is far too simple and irrelevant, by only taking identical labels. By using string distances and giving certain thresholds (Levenshtein and SMOA), we can see that the results are much less precise, but have a better recall, since this allows for entities having similar names to be discovered, at the expense of having quite a few incorrect results (lower precision); the thresholds allow for low-scored cases to be eliminated, although this results in the exclusion of some correct correspondences. The last column (JWNL) contains the results of the Wordnet-enabled method, which shows quite an improvement (precision of 0.67 and a recall of 0.75[5]), due to the lexical analysis, which performs a much more relevant comparison of strings, giving a high number of correct results. The precision of the JWNL alignment shows only a tiny drop below the recall

[4]Thresholds were selected based on suggestions in the Alignment API documentation
[5]The scores are orientative, giving a general picture of how the process approaches 100% efficiency

value, meaning that the number of incorrect correspondences discovered is small, and the main source of error is from the number of correspondences not discovered.

We can deduce that the results provided are satisfactory, even though the methods used were simple, string-based ones, and the process was completely automated without any user input. We are therefore confident that through some user assistance or an initial input alignment the tool can achieve 100% correct results, which is our aim for the tool to be used in practice.

3.4.2 Manual Mapping

The GoodRelations ontology employs a unique paradigm, different from the paradigms of Amazon and eBay. In GR everything is centred around an instance of `Offering` and a graph of other instances attached to it, whereas for Amazon (and similarly for eBay), the main class is `Item`, which holds all relevant properties. In principle, `Item` would correspond to `ProductOrService` in GoodRelations, but the properties of the `Item` class are reflected as properties of many different classes in GR.

Though the infeasibility of automating this alignment became obvious, we have represented the alignment in the mapping language supported by the tool, as a level 2 mapping (described in Sect. **??**). This mapping description can later be used by the run-time gadget code. List. **??** shows an example mapping between two properties of the two ontologies, specifying that the relationship is `Equivalence` with a certainty degree of `1.0`. This fragment does not show, but assumes the equivalence correspondence between the classes `Item` and `Offering`, which is a trivial level 0 mapping. This mapping specifies the relation

$$\forall v, z; hasEAN(v, z) \Longrightarrow \exists x, y; includesObject(v, x) \wedge$$
$$typeOfGood(x, y) \wedge hasEAN\_UCC\_13(y, z),$$

meaning that the `hasEAN` property of `v` in the Amazon ontology corresponds to the `hasEAN_UCC_13` property of the `typeOfGood` of the `includesObject` of `v` in GoodRelations. The domains and ranges of the properties are inferred, thus it is deduced, that in `Amazon v` is of type `Item` and `z` is `int`, and in `GoodRelations v, x, y` and `z` are instances of the classes `Offering`, `TypeAndQuantityNode`, `ProductOrService` and `int`, respectively.

Using this representation, complex correspondences can be modelled, using first order logic con-

```
<level_2_mapping> a align:Cell;
    align:entity1 amzn:hasEAN;
    align:entity2
       [ a align:Property;
         align:first gr:includesObject;
         align:next gr:hasEAN_UCC_13,
            gr:typeOfGood ];
      align:measure "1.0"^^xsd:float;
      align:relation "Equivalence" .

amzn:hasEAN a align:Property .
gr:hasEAN_UCC_13 a align:Property .
gr:includesObject a align:Relation .
gr:typeOfGood a align:Relation .
```

Listing 3: Fragment of the Amazon–GoodRelations mapping

structs.

3.5 Facts Transformation Tool

To support manual ontology mapping, we plan to develop a rule based, interactive Facts Transformation Tool. This tool utilizes techniques of our service wrapper tool, cf. deliverable D4.3.2. One part of the service wrapper tool deals with a rule based approach for the transformation of server responses into FAST facts. This problem is closely related to the problem of ontology mediation. Thus, these techniques will also allow to create data transformation operators, i.e. operators that mediate between two ontologies.

Figure **??** shows a photoshoped preview of the planned Facts Transformation Tool. The Facts Transformation Tool has two tabs. The *General* tab allows to define the input and output ports for the desired data transformation operator. These ports shall be typed according to the participating semantic elements to be derived from the FAST catalogue. This typing will provide the structure of the input and output data of the desired data transformation operator.

Figure **??** shows the planned editor for transformation rules. On the left of Figure **??** there is a tree of example data that is used to validate the transformation rules during editing. Thus, the Facts Transformation Tool will execute the transformation rules after any editing in order to give direct feedback on their functionality and to enable the user to validate the achieved transformation. The transformation results are shown on the right of Figure **??**.

The middle of Figure **??** shows some possible transformation rules for our example. A *createObject* rule may be used to create output fact objects. The *Target* field of such a rule specifies the

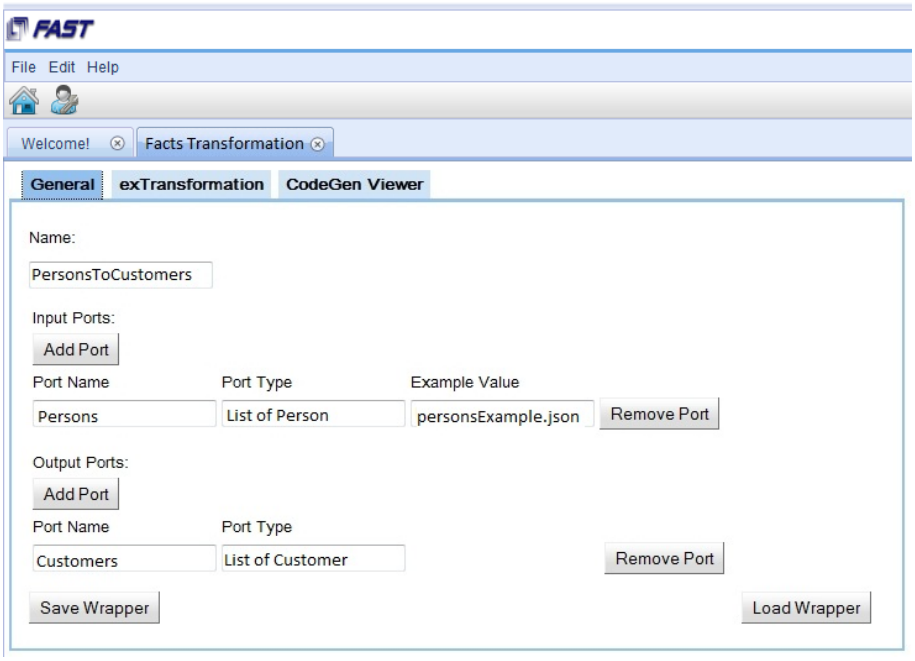

Figure 2: Definition of operator pre- / postconditions (photoshoped screen dump)

type of the created objects, in our example *List*. The top rule fires for each element of the input data with the type specified in the *From* field of the transformation rule. Thus, in our example, the input data is searched for objects of type *List* firing for the root of our data.

The second rule of our example creates *Customer* objects. This rule is a subrule of the root rule and thus the created objects are added to the *List* object created by the parent rule. Similarly, the search for triggering objects of type *Person* is restricted to the subtree below the from element that has triggered the parent rule, i.e. to our input *List*.

The third rule is used to fill the attribute *fullName* of the *Customer* objects. This rule is an example for a many to one transformation. The result is a string concatenation of the *lastName* of the *Person* that has triggered the parent rule plus a text constant plus the *firstName* of the current *Person*. The resulting string is transferred to the *fullName* attribute of the corresponding *Customer*.

Note that simple one-to-one transformation rules should be derived with the help of the ontology mediation approach described in Section **??**. Such simple rules should be added to the transformation tool, automatically. The more complex rules may then be added interactively. During editing, the semantics provided by the FAST catalogue will be used to support the user with completion proposals and consistency checking. In addition, the direct execution of the rules on example data will help the user to create appropriate rules and to validate their effects.

So far our transformation rules are restricted to simple cases. Thus, we have focused on simple cases allowing a simple transformation approach that does not require sophisticated skills from our users. However, as discussed on the Amazon to GoodRelations example, there exist more complex cases. In the third year of the project we will investigate how such more complex cases can be covered, too, and whether it is still possible to keep the rules simple and understandable.

3.6 Creating the Operator

Once the data transformation operator has been defined, it will be added to the FAST catalogue in order to allow its use for the definition of piping configurations of new screens within the FAST CGS tool. For this purpose a simple semantic description will be sent to FAST catalogue defining the semantics of the operator and the semantics of its input and output ports.

Currently, the Service Wrapper Tool uses its own repository for trees of transformation rules. First of all this has technical reasons: the development of the transformation tool is based on a reference architecture and libraries for web based interactive systems provided by Kassel University, cf. [**?**]. This repository has the special advantage that it provides support for teams of developers, thus the service wrapper tool allows to merge the contributions of multiple developers. However, on the long run, we shall store the transformation rules together with the service wrapper and transformation operator definitions in the FAST catalogue.

4 Gadget Building Phase

Every time ontology matching is performed and an alignment is created between two underlying resources, the corresponding alignment rules are stored and an operator is created in the catalogue.

At the gadget building phase, whenever preconditions and postconditions of screens, resources or operators show a semantic mismatch, the catalogue will offer the possible matching operators that would solve the incompatibility problem. This way a matching operator can be placed between two pre- postconditions that would otherwise be incompatible, making interoperation between

Figure 3: Interactive, rule based transforming of Person facts (photoshoped screen dump)

the two possible. In seldom cases, the catalogue may fail to retrieve an appropriate mediation operator. In such cases, the possibility of building a new matching operator with the help of the Facts Transformation Tool will be presented, in order to meet the user's need.

5 Runtime

For deployment, the gadgets implementation is generated in Javascript. In addition, a small FAST runtime library (again Javascript) is added to the gadget. This runtime library provides notification mechanisms and manages the fact base of the gadget. For data transformation we will generate Javascript code that connects itself to the fact base and any time a fact is sent to the input port of the operator, it will do the data transformation and send the result to the output port. The code generated from the data transformation rules follows a recursive pattern where each rule becomes a Javascript function with parameters for the source and target fact subtrees to be transformed. The function body then uses code to search the source tree for sub elements that match the *From* part of the rule and then code for the desired operation is generated. For debugging purposes the user may examine the code generated for an operator in the *CodeGen Viewer* tab of the Facts Transformation tool.

Our code generation for service wrappers and transformation operators uses a template based approach. Thus, the *CodeGen Viewer* tab will also allow to examine the templates used for code generation and to adapt the templates in order to meet changing APIs or to meet new requirements. Thereby, the code generation is not hard wired into our tool but may be adapted by (skilled) end users on the fly.

6 Summary

With the Semantic Web winning more and more ground both in research and industry, the mismatches between ontologies used by the inter-operating systems present an increasing problem. Ontology matching deals with identifying these mismatches, and overcoming them.

In the first iteration (described in the Appendix) we laid the theoretical foundations of ontology matching. After presenting the main problem, and the types of mismatches between ontologies, we have discussed - based on the most prominent research results in these areas - the three main approaches to ontology mediation, that is *ontology mapping* (representing correspondences between entities), *ontology alignment* (the process of finding the correspondences between ontologies) and *ontology merging* (the process of creating a new ontology based on the source ontologies that need to be reconciled).

In this second iteration (described in the main body of this document) we established the approach to integrate ontology matching into FAST, and we have shown how this matching is done at the different phases of the gadget life cycle and the approach taken at every level. Firstly, we have shown the implementation details as well as test scenarios, we have presented a tool for finding the alignments between ontologies as well as a powerful language to represent them, we have described a tool to compose mapping rules in a consistent manner with other FAST components and the means to save them in the Semantic Catalogue. Secondly, we described the way in which mapping operators are used in the GVS to combine discrepant screens. Lastly, we presented the way the matching operator and mapping rules are deployed and exploited in the JavaScript gadget.

For the last and final iteration of deliverable D2.4 we plan to evaluate real-life scenarios and finalize the implementation of ontology matching for the FAST platform.

A Ontologies

This section provides a brief introduction to the concept of ontologies.

A.1 Definitions

As specified in [**?**] “an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse”. These primitives are typically classes, attributes and relations among classes or class instances (and others, such as function terms, rules, restrictions, axioms and events). Their definitions provide information about their semantics (meaning) and constraints on their logically consistent usage.

Ontologies are typically specified in languages that allow to abstract away from implementation details, and focus on the so-called semantic level. This way ontologies provide a representation independent from data models and implementation, thus providing a means to enable interoperability among disparate systems and specifying interfaces to independent, knowledge-based systems. In the technology stack of the Semantic Web, ontologies define a separate layer, and have a set of languages and tools (both commercial and open-source) to work with them.

We would also mention a widely-known and used classical definition of ontologies, presented in [**?**]: ”[An ontology is] an explicit, formal specification of a shared conceptualisation”. [**?**] explains this definition, by elaborating on the four main terms of the definition:

1. it is a conceptualisation, because it models and categorises relevant concepts from the real world,
2. it is explicit, because the model explicitly states the types of the concepts, the relationships between them and the constraints of their use,
3. it is formal, because the ontology has to be machine-readable and consistent, and
4. it is shared, because an ontology is consensual, i.e. accepted by a group of people and it is used in a shared environment to allow a common conceptualisation of data.

A.2 General Description

An ontology is used to define the vocabulary that agents use to exchange queries and assertions. If an agent commits to a common ontology, it guarantees the consistency (but not completeness) of the queries and assertions using the vocabulary defined in the ontology [**?**]. An agent supporting the interface defined in an ontology is not required to use the terms of the ontology as an internal encoding of its knowledge. Nonetheless, the definitions and formal constraints of the ontology do restrict what can be meaningfully stated in the language defined.

Two analogies to ontologies are given in [**?**]. Ontologies are similar to global type declarations in a conventional software library, and ontological commitments are similar to type restrictions over the inputs and outputs of program modules. Formal argument restrictions can be checked mechanically by compilers, to make sure that calling procedures pass legal data to called procedures. Similarly, sentences in an exchange between agents can be checked for logical consistency using the definitions in the ontologies. Just as the formal argument list hides the internal structure of a procedure from its calling environment, a common ontology allows one to interact with a knowledge-based program without committing to or being aware of its internal encoding of knowledge.

Ontologies are also like conceptual schemata in database systems. Such a schema provides a logical description of the data being shared, allowing applications to interact with it without having any knowledge of internal data structures. So while a conceptual schema defines relations and constraints on data, an ontology defines terms with which to represent knowledge, and constraints on their relationships. [**?**] also points out that the requirement of inputs and outputs to be logically consistent with the definitions and constraints of an ontology is analogous to the requirement that the rows of a database table (or insert statements in SQL) must be consistent with the integrity constraints imposed on the table, stated separately and independently of the internal data formats.

[**?**] enumerates the key applications of ontologies, by saying that they “are part of the W3C standards stack for the Semantic Web, in which they are used to specify standard conceptual vocabularies in which to exchange data among systems, provide services for answering queries, publish reusable knowledge bases, and offer services to facilitate interoperability across multiple, heterogeneous systems and databases.” The key role of ontologies in database systems is to provide a data modeling layer above specified database designs, so that data can be used across independently developed systems and services. This made it possible to achieve database inter-

operability, cross database search and web service interrogation.

There are many issues not addressed by common ontologies, related to knowledge sharing. One important question is how a group of people can reach a consensus on a common ontology. The discussion of ontology mediation aims to address this problem by attempting a solution to the case of several ontologies conceptualising a similar domain of the real world in different ways.

B The Mediation Problem

Ontologies are employed as explicit descriptions of the information source semantics. According to [**?**], integrating these semantics along multiple systems can be done using three main approaches. (1) The single ontology approach, using a single ontology as a global shared vocabulary. Unfortunately this can not provide different views on a domain, and using it for multiple similar catalogues is unfeasible due to the vast amount of information needed to be captured, and the susceptibility of the conceptualisation to change. (2) The multiple ontology approach has each information source describing its own ontology. This solves problems related to the single ontology approach, but introduces the problem of differences in conceptualising the same concepts. Ontology mediation can be used to map different parts of the ontologies to each other. (3) The hybrid approach uses multiple ontologies, but uses a global ontology to make all the ontologies comparable to each other. This is closely related to ontology merging, where overlapping entities are merged, and the rest is simply composed into one unified ontology, allowing for the definition of the concepts of both ontologies. We suggest that the FAST Gadget Ontology (as defined in D2.1) could act as a global ontology in the context of FAST, although further research needs to be done in this direction, since the Gadget Ontology has a slightly different epistemological commitment.

In the context of FAST, the problem is that of reconciling the ontologies in the back-end with the ontologies in the front-end, to allow for interoperability between the two layers.

In the following we present a motivating scenario for ontology mediation in Section **??**, and a formal description of the alignment problem and process, respectively, in Section **??**. After that we show some examples of ontology mediation in Section **??**.

B.1 Motivating Example

This section presents a brief example, illustrating a problem that presents the need for ontology mediation.

```
vsmlVariant -"http://www.smo.org/wsml/wsml-syntax/wsml-flight"

namespace { -"http://see.deri.org/adrian/thesis/ontologies/O1#" ,
       wsml - "http://www.wsmo.org/wsml/wsml-syntax#" ,
         dc - "http://purl.org/dc/elements/1.1/" }

ontology - "http://see.deri.org/thesis/ontologies/O1"
    nonFunctionalProperties
    dc#description hasValue "A simple ontology modeling
                             the concept of a person"
    endNonFunctionalProperties

concept person
    name ofType _string
    age ofType _integer
    hasGender ofType gender
    hasChild ofType person
    marriedTo ofType person

concept gender
    value ofType _string

instance male memberOf gender
    value hasValue "male"
instance female memberOf gender
    value hasValue "female"
```

Listing 4: Ontology O1.

Consider the two ontologies o1 and o2 [**?**] presented in Listing 1 and Listing 2 respectively, in the Human Readable Syntax of the Web Service Modeling Language (WSML) [**?**]. Elements from ontology o1 need to be mapped to the elements described in Ontology o2.

Ontology o1 describes the concept person as one having five attributes, each of them having a type that is either a concept or a value from a given data type. The concept gender has two instances defined, that have attributes pointing to the corresponding values.

```
vsmlVariant -"http://www.smo.org/wsml/wsml-syntax/wsml-flight"

namespace { -"http://see.deri.org/adrian/thesis/ontologies/02#" ,
       wsml - "http://www.wsmo.org/wsml/wsml-syntax#" ,
         dc - "http://purl.org/dc/elements/1.1/" }

ontology - "http://see.deri.org/thesis/ontologies/02"
    nonFunctionalProperties
    dc#description hasValue "A simple ontology modeling
                             the concept of a human"
    endNonFunctionalProperties

concept human
    name ofType _string
    age ofType _integer
    noOfChildren ofType _integer

concept man subConceptOf human

concept woman subConceptOf human

concept marriage
    hasParticipant ofType human
    date ofType _date
```

Listing 5: Ontology o2.

Ontology O2 describes the concept human as one having three attributes. The concepts man and woman are subclasses of the concept human.

Additionally a fourth concept called marriage is defined having two attributes. (For simplicity we do not deal with the exceptions of polygamy and same-sex marriage).

Suppose that a service (or agent) using ontology O1 needs to interact with a service accepting and providing data that is semantically defined in ontology O2.

The first step is to identify candidates to be mapped or to have taxonomic relationships under an integrated schema [**?**]. It is obvious that person and human are candidates to be mapped, as well as the class man with the class person having the attribute gender equal to "male" and the class woman with the class person having the attribute gender equal to "female".

After having identified these correspondences, the next step is to generate query expressions and assertions that automatically translate data instances of one of these ontologies into the other one, or that automatically handle their semantic relations under an integrated ontology.

B.2 Problem Statement

Given two ontologies o and o' that need to be mapped to each other, according to [**?**] we can define a mapping element as a 5-tuple <id, e, e', n, R>, where

- id is a unique identifier, identifying the mapping element,
- e and e' entities (classes, attributes, relations) of the first and second ontology, respectively,
- n is a confidence measure in some mathematical structure holding the correspondence value between e and e',
- R is the correspondence relation holding between e and e' (e.g., equivalence (=); more general($\sqsupseteq$); disjointness($\perp$); overlapping($\sqcap$).

A mapping is a set of the above defined mapping elements. The alignment operation determines the mapping (M') for the pair of ontologies (o and o'). There are some parameters than can extend the definition of the alignment process, namely:

1. an input mapping, M, which is to be used and completed by the process,

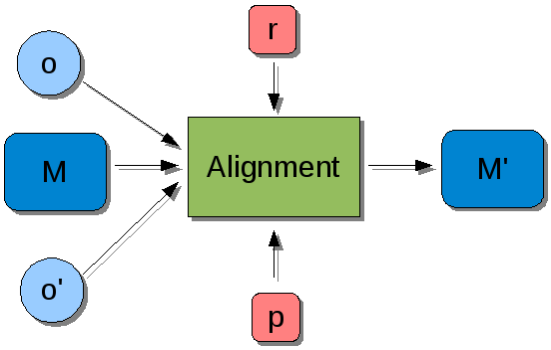

Figure 4: The alignment process

2. the mapping parameters, `p` (such as weights and thresholds), that configure the alignment process,

3. other external resources used by the alignment, `r` (such as dictionaries, thesauri, etc.); See Figure **??**.

For example, suppose that based on some algorithm (using linguistic and structural analysis) the confidence measure for the equivalence relation between the classes `person` in `O1` and `human` in `O2` is 0.62. Suppose that a mapping parameter specifies a threshold of 0.55 for this algorithm, that is all pairs of entities with a confidence measure higher that 0.55 will be considered correct mapping elements. Thus our algorithms would return the following mapping element: `< id11, person, human, 0.62, = >`.Now, another algorithm determines the two concepts to mean exactly the same thing, thus it either does not compute a confidence measure (returning `n/a`), or it returns the upper limit of the confidence measure interval (e.g. 1). Thus the mapping element would be: `< id11, person, human, n/a, = >`.

It should be noted, that the entities mapped between two ontologies can be constructed using patterns, i.e. more than just mapping concepts to concepts, and so on. For example, we can map a concept to a concept having a certain attribute, like in the case of the concept `person` having the attribute `hasGender` equal to `female` from `O1` being mapped to the concept `woman` from `O2`.

B.3 Mediation Examples

We present two further examples to better illustrate the use of ontology mediation.

The first example is taken from [**?**]. Based on the idea that we view ontologies as graph-like structures, we exemplify ontology matching with the help of fragments of two tree-like structures, such as Google and Looksmart. Notice that in the general case the relation holding between nodes is not specialisation, but classification (parent-child relation).

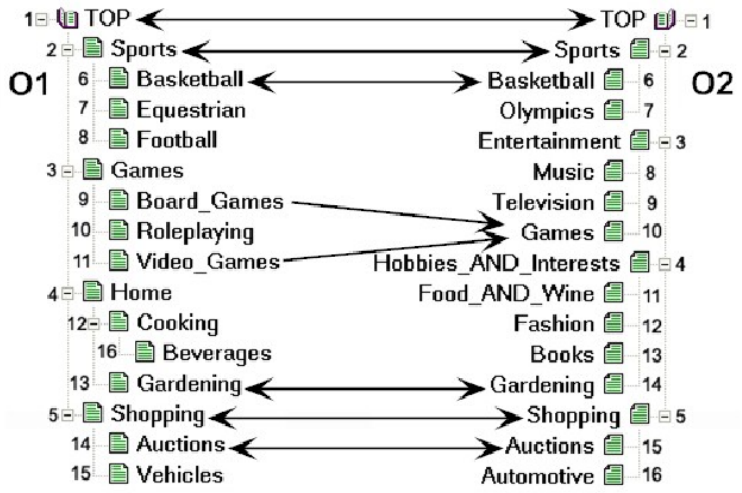

Figure 5: Fragments of Google and Looksmart directories and some correspondences. Double arrowheads denote equivalence, while single arrowheads stand for the more general relation. Image taken from [**?**].

Suppose we want to merge the merge the two structures. Such situations can arise, for example when an e-commerce company acquires another one. One probable mapping is given in Figure **??**. We have found, for example, that `O1:Basketball` is equivalent to `O2:Basketball`, and that `O2:Games` is more general than `O1:Board_Games`.

The second example given in [**?**] illustrates the need to use several techniques in matching two ontologies. Figure **??** shows two purchase order schemas that need to be matched.

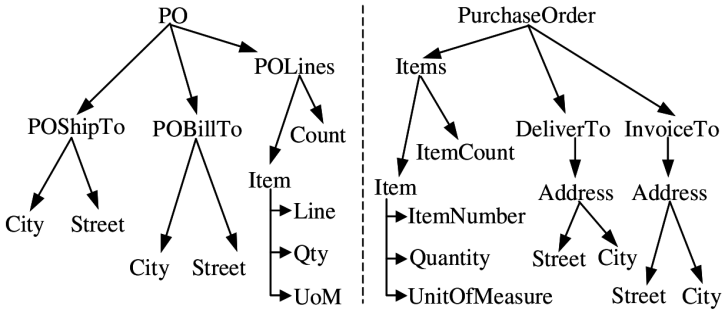

Figure 6: Matching purchase order schemas. Image taken from [**?**].

As stated above in Section 3.2 we look for similarity coefficients between the two schemas and

deduce a mapping from those coefficients. Suppose, as a first phase we use linguistic matching, using thesauri to help match names, acronyms, synonyms, etc. resulting in a set of candidate pairs (such as `PO` and `PurchaseOrder`, or `Bill` and `Invoice`) and language similarity coefficients (`lsim`). In the second phase we would perform a structural matching of the two schemas. For example, `Line` would be mapped to `ItemNumber`, because their parents and their other two children match, `City` under `POBillTo` will be mapped to `City` under `InvoiceTo`, because bill is a synonym to invoice. This process results in structural similarity coefficients (`ssim`). By using a weighted similarity, we can compute the final mappings (e.g. by using the formula `wsim=ws*wsim+(1-ws)*lsim`, where `ws` is the interval [0,1]).

C Ontology Mismatches

Ontology mismatches are the main obstacles in the combined use of independently developed ontologies. We will now give an overview of the main type of ontology mismatches as presented in [**?**], which is the main source of input to this state-of-the-art report. We will classify ontology mismatches into given types and will find their relationships.

Firstly, two levels of mismatches can be distinguished. The first level is the *language* or meta-model level. These are differences in the primitives that make up the language used to specify the ontology. They produce mismatches in the mechanics to define classes, relations, attributes and so on. The second level is the *ontology* or model level. This is were the actual ontology is located. These are differences in the way the domain is modeled.

In this section we discuss the overlap and mismatches between concepts, relations, attributes and instances. We will detail the two levels of mismatches and the different types that occur at each level.

C.1 Language Level Mismatches

We define four types of mismatches, although they often coincide. These occur when ontologies written in different languages are combined. We note that these type of mismatches will be less

relevant for FAST because we assume that standards for ontology languages are now so well established that in practice we will not very often find ontologies written in different languages anymore, but we discuss them briefly to give a better overview of the meditation problem.

C.1.1 Syntax

Different languages obviously use different syntax. Therefore the same concept can (and will be) often modeled in a completely different way. For example, the concept of chairs would be defined in RDF Schema as `<rdfs:Class ID=''Chair''>`, while in LOOM it would be `(defconcept Chair)`.

This is the simplest possible mismatch, although it often comes coupled with other mismatches as well. The simplest form is when a language has several syntactical forms.

The solution is to simply to have a rewrite mechanism that returns the corresponding translated form.

C.1.2 Logical representation

A slightly more complicated mismatch is when a logical notion can not be represented in a given language. For example, in certain languages it is possible to assert that two classes are disjoint, say using `(disjoint A B)`, whereas in other languages it is necessary to express this using simpler statements, such as the negation of set-inclusion statements, like `A subclass-of (NOT B), B subclass-of (NOT A)`.

The actual problem is not the question if something can be expressed, but finding the language constructs to express a logical notion (not a concept).

Solving this mismatch is relatively-easy: it consists in finding translation rules to transform one representation into another.

C.1.3 Semantics of primitives

This difference is a more subtle one, and occurs at the meta-model of the language. It sometimes occurs that the semantics of an identical name differs from language to language. For example, the expression `A equalsTo B` can have many different interpretations.

Even when two ontologies use the same syntax, the semantics may differ. For example, the OIL RDF syntax interprets several `<rdfs:domain>` statements as the intersection of the arguments, whereas RDF Schema interprets them as the union of the arguments.

C.1.4 Language expressiveness

This mismatch, at the meta-model level of the languages, is the one having the greatest impact on the attempt to combine and reconcile ontologies.

The difference implies that some languages can express notions that others can not. For example, some languages have no constructs to express negation, whereas other do. Other typical examples are support for sets, lists, etc.

These type of mismatches have the greatest impact, and are sometimes mentioned as the “fundamental differences” between knowledge models.

C.2 Ontology Level Mismatches

Mismatches at the model level happen both in the case of having the two ontologies written in the same language and syntax and in the case of different languages. We can distinguish between several types of ontology level mismatches.

A conceptualisation mismatch is a difference in how the domain is modeled (conceptualised), therefore we will have different concepts and different relations between them. Explication mismatches are differences in the way the concepts are specified. These occur as mismatches in definitions, or terms, or the combination of the two. Terminological mismatches are naming differences or conflicts. Finally, encoding mismatches are differences in the way values are specified,

for example currencies for prices.

These kinds of mismatches are likely to occur in the context of FAST and are therefore important factors in the implementation of ontology mediation in FAST.

C.2.1 Conceptualisation Mismatches

These are differences in the conceptualisation of the domain, that is not only the specification differs in the two ontologies, but the model as well.

Sometimes these mismatches can not be reconciled automatically, and need a domain expert. In this case, a good solution would be to use both ontologies together. In that case, the overlapping part needs to be aligned, and the rest can be merged.

Scope. Two classes that seemingly represent the same concept, but do not have the same instances, although some of these are identical. For example the concept of “employee” can be perceived and specified in different ways by different organisations.

Model coverage and granularity. This mismatch is related to the part of the domain that is covered by the ontologies, and the level of detail respectively. For example, an ontology could model cars, but no trucks, whereas another would have a few categories for trucks, and another third ontology could make very detailed distinction between all the types of trucks and their properties.

C.2.2 Explication Mismatches

The first two types of explication mismatches (*paradigm* and *concept description*) result from the **style of modeling.** The next two types (*synonym terms* and *homonym terms*) can be classified as **terminological mismatches**. Finally, there is one trivial difference, cause by *encoding* mismatches.

Paradigm. Different paradigms can be used to represent different concepts, or different top-level ontologies can be used, which all result in this kind of mismatch. For example, one could model time based on interval logic, while another could use discrete time points.

Concept Description. These types of mismatches are called modeling conventions. Several choices can be made in different modeling questions. One choice, the categorisation of concepts can be done based on an attribute, or using subclasses. For example, as presented in Section 3.1, the concepts "man" and "woman" can be modeled in several ways. They could be instances of the concept, and differ only in an attribute (e.g. `gender`), or they can be separate subclasses of a top-level class representing "humans" in general.

Another choice is the way in which a hierarchy is built - distinction between features can be made higher or lower in the hierarchy. For example, consider the place where the difference between scientific and non-scientific publications is made: a dissertation can be modeled as `dissertation < book < scientific publication < publication`, or as `dissertation < scientific book < book < publication`, or as a subclass of both `scientific publication` and `book`.

Synonym Terms. Concepts are represented by different names. For example, the term "car" from an ontology can be represented as "automobile" in another.

Although the solution looks simple in using thesauri, in reality much human effort is needed to solve these problems, and care must be taken not to overlap with scope mismatches (see above).

Homonym Terms. The meaning of a term is different in another context. For example, the term "conductor" has one meaning in a musical context, and a different meaning in an electric engineering domain.

These differences are much harder to solve than synonyms, and always need human contribution.

Encoding. The values of attributes in ontologies are encoded in different formats. For example, date can be encoded in a ''`yyyy-mm-dd`'' form, or a ''`dd/mm/yyyy`'', distance might be specified in miles or kilometers, money amount can be given in different currencies, and so on.

These kind of mismatches can very easily be solved, by using a wrapper, or a single transformation step.

D Ontology Mediation Approaches

In this section we give an overview of some of the major approaches to ontology mediation, following the structure of Section 2: Approaches in Ontology Mediation of [**?**]. Firstly, in Section **??** we present some representative approaches to the specification of mappings. Then in Section **??** we overview some of the main approaches and algorithms to overcome mismatches and find mappings between ontologies, that is we discuss the topic of ontology alignment. Finally in Section **??** we discuss the process of merging several ontologies. Further research is needed to decide on the type of ontology mediation to be used in the context of FAST, the algorithms and approaches applied, and whether to use alignment or merging.

D.1 Ontology Mapping

An ontology mapping is a declarative specification of the semantic overlap between two ontologies [**?**]. It is the result of the ontology alignment process (see Figure 3.). This mapping is usually expressed as a set of axioms in a certain mapping language.

There are three main phases of the mapping process: (1) discovering the mapping, (2) representing the mapping and (3) using/exploiting/executing the mapping. This section is focused on (2), that is we survey a number of approaches in representing ontology mappings.

A common tendency in ontology mapping is to create an ontology of mappings, which represents the vocabulary for the representation of mappings.

MAFRA. MAFRA [**?**], or MApping FRAmework for distributed ontologies supports the interactive, incremental and dynamic process of ontology mapping, which transforms instances of a source ontology into instances of a target ontology.

Framework Components. The framework consists of five main phases (called horizontal dimensions) and for components that run along the entire process, called vertical components. Within the horizontal dimension, the authors identified five modules:

- *Lift & Normalisation*. Both ontologies must be normalised to the uniform representation of

RDFS. As a result both ontologies are represented in RDF Schema with their instances in RDF. Thus syntactical end lexical differences are eliminated and semantic differences become more apparent.

- *Similarity.* This module establishes similarities between entities using different approaches (*lexical similarity*, *property similarity* based on properties of concepts, *bottom-up similarity* propagating the similarity from lower parts of the taxonomy to the upper part and *top-down similarity* that assumes special relevance when top level concepts have a higher or lower similarity).
- *Semantic Bridging*. Based on the similarities computed in the previous phase, correspondences are identified between entities, using so-called semantic bridges, which define the actual mapping.
- *Execution*. Transforms instances between the two ontologies using the semantic bridges.
- *Post-processing.* Revisiting the mapping for improvements.

The vertical dimension contains modules like evolution (synchronising changes between in the ontologies and the bridges), the GUI (extensive graphical support must be given for human assistance of the process), etc.

Semantic Bridges. Semantic bridges are captured in the Semantic Bridging Ontology (SBO). Five dimensions have been identified for the semantic bridges, as presented in the following:

- *Entity dimension.* Semantic bridges may relate to the following entities: (i) concepts modeling object classes from the real world, (ii) relations modeling relationships of objects, (iii) attributes modeling simple properties and (iv) extensional patterns modeling the content of instances.
- *Cardinality dimension.* Defines the number of entities on both sides of the semantic bridge (mostly 1:1, 1:n and m:1; m:n is rarely encountered, and can be usually decomposed onto m:1:n).
- *Structural dimension.* Reflects on how elementary bridges may be combined into more complex bridges. Relations that hold among bridges are *specialisation* (reuse a bridge and provide additional information), *abstraction* (should only be used as a super-class),

composition (a bridge is composed of other bridges) and *alternatives* (mutually exclusive bridges).

- *Constraint dimension.* Permits to control the execution of the semantic bridge. Constraints must hold on the source ontology in order for the transformation procedures to be applied.
- *Transformation dimension.* Reflects on how instances of the source ontology are transformed during the mapping process.

The Semantic Bridging Ontology (SBO) contains a specification of all semantic bridges, organised in a taxonomy. To actually relate the source and the target ontology the mapping process creates an instance of the SBO containing semantic bridge instances along with all the necessary information to transform instances of entities from the source to the target ontology.

The five semantic bridge dimensions are employed as follows:

- There are three basic entity types: *Concepts*, *Relations* and *Attributes*.
- The class *SemanticBridge* is the most generic bridge type. It defines the relations to the source and target entities. It is specialised according to entity type and cardinality of the relation.
- The class *Service* can reference resources that are responsible to connect to, or to describe transformations.
- *Rule* is the general class for constraints and transformation-relevant information.
- The class *Transformation* uses the *inService* relation to link to the transformation procedure.
- The class *Condition* represents the conditions that need to be satisfied in order to execute the semantic bridge.
- The Composition primitive can be accomplished using the *hasBridge* relation of *SemanticBridge*, without any cardinality or type constraint.
- The Alternative primitive is supported by the *SemanticBridgeAlt* class. It groups several mutually exclusive bridges, of which the first consistent bridge (with satisfied conditions) is executed.

For an extensive and relevant example please refer to Section 3.3. of [**?**].

RDFT. RDFT [**?**] is a small language (meta-ontology) defined for mapping XML DTDs to/and RDF Schemas specially targeted for business integration tasks. It is built on top of RDF Schema and it is used to map RDF Schemas and concepts like events, messages, vocabularies to XML-specific parts of the conceptual models that occur in the integration tasks.

The model is derived from WSDL [**?**] and provides an RDF Schema for RDF annotations of WSDL documents, and is extended with the temporal ontology PSL. The business integration task in this context is seen as a service integration task, where each enterprise is represented as a Web service specified in WSDL. WSDL annotations made according to the defined meta-ontology allow performing inference over WSDL descriptions to validate the links established between the enterprises.

PSL. To be able reason about the inputs and outputs of the companies being integrated, the authors developed a conceptual model of WSDL, which is directly derived from the WSDL specification. WSDL defines the following basic elements of services, on top of which PSL is built:

- `Types` that provide links to the XML Schemas of the messages exchanged;
- abstract definitions of `Messages` in accordance with `Types`;
- `Port Types` that specify input and output messages;
- `Bindings` that specify concrete protocols and formats for messages according to `Types`.

These elements can describe services but do not represent any temporal knowledge about the messages requiring integration. The Process Specification Language or PSL temporal ontology includes classes to capture temporal knowledge:

- `activity` , which is performed during a certain time interval;
- `activity occurrence` , contains the snapshot of an activity at a moment in time;
- `timepoint` , marks the time interval of the activity;
- `objects` , things that do not possess temporal properties, but may participate in certain activities at given timepoints.

RDFT. The basic RDFT class is `Bridge` that connects two concepts. It describes common properties of bridges, which enables one to specify correspondences between an entity and a set of entities. allowing for one-to-many and many-to-one connections.

The bridges contain the `Relation` property, pointing to one of the subclasses of `BridgeRelation`:

- `EquivalenceRelation`, stating that the source element is equivalent with the target set of elements, or the source set of elements is equivalent to the target element respectively, depending on the relation cardinality;
- `VersionRelation`, specifying that the target set of elements forms a later version of the source set of elements. Unlike equivalence bridges, assumes that both concepts belong to the same domain.

Several types of bridges are defined in RDFT:

- `Event2Event` bridges link different events, specify temporal event generation conditions, and link the events to the messages transmitted with them. They connect instances of the meta-class `mediator:Event`.
- two kinds of `RDFBridges`: `Class2Class` and `Property2Property` bridges between RDF Schema classes and properties. They contain `rdfs:Class` and `rdfs:Property` instances, respectively.
- four kinds of `XMLBridges`: `Tag2Class`, `Tag2Property`, `Class2Tag`, `Class2Property`, which link XML DTD tags and RDF Schema classes and properties.

The bridges are grouped into `Maps`, which are collections of bridges serving a single purpose. The maps are identified by their names and form minimal reusable modules of RDFT bridges. Each map can include other maps and serves as a container for `Bridges`. Described in this manner, as a set of bridges, mappings are said to be *declarative*, while *procedural* mappings can be defined as Xpath [**?**] expressions, transforming instance data.

Connecting two services with RDFT consists of connecting their events (instantiating `EventMap`) consisting of `Event2Event` bridges. Each of the bridges points to a `DocumentMap` aligning the documents attached to the bridges and, in turn, consisting of `RDFBridges`, `XMLBridges` and `VocabularyMaps`.

C-OWL. C-OWL [**?**] gives another perspective on ontology alignment, extending the widely-known OWL [**?**] language.

The Vision. In their vision, similar to the one discussed at the beginning of Section 3, two types of domain conceptualisations exist:

- *ontologies*, that encode a view common to a set of different parties. They make it easy to exchange information, although they have the requirement of common consensus and the problem of hard maintenance.
- *contexts*, that encode the view of a party, and are local conceptualisations, that are not shared. These are easy to maintain, and no consensus is required, but need explicit mapping among the contexts of different parties to be able to exchange information.

In this vision, an ontology is contextualised, i.e. it is a *contextual ontology*, when its contents are kept local (therefore not shared) and can be mapped to the contents of other ontologies via explicit mappings, allowing for a controlled form of global visibility. This is opposed to the OWL importing mechanism, where a set of local models is globalised in a unique shared model, by importing complete models and using the imported elements by direct reference.

Context OWL. Context OWL or C-OWL is a language that extends OWL both syntactically and semantically to support the concept of contextual ontologies.
Therefore a contextual ontology defined in C-OWL is a pair consisting of:

1. an OWL ontology, and
2. mapping between contexts, namely a set of bridge rules with the same target ontology.

A C-OWL mapping therefore is a 4-tuple having the following form:

1. a mapping identifier (URI),
2. a source context containing an OWL ontology (URI of the ontology),
3. a target context containing an OWL ontology (URI of the ontology),
4. a set of bridge rules form the local language of the source ontology to the local language of the target ontology. Each mapping is composed of three elements:

(a) a source element (concept, role or individual) of the source ontology;

(b) a target element, which must be of the same type as the source element;

(c) the type of mapping (subsuming($\sqsupseteq$), equivalence(=), disjointness($\perp$) and overlapping($\sqcap$)).

Thus C-OWL has the complete representational power of OWL, augmented with appropriate local model semantics for mapping between contexts (local ontologies). A nice property of C-OWL is that the two components (the OWL ontology and the mapping) are orthogonal, thus one can use the ontology or the contextual component in an independent manner.

The local model semantics defined in C-OWL, as opposed to OWL, considers that each context has a local set of models and a local domain of interpretation. Thus, it is possible to have contradicting axioms or unsatisfiable ontologies in the local models, without having the entire context space unsatisfiable.

D.2 Ontology Alignment

D.2.1 The Match Operator

Ontology alignment is the process of discovering similarities between two source ontologies [**?**]. This process can be described as the application of the so-called *Match* operator described in [**?**]. The result of the matching process is a specification of the similarities between the two ontologies. The input of the Match operator consists mainly of two ontologies and some other inputs (initial mapping that will be extended, mapping parameters that configure the process and external resources used by the matching process; see Section 3.2).

The generic implementation of the Match operator should contain a uniform internal representation of the schemas to be matched, in order to significantly reduce complexity. This, of course, requires a semantics-preserving schema importer and exporter, respectively.

In general it is not possible to fully automate the matching between two ontologies, primarily because they often have semantics that affects the matching criteria, but is not formally expressed or often even documented. Therefore the implementation of Match should determine match candidates, which the user can accept, reject, or change. Furthermore the user should be able to specify matches for elements for which the system was unable to find satisfactory match candi-

dates.

D.2.2 The Alignment Process

The alignment process proposed in [**?**] relieves the user of some of the burdens in creating the mappings. The input of the process consists of two ontologies which are to be aligned. The output is a set of mappings.

The steps of the alignment are as follows:

1. *Feature engineering*. Selects only parts of an ontology definition in order to describe a specific entity. A feature may be as simple as a label or it may include super- and sub-concepts, relations or extensional descriptions.
   For example, suppose a fragment of an ontology describes the instance `Daimler`. In this case we should consider the generic ontology feature called type which has the values `luxury` and `automobile`, respectively.

2. *Selection of Next Search Steps*. This step chooses the next set of candidate pairs. It may choose to compute a restricted subset of candidate concept pairs of the two ontologies and ignore the others.

3. *Similarity Assessment*. Determines the similarity degree of candidate pairs. Heuristics are used, that is similarity functions such as on strings, object sets, related concepts, and so on. For example, one similarity function can check whether the parents of the compared two concepts are identical.

4. *Similarity Aggregation*. For a candidate pair we can (and often do) use several similarity functions. The results of these functions need to be aggregated to compute the final similarity measure between the two entities. This can be done by either a simple averaging, or a complex aggregation function using weighting schemes.

5. *Interpretation*. This step uses the aggregated similarity values to align entities. It can use thresholds for similarity, perform relaxation labelling, or combine structural and similarity criteria.
   For example, suppose `simil(o1:Person, o2:Human)=0.62` $\geq$ `0.5`, thus it is added to the

result as a mapping element.

6. *Iteration*. Several algorithms perform more than one iteration in order to bootstrap the amount of structural knowledge. Iteration may stop when no new alignments are proposed, or when a given number of mappings have been discovered. Note that certain steps can be omitted in later iterations, since we can use previously computed values (such as selecting features, computing a given similarity, etc.).

The output of the process is a set of correspondences between entities of the two ontologies. We cannot expect all correspondences to be discovered, therefore the results of the alignment process can be considered as the input to a manual refinement process.

D.2.3 Classification of Approaches

An implementation of Match may use multiple matching algorithms or *matchers.* This allows us to select matchers according to the application domain or schema types.

Given that we want two use multiple matchers we distinguish two subproblems. First, there is the need to realise individual matchers, each of which uses a single matching criterion to find mappings. Second, we want to combine different individual matchers, either by using multiple matching criteria within an integrated *hybrid matcher*, or by combining multiple match results provided by different match algorithms within a *composite matcher*.

For individual matchers, we consider the following mostly-orthogonal classification criteria (of which the first one is the most important distinctive criterion):

- *instance-based/schema-based* matching: A schema-based matcher uses different approaches to determine correspondence between ontologies based on concepts and relations [**?**]. An instance-based matcher takes instances (i.e. instance data) belonging to the different concepts in the ontologies and uses these to discover similarities between the concept.
- *element-level/structure-level* matching: an element-level matcher takes individual schema elements, properties of the particular concept or relation, and uses these to find similarities, whereas a structure level matcher compares the combination of elements, the structure (e.g. the concept hierarchy) of the ontologies to find similarities.

- *language/constraint*: a matcher can use a linguistic approach (e.g. based on names and textual description of schema elements) or a constraint-based approach (e.g. based on keys and relationships).
- *matching cardinality:* an overall matching result can relate one or more elements of an ontology to one or more elements of the other, giving four cases: 1:1, 1:n, m:1 and n:m. In addition each mapping element can may interrelate one ore more elements of the given ontologies. Furthermore, there may be different cardinalities at the instance level.
- *auxiliary information*: most matchers rely on external information besides the two given ontologies, such as global schemas, dictionaries, previous matching decisions and user input.
- *accepted input:* this dimension concerns the type of input on which the algorithms operate, classified depending on the data/conceptual models they accept.
- *produced output*: this pertains to the way the mapping is described, whether it is a one-to-one correspondence or a more general one, whether it is a graded answer (98% or 4/5), or an all-or-nothing approach, whether the result focuses only on equivalence between entities, or can it give a more expressive result, and so on.

Schema-based matching. A schema-based matcher [**?**] takes different aspects of the concepts and relations in ontologies and uses some similarity measure to determine correspondence. The available information includes the usual properties of ontology elements, such as name, description, data types, relationship types (part-of, is-a, etc.), constraints and ontology structures. In the following we discuss some of the main aspects of schema-based matching.

- *Granularity of match*. We distinguish two main alternatives, element-level and structure-level matching. For each element of the fist ontology, an *element-level matcher* determines the matching element of the second ontology at the same granularity level (usually at the atomic level). For instance, in two ontology fragments describing addresses we could find that ‘‘`Address.ZIP`” from one ontology matches “`CustomerAddress.PostalCode`’’ from the other one. *Structure level matching*on the other hand refers to matching combinations of elements that appear together in a structure. This can be a complete, or a partial match. Known equivalence patterns can be kept in a library. For example (this conforms to a common equivalence pattern), one ontology could have a class `ParttimeEmployee`, which is

the subclass of `Employee`, while another could have a class `Employee` having the attribute `IsParttime`. A structure-level matcher should identify these as being identical, when the `IsParttime` property equals to true.

- *Match cardinality*. An element can participate in zero, one or many mapping elements of a match result. Moreover, within a mapping element, one or more elements from one ontology can match one or more elements from the other one. Thus we have the usual relationship cardinalities of 1:1, n:1, 1:m and n:m between matching elements, both with respect to different mapping elements (*global cardinality*) and with respect to an individual mapping element (*local cardinality*).
  Most existing matchers map each element of one schema to one element of the other schema. This results in local 1:1 mappings and global 1:1 or 1:n mappings.

- *Linguistic approaches*. Linguistic matchers use name and text (i.e. words or sentences) to find semantically similar elements. *Name matching* matches elements with identical or similar names (using synonyms, common substrings, user-provided name matching, etc.). *Description matching* uses natural language comments of ontology elements. For example we can have "`empn //employee name`" in an ontology matching "`name //name of employee`" in the other. This can be done by simple keyword extraction, or complex natural language processing.

- *Constraint-based approaches.* These approaches use the fact that ontologies often contain constraints on data types and ranges, relationships, optionality, uniqueness, cardinalities, etc. This information can be used to determine similarities between ontology elements. For example, in a candidate pair having the fields `Born` and `BirthDate` respectively, both having the type `Date`, a constraint-based matcher could find these to be similar.

- *Reusing schema and mapping information*. In addition to already mentioned auxiliary information, matchers could well reuse common ontology components and previously determined mappings. Some mappings occur very often in the same domains, and reusing these increases effectiveness. For example, in e-commerce domains substructures often repeat with different message formats (like address, name, etc.), so we could keep a library of previously found results on these, so when we find two elements labelled `ProductOrder` and `Product` in an ontology `O1`, and `Porder` and `Article` in an ontology `O2`, the matcher

would already know that these match (`ProductOrder` matches `POrder` and `Product` matches `Article`).

Instance-based matching. An instance-based matcher [**?**] takes instances (i.e. instance data) belonging to the different concepts in the ontologies and uses these to discover similarities between the concepts. It can be useful when the data is semistructured or particularly useful to uncover incorrect interpretations of schema information.

The approaches we discuss for instance-level matching primarily work in finding element-level matches, because of the complexity of comparing large numbers of combinations of instances.

Most of the approaches of schema-level matchers can be applied here, but some are especially applicable:

- For text elements, a linguistic approach based on information retrieval techniques is the preferred approach. For example, if we have a field called `Dept` in an ontology, and two fields `DeptName` and `EmpName` in another, instance information may help to deduce that `DeptName` is the primary match candidate for `Dept`.
- For more structured data, such as numerical or string elements, we can apply a constraint-based characterisation, such as numerical value ranges or character patterns. For instance, this may help recognising phone numbers, postal codes, birth dates, ISBNs, money-related entries (e.g. based on currency symbols), etc.

The results of instance-based matchers can be used to enhance schema-level matchers as a first approach. For instance, a constraint-based schema-level matcher can more accurately determine data types and ranges for an element using the results of instance-based matching. A second approach is to perform instance-level matching on its own. Matching elements from the first ontology to the second one, and then vice-versa, we can obtain schema-level matches. Instance-level matching can also be performed by using auxiliary information. This is especially helpful for matching text elements by providing match candidates for individual keywords. For example, a previous analysis may have revealed that the keyword ‘‘`HP`’’ frequently occurs for ontology elements ‘‘`CompanyName`’’, ‘‘`Manufacturer`’’, etc. For a new match task, if an `O2` ontology element `X` frequently contains the term ‘‘`HP`’’, this can be used to generate ‘‘`CompanyName`’’ in `O1` as a match candidate for `X`, even if ‘‘`HP`’’ does not often occur in instances of `O1`.

D.2.4 Overview of Approaches

In this section we give a brief overview of some of the main approaches to ontology alignment.

Anchor-PROMPT. Anchor-PROMPT [**?**] uses a set of heuristics to analyse non-local context, as opposed to algorithms like PROMPT [**?**] (See Section **??**) and Chimaera, that only analyse local context. It doesn't provide a complete solution, but augments existing methods.

The algorithms takes as input two pairs of related terms (anchors) from the source ontologies. These anchors are either given by the user, or generated automatically with string-based techniques or another matcher using linguistic similarity.

From this set of anchors, the algorithm produces a set of new pairs of semantically close terms. To do that, Anchor-PROMPT traverses the paths between the anchors in the corresponding ontologies. A path follows the links between the classes, defined by the hierarchical relations or by slots and their domain and range. Anchor-PROMPT then compares these terms along the paths to find similar terms. Terms with a high similarity score are presented to the user to improve the set of possible suggestions, for example, a merging process in PROMPT.

For example, suppose we have ontology `O1` having the classes `A-C-E-H` in a hierarchical chain, and `O2` having the classes `B-D-F-G` in its hierarchy. Then, suppose we identify the anchors as `A-B` and `G-H`. We then traverse each path in parallel, incrementing the similarity score of the classes we encounter at each step. We repeat the the process for all existing paths between the anchor points, cumulatively aggregating the similarity score.

The central observation behind Anchor-PROMPT is that if two pairs of terms are similar and there are paths connecting the terms, the elements along those paths are often similar as well.

GLUE. GLUE [**?**] applies machine learning techniques to semi-automatically create semantic mappings between heterogeneous ontologies based on instance data. It regards ontologies as being taxonomies of concepts, and focuses on finding 1-to-1 correspondences between the concepts of the ontologies: for each concept node in one taxonomy, find the most similar concept node from the other taxonomy.

Instead of committing to a certain similarity definition, it computes the *joint probability distribution* of the concepts involved, and lets the application use the joint distribution to compute any suitable similarity measure. Specifically, for any two concepts `A` and `B` the joint probability distribution

consists of P(A,B), P(A,¬B), P(¬A,¬B), and P(¬A,B), where a term such as P(A,¬B), is the probability that an instance in the domain belongs to concept A, but not to concept B.

Computing this joint distribution is based on the sets of instances that overlap between the two concepts. A term such as P(A,B) can be approximated as the fraction of instances that belong to both A and B, that is we need to decide for each instance whether it belongs to AΠB. GLUE addresses this by using the instances of A to learn a classifier for A. Then classifies instances of B according to that classifier, and vice-versa.

Semantic Matching. Semantic Matching [**?**] is an implementation of the Match operator, that takes two graph-like structures (like database schemas or ontologies) and produces a mapping between elements of the two graphs that correspond semantically to each other. The authors argue that most previous approaches used syntax driven techniques, like matching labels(substrings, abbreviations, similar soundex) or syntactical structure and calculating a similarity measure, all being different variations of syntactic matching, that searches for semantic correspondences based on syntactic features.

Semantic matching has the following main features:

- searching is done by mapping meanings (concepts), not labels. It is not sufficient to consider the meaning of labels in the nodes, but also the positions the nodes have in the graph.
- semantic similarity relations are used between elements (concepts) instead of syntactical similarity relations. In particular, those relations are considered, which relate the extensions of the concepts under consideration.

The semantics of a node is given by the concept attached to that node (the concept denoted by the label of the node), by the position of the given node in the graph, and the semantics of all the nodes which are higher in the hierarchy.

The possible returned relations between elements are equality, overlapping, mismatch, and more general/specific.

QOM. QOM [**?**] was designed to create an efficient matching tool for on-the-fly creation of mappings between ontologies.

Given that the a major ingredient of run-time complexity is the number of mapping pairs which have to be compared to actually find the best mappings, QOM uses heuristics to lower the number

of candidate mappings, thus not comparing all entities of the first ontology to all entities of the second ontology. It uses the ontological structures to classify candidate mappings into promising and less promising pairs, using a dynamic programming approach to refine the set of candidate mappings in each iteration.

The similarity measure is computed using a wide range of similarity functions, such as string similarity. Several of such similarity measures are computed, which are all input to the similarity aggregation function, which computes the actual similarity measure. QOM then applies a sigmoid function, which emphasises high individual similarities and deemphasises low individual similarities. The actual correspondences are then extracted using a threshold to the aggregated similarity measure.

The output of one iteration of QOM can be used as part of the input of a subsequent iteration in order to refine the result. After a number of iterations the actual mapping between the ontologies is obtained.

D.2.5 Alignment Examples

This section provides a few examples of the matching process' results.

Table 2: Constraint-based matching example.Taken from [**?**].

S1 elements	S2 elements
Employee	
EmpNo – int, primary key	
EmpName – varchar (50)	Personnel
DeptNo – int, references	Pno - int, unique
Salary – dec (15,2)	Pname – string
Department	Dept – string
Birthdate – date	Born – date
DeptNo – int, primary key	
DeptName – varchar (40)	

The two schemas (corresponding to ontologies) shown in Table **??** present an example of constraint-based schema-level matching [**?**]. Type and key information suggest that `Born` matches `Birthdate` and `Pno` matches either `EmpNo` or `DeptNo` (this could be refined using instance-based matching, which could reveal that `Pno` matches `EmpNo`). By using a matcher that detects more than just atomic-level similarities, we match `S2.Personnel` to `S1.Employee` joined with `S1.Department`. This can be detected automatically by observing that elements of `S2.Personnel` match elements of `S1.Employee` and `S1.Department` and that `S1.Employee` and `S1.Department` are connected by the foreign key `DepNo`. This allows us to determine the correct SQL-like n:m mapping:

```
S2.Personnel (Pno, Pname, Dept, born) ~
SELECT S1.Employee.EmpNo,
       S1.Employee.EmpName,
       S1.Department.DeptName,
       S1.Employee.Birthdate
FROM S1.Employee, S1.Department
WHERE (S1.Employee.DeptNo = S1.Department.DeptNo)
```

Table **??** [**?**] illustrates match cardinalities for two given schemas S1 and S2. When matching multiple elements, we can see that expressions are used to combine them.

Table 3: Matching cardinalities example.Taken from [**?**].

Local match cardinalities	**S1 element(s)**	**S2 element(s)**	**Matching expression**
1:1, element level	Price	Amount	Amount = Price
n:1, element-level	Price, Tax	Cost	Cost = Price*(1+Tax/100)
1:n, element-level	Name	FirstName, LastName	FirstName, LastName = Extract (Name, . . .)

n:1 structure-level (n:m element-level)	B.Title, B.PuNo, P.PuNo, P.Name	A.Book, A.Publisher	A.Book, A.Publisher = SELECT B.Title, P.Name FROM B, P WHERE B.PuNo=P.PuNo

In the first row, the match is 1:1, which is the most simple to determine. Row 2 shows how `Cost` can be matched to a formula based on `Price` and `Tax`. Row 3 explains how `FirstName` and `LastName` are extracted from `Name`. Row 4 uses a SQL expression combining attributes from two tables. It corresponds to an n:m relationship at the attribute level (four `S1` attributes correspond to two `S2` attributes) and to an n:1 relationship at the structure level (two tables from `S1` match one table from `S2`).

The example in Table **??** [**?**] shows a mapping element from the motivating example from Section **??**. Assume we have two ontologies `O1` and `O2` which both describe humans and their gender. Ontology `O1` describes the concept `Person` with an attribute `hasGender`, which has the two possible values "`male`" and "`female`". Ontology `O2` describes the concept `Human`, and its subclasses `Man` and `Woman` to distinguish the gender.

Table 4: Class by attribute mapping pattern and example.Taken from [**?**].

Name: Class by Attribute Mapping
Problem: The extension of a class in one ontology corresponds to the extension of a class in another ontology, provided that all individuals in the extension have a particular attribute value.
Solution: *Solution description*: A mapping is established between a class/attribute/attribute value combination in one ontology and a class in another ontology. *Mapping syntax:* `mapping ::= classMapping(direction A B attributeValueCondition(P o))`
Example: `classMapping(Human Female attributeValueCondition(hasGender` "female"))

Notice that this is a typical case of a conceptual mismatch, namely a mismatch in the style of modeling. The solution illustrates an elementary mapping pattern. The pattern is described in terms of its name,the problem addressed, the solution of the problem, both in natural-language description and in terms of the actual mapping language, and an example, namely a mapping between a class `Human` from ontology `O1` to the class `Woman` from ontology `O2`, but only for humans having the gender "`female`".

D.3 Ontology Merging

Ontology merging is the creation of a new ontology from one or more source ontologies. The new ontology will unify and in general replace the original source ontologies.

We distinguish between two main approaches to ontology merging. In the first approach the input of the process is a collection of ontologies and the output is a new, merged ontology which captures the original ontologies. A prominent example of this is PROMPT [**?**]. In the second approach the ontologies are not replaced, but rather a 'view', called *bridge ontology* is created which imports the original ontologies and specifies the correspondences using bridge axioms. OntoMerge [**?**] is an example of this approach, creating a bridge ontology that imports the original ontologies and relates the overlapping concepts in these ontologies using bridge axioms. We

describe PROMPT and OntoMerge in more detail below.

D.3.1 PROMPT

PROMPT [**?**] is an algorithm and interactive tool for ontology merging and alignment. It was one of the first ontology merging tools, comparing every pair in the two ontologies and looking for syntactic similarity between labels. It has been implemented as a Protegé-2000 extension.

The algorithm of PROMPT defines a number of steps for the interactive merging process:

1. An initial list of merging candidates is created, based on class name similarities. The list is presented to the user as potential merging operations. Then the following cycle happens:
2. The user chooses the next operation: selects an option suggested by the algorithm, or edits the ontology to specify the desired operation directly.
3. The third step is composed of the following three sub-steps:
   (a) The system performs the requested action and automatically executes additional changes derived from the operation.
   (b) A new list of suggested operations is presented to the user, based on the structure of the ontology around the arguments of the last operation.
   (c) The system determines the conflicts introduced by the last operation and finds possible solutions to them.

PROMPT defines the following set of ontology-merging operations: merge classes, merge slots, merge bindings between a slot and a class, perform a deep copy of the class (copy all parents up to the root), perform a shallow copy of the class.

The conflicts that may appear as a result of these operations are: name conflicts, dangling references, redundancy in the class hierarchy, slot-value restrictions that violate class inheritance.

The result of the algorithm is a new ontology which replaces the original ones.

D.3.2 OntoMerge

OntoMerge [**?**] is an online ontology translation system, which translates a dataset to a new dataset which captures the same information in a different ontology.

The approach divides the process into two main parts, namely separating the syntactic translation from the semantic translation:

1. Converting the datasets into a uniform internal representation (a language called Web-PDDL) in order to clear away syntactic differences.
2. Perform the semantic translation from the internal representation of a dataset in the source ontology to the internal representation of the a dataset in the target ontology.

The result of the process is not an ontology replacing the original ones as in PROMPT, but a bridge ontology that imports the source ontologies and contains a set of Bridging Axioms, which are translation rules used to connect the overlapping parts of the source ontologies. The two source ontologies, together with the set of bridging axioms, are then treated as a single theory by a theorem prover (optimised for operations like dataset translation, ontology extension generation based on the extension of a related ontology, and query rewriting).

A Lists of Tables and Figures

List of Tables

List of Figures