# BRIDGING THE GAP BETWEEN THE COGNITIVE ENGINE AND THE SDR

Jakub Moskal (Northeastern University, Boston, MA, USA; jmoskal@ece.neu.edu); Mieczysław M. Kokar (Northeastern University, Boston, MA, USA; mkokar@ece.neu.edu); Shujun Li (Northeastern University, Boston, MA, USA; shli@ece.neu.edu)

## ABSTRACT

Regardless of the type of an inference engine that a Cognitive Radio (CR) employs, each CR implementation requires access to the SDR's Knobs & Meters (K&M) in order to achieve self-awareness. Due to the lack of a standard K&M API, current CR architectures rely either on APIs provided by concrete SDR platforms, or specify arbitrary APIs that are not standardized by any standards organization. This leads to the situation in which existing CR architectures are rather tightly bound to a chosen SDR platform supported by a (usually large) business unit. Instead of relying on any specific SDR API, in our previous work we proposed a thin and generic interface between a reasoner and a SDR. In this approach the reasoner could access K&M of an SDR using only abstract, ontological terms. In this paper we show how the ontological terms are mapped to SDR-specific method invocations and show how the ontology-based interface could be used in different Cognitive Engine implementations and lead to CR architectures that are less dependent on the underlying software interfaces.

## 1. INTRODUCTION

In our previous work [1,2] we conducted a review of existing CR architectures with the focus on the interface that is employed between the reasoner and the SDR. Virtually all of the existing CR designs rely on non-standard domain-specific interfaces. In this approach (shown in Figure 1) a dedicated piece of code (controller) is responsible for invoking the radio's API and supplying the reasoner with facts representing the current state of the radio's operational behavior. Optionally, the API is available via CORBA to achieve platform-independence. Design of such controller requires design-time knowledge about the ontology, for the values returned by the API methods must be correlated with specific ontological terms. Moreover, the reasoner itself must be extended with API-specific procedural attachments that allow it to dynamically change the values of radio's parameters from within the rules.

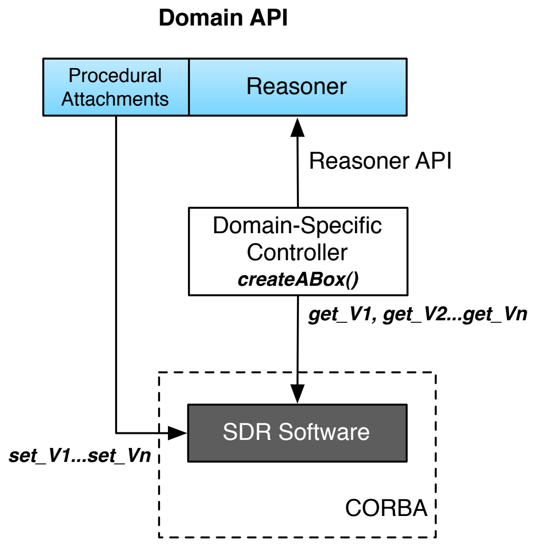

**Figure 1 Domain-Specific API approach**

The domain-specific API approach suffers from several issues, most importantly 1) it requires maintenance of the API-dedicated code each time the API changes, 2) it requires implementation and maintenance of adapters for radios that don't implement the API expected by the controller and the procedural attachments, 3) the API may become a bottleneck of the design when API changes are avoided to maintain compatibility with legacy components. Because there is no standard way for accessing radio's parameters, existing architectures that rely on radio-specific interfaces ultimately support only a few SDRs, or are tied to a single specific platform.

While CR architectures still lack industry support, SDR community has made use of well-defined standards, primarily the Software Communications Architecture (SCA) [3], which has been widely supported by both industry and the academia. The SCA itself does not define architecture for software radio; rather it provides hardware abstraction layer and the software infrastructure to develop waveform software on top of a radio system. The SCA design provides a great deal of independence for the waveform developers. Not only waveform applications once written can be theoretically ported to any SCA-compliant hardware, but the developers can also develop software in many programming languages, as long as the languages support CORBA.

Unfortunately, the SCA does not provide any support for the CR out of the box — there is no dedicated API for the cognitive software, thus there is no standard–supported way to realize the CR architecture on top of the SCA-based SDRs. Instead, one must treat every SCA-compliant SDR as a separate SDR, with its own specific API. One solution to this problem is to include cognitive capabilities in the application layer along with the waveform software. However, such a solution would be most likely proprietary and constrained by the existing APIs, available to the waveform developers, probably too generic to be used for this purpose [4]. Wellington suggests [4] that the cognitive software be implemented as a SCA-compliant component and interact with the waveform components by the means of a set of new interfaces.

However, the problems mentioned above would persist even if the current set of APIs provided by the SCA was augmented by a new interface dedicated to the cognitive engine functionality. In fact, the SCA–based CR architecture would resemble that of the aforementioned Domain API (c.f. Figure 1), facing the same issues described above. While this design would be superior to the one of the Domain API, because it would have the industry support behind it, it would only add the platform-independence. In addition, it would make major changes to the API even more expensive, because not only they would require updating the implementation, but also a new SCA compliance certification.

In our previous work we introduced the concept of a *generic* API that uses a LiveKB component and addresses the issues associated with the Domain API. In this paper, we discuss the details of how the LiveKB provides a generic access to a virtually unbounded number of radios and thus bridges the gap between the reasoner and the SDR.

## 2. NEED FOR A GENERIC API

The original idea behind designing the LiveKB component was to allow the reasoner to express its requests to read and write radio's parameters exclusively in ontological terms. These terms are shared by all radios through a standard ontology and do not pertain to any specific API. Instead of having a number of methods that correspond to different parameters, we would like to be able to use only two of them:

```
get(propertyName)
set(propertyName, newValue)
```

For instance, if `hasTxAmplitude` and `hasCarrierFrequency` are datatype properties defined in the CR ontology, in order to get or set a value of these parameters in a radio, instead of invoking radio API-specific methods like `get_txAmplitude()` and `setCarrierFrequency(2400)` we could invoke the following: `get("hasTxAmplitude")` and `set("hasCarrierFrequency",2400)`, respectively.

In a sense, the CR ontology becomes the standard in this scenario. However, because it represents domain knowledge, rather than a programming interface, it is far less likely to change in the future than API. What is more important is the fact that the generic API allows for writing rules (policies) that are reusable, because the procedural attachments corresponding to get and set methods, used within the rules, are independent of the SDR software structure. Moreover, changes made to the ontology would only require changes in the rules, leaving the implementation of API intact.

The generic API requests could be processed in at least two ways: 1) directly invoked on the radio, or 2) first *translated* to radio-specific methods and then invoked on the radio. The first approach would impose a substantial requirement on each radio, because the ontology would have to be known at design time and become part of the implementation. This defeats the purpose of a generic API, because radios would need to recode their interface each time ontology changes. The second approach allows the radio to provide its own API, yet keep the rules reusable. The crucial part of this design is translation, or mapping, from generic to radio-specific methods. A straightforward solution, similar to the ones used in most of the reviewed architectures, would be to define a standard API for all radios, then implement a layer of code that translates between get/set to the standard. As we indicated in the previous work, there are multiple problems with a standard API, such as a lack of consensus, a slow rate of changes, and problems with backwards compatibility. This is where the LiveKB component comes into play – instead of translating generic requests using a radio API, it does so *dynamically* at run-time using reflection, regardless of what API the radio provides.

The difference between invoking radio methods using its API and invoking them via LiveKB is shown graphically in Figure 2. Note that when using LiveKB, there is no need to know anything about the radio's API on the reasoner side, and at the same time the radio can implement its own API. The benefits of this design are twofold – the reasoner can theoretically access any radio, and the radios do not have to implement any standard API. In the next section we provide details of how the LiveKB is designed in order to support the dynamic mapping between the generic and SDR-specific API.

## 3. ONTOLOGY MATCHING

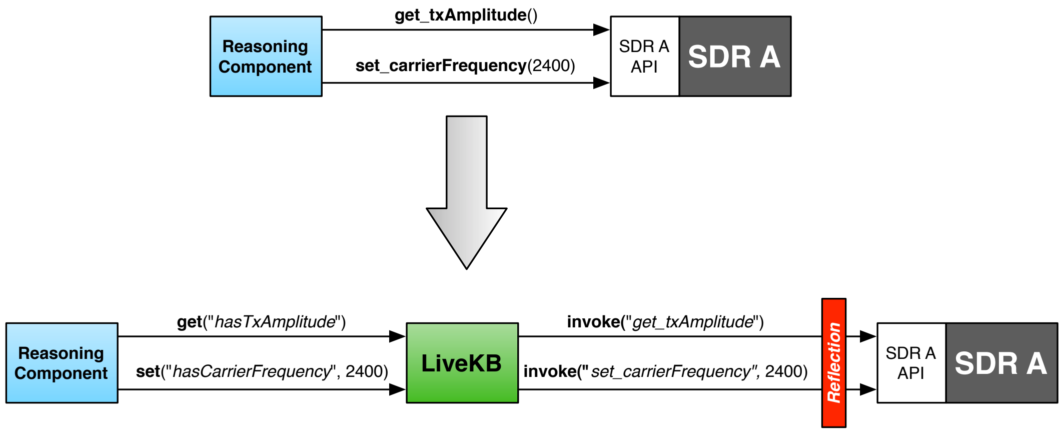

**Figure 2 Accessing SDR using the LiveKB component**

The primary goal of LiveKB is to dynamically translate requests that come from the reasoner to method invocations that are specific for the radio. The fundamental concept for implementing this goal is *ontology matching*.

Ontology matching is defined as "the process of finding relationships or correspondences between entities of different ontologies". The output of matching, called alignment, is a set of correspondences that express the relationship between two ontologies. Alignments include, but are not limited to, statements such as entity equivalence, sub-super relationship between entities, class intersection, or inverse relation. Alignment can be used to generate tools used for further automated processing, such as a *translator* for translating data instances between two different ontologies, or a *mediator* that can translate queries expressed in one ontology to another, and translate answers in the opposite direction.

Despite the use of sophisticated methods from AI, ontology matching can rarely be fully automated beyond relatively simple syntactic correspondences. When complex conceptual relationships come into play, matching algorithms often have difficulties identifying any correspondences at all, or find ones that are irrelevant. When the matching is incomplete, finishing the alignment manually is necessary, although even this task can be cumbersome. The manual alignment can be facilitated with the use of ontology alignment design patterns [5], which stem from the observation that different sets of ontologies, even from completely different domains, exhibit similar types of complex relationships. Design patterns are particularly useful for finding solutions to complex relationships, e.g. a property in one ontology has the same intention as a relation in the second ontology, which requires transforming data values into specific class individuals.

Expressive and Declarative Ontology Alignment Language (EDOAL) [6] is a language designed by the ontology matching community specifically to address the problem of expressing complex relationships between ontologies. The semantics of EDOAL is independent from any ontology language, which has two benefits – it can be used to match ontologies grounded in different syntax, and it allows for expressing design patterns at an abstract level.

EDOAL would certainly be of lower value if it was not accompanied by Alignment API [7], a comprehensive Java API for manipulating alignments. Alignment API was created around the time the first predecessor of EDOAL was designed. It aims to cover functionality related to the ontology matching process as a whole by providing abstractions for matchers, evaluators, renderers and parsers. Using the API, alignment documents can be parsed and then rendered to generate XSLT scripts, OWL axioms, etc. In its most recent version [8], support for processing EDOAL was added, although included renderers are fairly limited at this moment, e.g. the OWL axioms renderer does not take advantage of any of the features added in OWL 2 [9]. This limitation can be addressed by implementing custom renderers.

Although complete automatization of the ontology matching process still has a long way to go – and perhaps can never be fully realized – EDOAL and Alignment API form a solid platform for realizing matching use cases.

## 4. LIVEKB DESIGN

LiveKB uses methods and tools from the ontology matching research area in order to map the generic API requests to SDR-specific invocations. Figure 3 shows the ideal design of LiveKB. SDR, accessible via CORBA, provides its API as an IDL, which is automatically translated into its equivalent IDL ontology expressed in OWL. The Matcher matches the generated IDL ontology with the CR ontology and produces alignment. The alignment is passed to Generator to generate a mediator. At runtime, the mediator

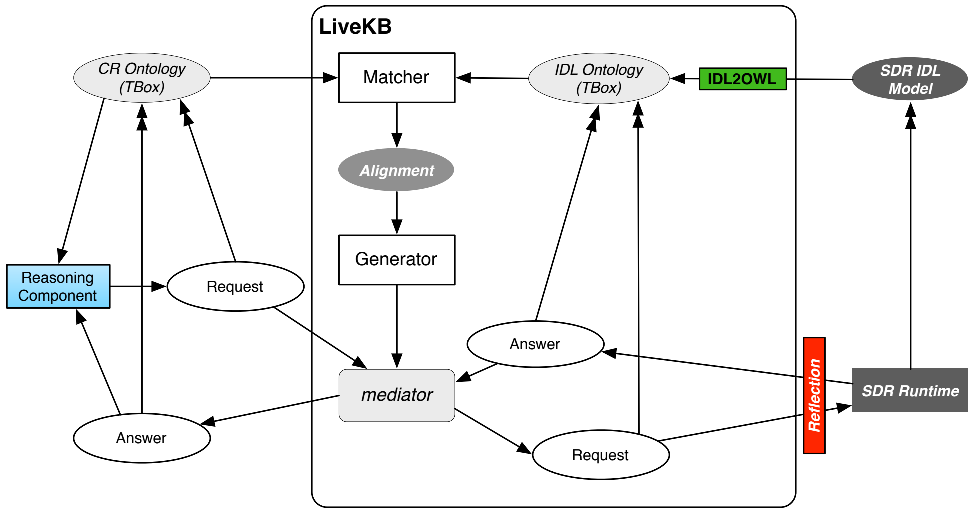

**Figure 3 Ideal design of LiveKB**

translates the requests coming from the reasoner to requests expressed in the IDL ontology. The axioms in the IDL ontology provide sufficient information to reflectively invoke the requested method on the SDR. Once an invocation is complete, the mediator translates the result of the invocation into terms of the CR ontology and sends it back to the reasoner.

The first three steps – generation of the IDL ontology, matching it with the CR ontology and generation of the mediator from the alignment, need to be done only once, at startup. After that, LiveKB can translate reasoner's requests by utilizing the artifacts produced earlier, and mostly performs reflective invocations.

The realization of the ideal design of LiveKB is very challenging since it depends on automatization of the three somewhat complex tasks: 1) translation of an IDL model into an OWL ontology, 2) ontology matching between IDL and CR ontologies, and 3) generation of a mediator from the alignment. The first task can be automated, as long as the IDL respects some constraints (we will discuss this later). However, as explained above, ontology matching is not ready to be fully automated for complex correspondences. As a consequence, manual or semi-manual matching is necessary to be done for each radio. Furthermore, since this field is relatively young, the tool generators for EDOAL alignments are still immature and thus this process also cannot be fully automated. Consequently, at this point we cannot fully realize the LiveKB design as depicted in Figure 3. Nonetheless, given the progress done each year in the field of ontology matching, it is anticipated that an automatic ontology matching will be realizable to a higher degree in the near future.

## 4.1. Feasible design of LiveKB

Since the matching cannot be fully automated yet, it needs to be done manually, or somehow *assisted* to produce a full alignment. In order to facilitate this process, we altered the LiveKB design and made it feasible for implementation. It allows for dynamic translation between generic and radio-specific API, but requires more input from the SDR vendor. The revised, more feasible LiveKB design is shown in Figure 5. In this design, not only SDR must be available via CORBA and provide its IDL, but the IDL must also be *annotated* to aid the matching process. The annotated IDL provides enough information for the Assisted Matcher to create a full alignment between the given CR ontology and the IDL ontology generated within LiveKB from the SDR IDL model. The matcher also generates an Invoker, which can execute SDR methods represented as properties in the IDL ontology with the use of the reflection mechanism.

Since the generation of tools solely based on alignment is still limited, we use alignment only to generate bridge axioms [10], which merge the two ontologies together. Bridge axioms can be easily generated, because they correspond to the alignment almost in a one-to-one fashion. Using rules and bridge axioms, requests formulated by the reasoner in terms of the CR ontology can be automatically translated into terms of the IDL ontology. A request expressed in the IDL ontology provides sufficient information for the Invoker to locate an object in the SDR runtime and execute an appropriate method. Invoker can also read all the parameters and represent a radio's current state as a collection of CR ontology ABox assertions.

If we look at LiveKB as a black box, it needs to be provided with (1) an annotated IDL model of the SDR and (2) the CR ontology. At bootstrap, it produces (1) an IDL ontology and (2) bridge axioms that need to be loaded into the reasoner's Knowledge Base. At runtime, it can produce a CR ontology ABox, which represents the SDR's current

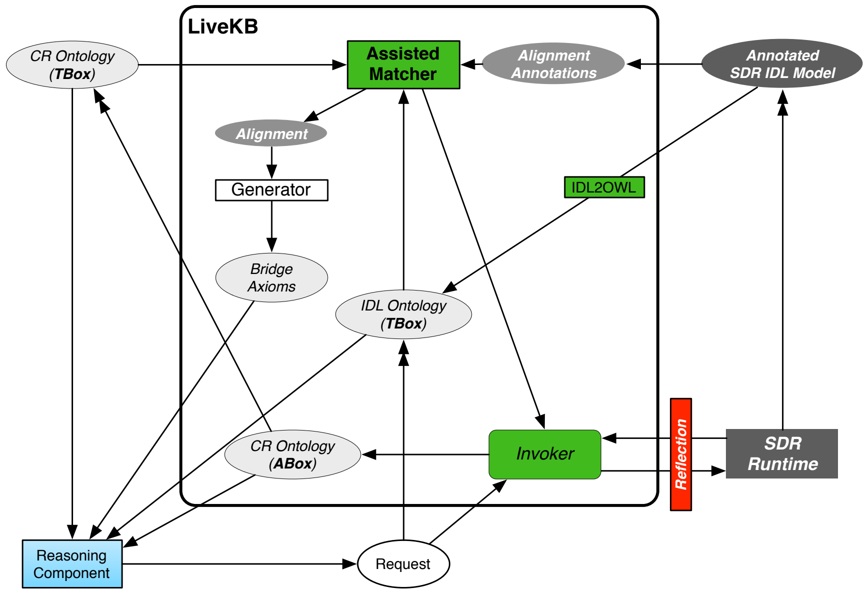

**Figure 5 Feasible design of LiveKB**

state and can respond to get/set requests invoked from within the reasoner's rules. A simple rule that invokes a setter to change a value of a parameter, given that some condition is met, is shown below using pseudocode:

```
if $k > k\_max$ then
    setter <- find a setter property in IDL ontology that is equivalent to $k$ in the CR ontology
    INVOKE(setter, newValue)
end if
```

Note that the rule writer is not required to know the name of the setter property in the IDL ontology – it is found by the reasoner using the bridge axioms.

The feasible design of LiveKB involves generation of three artifacts: an IDL ontology, alignment axioms and an Invoker tool. Since the explanation of how these artifacts are generated goes beyond the scope of this paper, we refer the reader to [11] for more details.

Using the LiveKB component, one can design a CR architecture in a way that does not depend on SDR-specific API and such is far less vulnerable to changes and can support numerous radio platforms. Figure 4 shows how LiveKB can be utilized within a CR architecture. Note that both the controller and procedural attachments no longer depend on SDR-specific method invocations, and as such remain intact regardless of changes made in the SDR API. Another great benefit of this design is the fact that the rules can be written once and executed on multiple platforms, as long as they use LiveKB to access radio's parameters.

## 4.2. LiveKB API

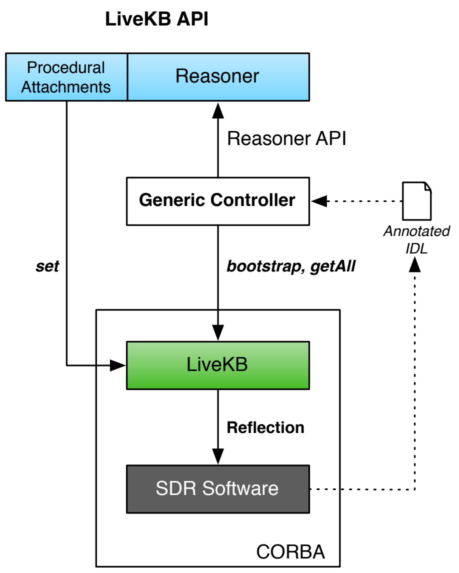

**Figure 4 LiveKB-based CR design**

Since LiveKB is available via CORBA, we present its API in IDL:

```
module livekb {
 interface LiveKB {
  string getAll();
  any get(in string property);
  void set(in string property, in any value);};

 interface LiveKBFactory {
  LiveKB getInstance(in string model,
    in string rootName, in string ontology);
 };};
```

Before LiveKB can be used, it must be instantiated by the LiveKBFactory. Reference to the LiveKBFactory implementation can be found in the CORBA Naming Service using a name specified in the LiveKB configuration file. Once a reference to the object factory is obtained, it can be used to create an instance of LiveKB by providing it with the annotated IDL model, the name of the SDR's root object, and the CR ontology.

This API does not contain any SDR-specific information. In fact, it does not contain terms specific to any domain. LiveKB API is more abstract than traditional APIs because it does not constrain the reasoner to a fixed number of radio-specific parameters selected during the design time. This feature of the CRF architecture allows the rules to be fully reusable. The hard-coded part of the interface is oblivious to the name of parameters or methods used to access them. The radio-specific information, instead, is used when *creating* the LiveKB component.

### 4.3. LiveKB Implementation

LiveKB was implemented in Java since this language offers a solid reflection mechanism, which is crucial for its successful implementation. During bootstrap, LiveKB generates Java stubs from the IDL description using an external tool. Once generated, stubs are dynamically compiled and loaded into the Java Virtual Machine (JVM). The last step is necessary for the reflection mechanism to work, because it requires class descriptions to be loaded in the same JVM where LiveKB is located. Finally, the root object reference is retrieved from the Naming Service.

### 4.4. Limitations

The generic nature of LiveKB does require that SDRs meet certain requirements:

1. SDR parameters are accessible via CORBA
2. The runtime-objects form a tree-like structure and the reference to the root is available via CORBA Naming Service
3. SDR implements IDL that respects the following constraints:
   a. Operations that are used to access knobs and meters have either (1) no parameters, or (2) one **in** parameter, or (3) one **out** parameter. In case of (1), the return type must be primitive. In cases (2) and (3), the return type must be **void**.
   b. Operations that are used to access knobs and meters are properly annotated (beyond the scope of this paper).
   c. IDL annotations combined together form a "proper" ABox (beyond the scope of this paper).

Given such constraints, CRF may have no support for some legacy SDRs, however, if an SDR is already available via CORBA and it follows the above requirements, it can be accessed by LiveKB out-of-the box. Note that LiveKB can interface numerous SDRs without requiring them to implement a specific API and without the need to implement an interface-dedicated code. Moreover, when radios change their APIs over time, as long as they still support the above requirements, they can be interfaced by LiveKB, even if the new version is not backwards-compatible.

## 5. CONCLUSIONS AND FUTURE WORK

LiveKB has been successfully implemented and used to access different radios built on top of the GNU Radio framework and executed on the USRP1 platform. Changes made to the ontology had to be reflected in the rules, but did not require any recoding of the controller. Changes made to the radio API required adjusting the IDL annotations, but also did not require any recoding of the interface between the reasoner and the radio.

The benefits of using LiveKB interface to access domain software's parameters are: support for knowledge reusability and exchange, significantly smaller effort required to adapt to changes, inherent platform-independence. The drawbacks of using LiveKB include the requirement for using CORBA, increased number of triples in the reasoner's KB due to addition of the bridge axioms, slightly longer rules and slower bootstrap. LiveKB offers great benefits when the APIs are not well standardized and are likely to change in the future. This certainly applies to the wireless domain, which is a very active area of research and new capabilities are likely to be reflected in new APIs.

As part of the future work, we will aim to improve the matching algorithm in order to support creating bridge axioms without requiring IDL annotations, and remove or decrease the limitations posed on the IDL models implemented by the SDRs.

## 6. REFERENCES

[1] J. Moskal, and M.M. Kokar, "Interfacing a reasoner with an SDR: A platform and domain API independent approach", *SDR Technical Conference,* Dec 2009

[2] J. Moskal, M.M. Kokar, and S. Li "Interfacing a reasoner with an SDR Using a Thin, Generic API: A GNU Radio Example", *SDR Technical Conference,* Dec 2010

[3] JTRS, Software Communications Architecture Specification Version 2.2.2. Joint Program Executive Office (JPEO), May 15, 2006. Available at http://sca. jpeojtrs.mil/.

[4] R. J. Wellington, "Cognitive policy engines," in Cognitive Radio Technology (B. A. Fette, ed.), ch. 6, pp. 195–222, Elsevier, 2nd ed., 2009.

[5] F. Scharffe, Correspondence Patterns Representation. PhD thesis, University of Insbruck, 2009.

[6] J. Euzenat, F. Scharffe, and A. Zimmermann, "Expressive alignment language and implementation," deliverable,

Knowledge Web NoE, 2007. Available at http:// ftp//ftp.inrialpes.fr/pub/exmo/reports/kweb-2210.pdf.

[7] J. Euzenat, “An api for ontology alignment,” in The Semantic Web ISWC 2004 (S. A. McIlraith, D. Plexousakis, and F. van Harmelen, eds.), vol. 3298 of Lecture Notes in Computer Science, pp. 698–712–712, Springer Berlin / Heidelberg, 2004.

[8] J. David, J. Euzenat, F. Scharffe, and C. Trojahn dos Santos, “The Alignment API 4.0,” Semantic Web, 2011.

[9] W3C, OWL 2 Web Ontology Language Primer. W3C Recommendation, 2009. Avail- able at http://www.w3.org/TR/owl2-primer/.

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

[11] J. Moskal “Interfacing a Reasoner with Heterogeneous Self-Controlling Software”, 2011.