An Architecture for Information Exchange Based on Reference Models

Heiko Paulheim[1], Daniel Oberle[2], Roland Plendl[2], and Florian Probst[2]

[1] Technische Universität Darmstadt
Knowledge Engineering Group
paulheim@ke.tu-darmstadt.de
[2] SAP Research
{d.oberle,roland.plendl,f.probst}@sap.com

Abstract. The goal of reference models is to establish a common vocabulary and recently also to facilitate semantically unambiguous information exchange between IT systems. However, IT systems are based on implementation models that typically deviate significantly from the reference models. This raises the need for a mapping mechanism, which is flexible enough to cope with the disparities between implementation model and reference model at runtime and on instance level, and which can be implemented without altering the established IT system. We present an architecture that solves this problem by establishing methods for representing the instances of an existing IT-System in terms of a reference model. Based on rules, the concrete nature of the representation is decided at run time. Albeit our approach is entirely domain independent, we demonstrate the feasibility of our approach in an industrial case study from the Oil and Gas domain, using the ISO 15926 ontology as a reference model and mapping it to different Java and Flex implementation models.

1 Introduction

Semantic modeling techniques have evolved in the areas of knowledge representation [1], object orientation [2,3], and recently ontologies [4]. However, neither software engineers nor knowledge engineers had considered enterprise applications for using semantic modeling techniques several decades ago. In the meantime, the complexity of enterprises as well as their need to exchange information ad-hoc continues to grow leading to an increased awareness of the need for semantically richer information. Since the early 1990s, the idea of modeling specific aspects of an enterprise has led to the definition of various *reference models* where semantic modeling techniques are used to formalize the concepts identified by the different standards bodies [5]. The initial purpose of such standards was to create a commonly accepted nomenclature for representing structural and operational aspects of enterprises. Once these standards had been accepted by a large enough community, industry solutions exploiting them have been implemented closing the loop to allow enterprises using standards-based IT systems. [6] This situation facilitates building new composite applications based on the reference model.

A. Sloane and U. Aßmann (Eds.): SLE 2011, LNCS 6940, pp. 160–179, 2012.

In order to illustrate this, we refer to the example of the Oil and Gas industry whose declared goal is to enable information integration to support, e.g., the integration of different data sources and sensors, the validation of information delivered by different sources, and the exchange of information within and between companies via web services [7]. This requires that Oil and Gas IT systems, e.g., asset management or facility monitoring, share information according to a reference model, and that they can interpret messages using that model. In the particular case of the Oil and Gas industry, the reference model is given by the ISO 15926 ontology which is formalized in the W3C Web Ontology Language (OWL) [8]. First middleware solutions are available to address the task of information integration, e.g., IBM's Information Integration Framework [9]. Such middleware solutions allow information exchange between the existing IT systems based on ISO 15926. Further, they facilitate the creation of new composite applications, such as production optimization or equipment fault protection. Such composite applications depend on information stemming from several existing IT systems, and, thus, benefit from semantically unambiguous information exchange.

Since the reference model is typically designed *ex post*, i.e., long after IT systems have been put in place, the IT systems' *implementation models* have to be mapped to such a reference model. Different IT systems feature implementation models specified in different implementation languages. Examples for such implementation languages are object-oriented languages or relational schemas. Therefore, the outlined setting requires to cope with *arbitrary* implementation languages. In addition, mapping between reference and implementation models is typically a non-trivial task which requires a *flexible* mechanism to cope with all kinds of disparities between the two kinds of models. The reason is that implementation models serve the purpose of providing a model which allows for simple programming resulting in efficiently executable code. In contrast, reference models serve the purpose of providing a clear, formal conceptualization of a domain. Further, the mapping has to be *bidirectional* since the IT system has to send and receive messages expressed according to the conceptualization underlying the reference model. In addition, for coping efficiently with the disparities between reference and implementation model, the mapping process itself must happen at *runtime* and on the *instance level*. For example, an asset management application might contain instance data about a specific pump. This instance data has to be represented by means of ISO 15926 for information exchange with other IT systems via a middleware solution. Finally, the established IT systems cannot be touched in most cases. That means, the mapping mechanism has to be implemented in a *non-intrusive* way.

Despite the existence of sophisticated solutions for model mapping [10], database integration [11], or ontology mapping [12], the existing approaches for mapping models of both kinds are still limited with respect to supporting the outlined settings. Therefore, we contribute a flexible, bidirectional, and non-intrusive approach for performing the mapping process between reference and implementation models at run-time and on the instance-level. The approach can be used

with a multitude of arbitrary implementation languages and especially when the reference model is given ex post. Although we explain our approach along an example of the Oil and Gas domain, our approach is generic, meaning that it prescribes an architecture that can be instantiated differently depending on the language of the reference model, implementation model, IT system landscape or the application domain.

We start by introducing typical deviations between reference and implementation models in Section 2, using the ISO 15926 ontology as a running example. Section 3 surveys related approaches along our distinguishing features of being flexible, bidirectional, non-intrusive, runtime, instance level, and ex post. Section 4 introduces our reference architecture, which is instantiated in a case study from the oil and gas domain in Section 5. A scalability and performance evaluation can be found in Section 6. Finally, we give a conclusion in Section 7.

2 Typical Deviations

Reference models and implementation models are different by nature. The reason is that implementation models are task-specific, with the focus on an efficient implementation of an application. In contrast to reference models, modeling decisions are geared towards a pragmatic and efficient model. Due to those differences, one often faces the situation where implementation models and reference models are incompatible in the sense that a 1:1 mapping between them does not exist. This also holds when both kinds of models are specified in the same language.

To show some typical deviations between the two kinds of models, we use examples from the Oil and Gas domain. As discussed in the introduction, the ISO 15926 ontology serves as a reference model. Facilitating information exchange between IT systems using ISO 15926 requires serializing data (e.g., Java objects) from IT systems to RDF data and deserializing that data back to data in the receiving IT system. The W3C Resource Description Framework (RDF) [13] is a semi-structured, graph-based language that applies triples to represent statements about Web URIs. For example, the triple

```
sys:valve_0243 rdf:type iso15926:Valve.
```

states that `sys:valve_0243` is an instance of the ontology category `iso15926:Valve`. The subject (`sys:valve_0243`), predicate (`rdf:type`), and object (`iso-15926:Valve`) of the statement are all specified as URIs, using abbreviated namespaces [14].

Fig. 1 shows a typical mismatch between reference and implementation models, viz., a *multi-purpose class*. A class `EquipmentImpl` could be used to model different types of equipment, distinguished by the `toE` (type of equipment) flag. In the ISO 15926 ontology, several thousand types of equipment are defined as separate ontology categories. Representing each of the types as a separate class would lead to an ineffective class model, so using a single class with a flag is a more pragmatic solution.

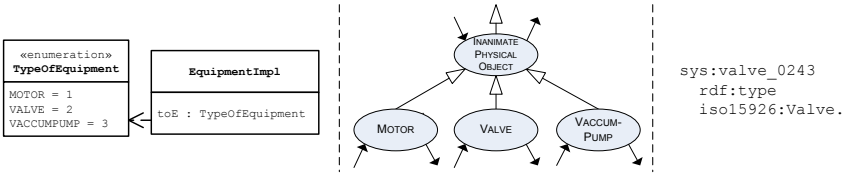

Fig. 1. Typical mismatch 1: Multi-purpose classes. The left hand side shows a class model, the middle depicts an excerpt of the ISO 15926 ontology, and the right hand side shows a sample desired RDF serialization

With a 1:1 mapping, however, an `EquipmentImpl` object cannot be serialized without information loss. A 1:1 mapping can only map `EquipmentImpl` to the ontology category INANIMATE PHYSICAL OBJECT, which serves as a common super category for all equipment categories. With that mapping, an `EquipmentImpl` object `valve_0243` would be serialized as

```
sys:_0243 rdf:type iso15926:InanimatePhysicalObject.
```

This serialization implies the loss of information stored in the `toE` attribute, as it does not support a deserialization to an `EquipmentImpl` with a proper value for the `toE` attribute. Therefore, an approach relying on a 1:1 mapping is not very useful here.

Other similar deviations encompass *conditional classes*, e.g., classes which depict objects that may or may not exist (usually determined via a `deleted` flag), and *artificial classes*, which depict objects of different kinds without a meaningful common super category (such as a class `AdditionalCustomerData` storing both a social security number and an email address). Furthermore, there are also *multi-purpose relations*, which may depict different relations in the ontology, depending on a flag or on the nature of the related object.

Another typical deviation between implementation and reference models are *shortcuts*. Shortcuts may span across different relations between objects, leaving out intermediate entities. Fig. 2 shows such a deviation: the ISO 15926 ontology defines a category APPROVAL, which has a relation both to the approved THING, as well as the approving authority. The APPROVAL itself has more detailed attributes, such as a DATE.

A corresponding implementation model defines an `Order` and a `Person` class (both of which are categories also present in the ISO 15926 ontology), but omits the intermediate APPROVAL category in favor of a direct relation, implemented as an attribute in the `Order` class. To properly serialize an `Order` object, an APPROVAL instance has to be created as well in the serialization, as depicted in Fig. 2.[1] During deserialization, this instance is then used to create the link

[1] The underscore namespace prefix is a standard RDF N3 notation which denotes an *anonymous resource*, i.e., an object that is known to exist, but whose identity is unknown [14].

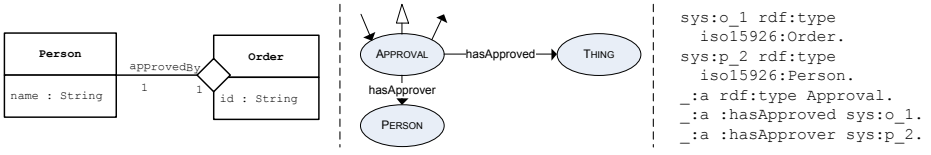

Fig. 2. Typical mismatch 2: Shortcuts

between the `Order` and the `Person` object. Such a serialization encompassing multiple categories in the ontology cannot be implemented using a 1:1 mapping.

Deviations may also occur on the attribute level. A typical example are *compound data types* such as dates, which are most often represented as one variable in a class model. In the ISO 15926 ontology, dates are represented using single individuals for the day, month, and year part of a date, respectively. Another deviation are *counting attributes*, such as an integer attribute `numberOfParts`, which has to be serialized as a set of (anonymous) individuals, and deserialized back to an integer number.

3 Related Work

The deviations introduced in the previous section require a careful mapping between implementation model and reference model if the goal is to exchange information based on the reference model. A semantically correct mapping is the prerequisite for unambiguous, lossless information exchange, and, thus, for building new composite applications relying on information from several IT systems. Related approaches have been proposed in many fields, such as information integration or ontology mapping, as depicted in Table 1. In this section, we provide a survey of such related approaches and conclude that none of them supports all required features.

As explained in the introduction, reference models are typically designed after IT systems' implementation models are established. Therefore, an adequate approach must support an *ex post* mapping. Moreover, simple 1:1 mappings between both types of models are not sufficient. Instead, *flexible* mappings are required to cope with the typical deviations addressed in the previous section. Since IT systems or new composite apps have to send and receive messages in terms of the reference model, the mapping must be *bidirectional*. Further, the mapping must happen at *runtime* of the IT system and on the *instance level*, i.e., concrete instance data has to be mapped that adheres to the implementation and reference model, respectively. Most IT systems are established and cannot be altered. Therefore, the mapping mechanism must be *non-intrusive*. If a multitude of IT systems is involved, it is likely that they use different implementation languages to specify their implementation models. Correspondingly, *arbitrary* implementation languages have to be supported.

The first category of related approaches shown in Table 1 is the field of *database design* which distinguishes between conceptual models and logical

Table 1. Categorization of approaches according to different criteria. A "yes" means that there are approaches in the category that fulfill the criterion, not that *each* approach in the category fulfills the criterion.

Approach \ Criterion	ex post	flexible	bidirectional	runtime	instance level	non-intrusive	arbitrary
Database Design	no	yes	no	no	no	no	no
Model-Driven Engineering	no	yes	yes	no	no	no	yes
Information Integration	yes	yes	yes	yes	yes	yes	no
Direct Semantic Programming Models	yes	no	yes	yes	yes	yes	no
Indirect Semantic Programming Models	no	yes	yes	yes	yes	no	no
API Generation from Ontologies	no	yes	yes	no	no	no	yes
Ontology Mapping	yes	yes	yes	no	yes	yes	no

models according to [15]. Both bear resemblance to our notion of reference and implementation models. CASE tools support the database designer to create a conceptual model, e.g., an ERM [16], and automatically transform it to a logical model, e.g., a relational schema. More recent tools, such as *Together* by Borland,[2] also support reverse engineering models from existing databases. However, ex post mappings to other schemata are typically not supported. Also, the approaches do not work on the instance level and usually not at runtime.

In the area of *model-driven engineering*, platform independent models (PIMs) are transformed to platform specific models (PSMs), which generally correspond to our notion of reference and implementation models. However, this transformation does not happen at runtime and is also not intended to work on the instance level. Originally conceived as unidirectional (transformation from PIM to PSM), recent approaches also allow bidirectional mappings by implementing reverse engineering [17,18].

Information integration deals with accessing information contained in different IT systems using one central, mediated schema. The integrated systems are addressed using *wrappers*, which provide an interface to the information contained in the system, typically to their database. Queries posed using the mediated schema are translated to sub queries posed to the individual wrappers, and the results are collected and unified by an information integration engine [19]. While there are a number of very powerful information integration systems, e.g., for using an ontology as a central, mediated schema [20], they are most often limited to integrating sources of one technology, i.e., SQL-based databases.

With a growing popularity of ontologies, different *semantic programming models* have been proposed for building ontology-based software. Those programming models can also be used for mapping instances in a program to a reference

[2] `http://www.borland.com/us/products/together/`

ontology. There are two types of semantic programming models: *direct* and *indirect* models [21].

Direct semantic programming models let the user work with any object-oriented programming language (e.g., Java) and allow for mapping classes of that language to categories in the ontology. With such a mechanism, it is possible to serialize programming language objects as RDF data according to a reference ontology, and vice versa. While most direct programming models are intrusive (see [22] for a survey), *ELMO*[3] and the work discussed in [23] provide non-intrusive implementations as well and foresee ways of dynamically creating mappings at runtime. Since the possibilities for expressing mappings are limited in all those approaches, it is required that the "the domain model should be rather close to the ontology" [23].

Indirect semantic programming models provide a set of meta-level programming language constructs (such as an `OWLClass` and an `OWLObjectRelation` class in Java), instead of providing mechanisms for mappings on the model level. The most well-known examples are *JENA* [24] and *OWL API* [25]. Since the developer works directly with the ontology-based constructs, the approach allows for flexibly using arbitrary ontologies. The drawback is that an ex post approach is not possible, since the concepts defined in the ontology are used directly in the code, typically as hard coded strings. Furthermore, since the indirect programming model is used directly and deeply in the software, the approaches cannot be regarded as non-intrusive.

Approaches for *generating APIs from ontologies* are a special type of model-driven engineering approaches (see above) which take ontologies, e.g., OWL files, as input for generating class models. Thus, they share the same set of characteristics as MDE approaches. Typical examples for such approaches are *RDFReactor* [26] and *OWL2Java* [27] (see [22] for a survey).

Ontology mapping approaches deal with creating mappings between different ontologies. If instance data are described using an ontology A, they can be interpreted using the terms of an ontology B, if a mapping from A to B exists. The first approaches to ontology mapping relied on tables storing pairwise correspondences between elements in each ontology, and were thus of limited expressivity. Recently, approaches such as *SPARQL++* [28] also allow for flexible mappings, which may also be stored in a non-intrusive way, i.e., external to the mapped ontologies, e.g., by using *C-OWL* [29], or build bridges between RDF and XML following different schemata, such as *XSPARQL* [30]. *Ontology matching* [31] aims at automatically discovering such mappings. However, a runtime mapping to the implementation model level is not foreseen and has to be provided by additional mechanisms, e.g., by employing a semantic programming model (see above).

In summary, there is a large number of approaches which can be exploited for implementing information exchange based on reference models. However, none of those approaches fulfills the complete set of requirements and provides the full flexibility which is needed in many real-world industrial projects.

[3] `http://www.openrdf.org/doc/elmo/1.5/`

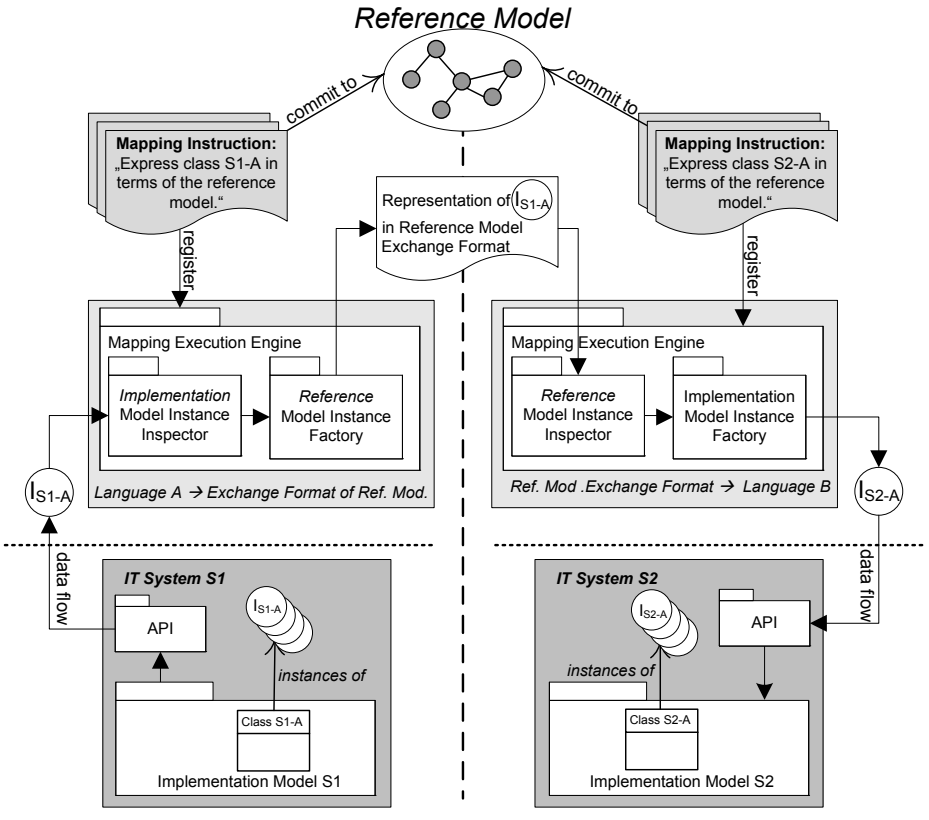

Fig. 3. General architecture for information exchange between two systems

4 Reference Architecture

Considering the shortcomings of existing mapping approaches, we propose a reference architecture for mapping between implementation and reference models. The architecture meets the criteria of flexible, bidirectional, and runtime mappings which operate on the instance level. The architecture is designed to be implemented in a non-intrusive way, so that it can be employed with legacy systems where the reference model is provided ex post. This also covers the development of new composite applications and mashups from existing applications. The instantiations can operate on arbitrary implementation languages.

Fig. 3 shows the central elements of the architecture which are needed for transferring information from one IT system (S1) into another one (S2) in a semantically consistent way and without changing the implementation models of S1 and S2.

The central means for interoperability is a *reference model*. For each class of the implementation models (or any other relevant element), mapping instructions are written that explain the classes (or other constructs) of the

implementation models in terms of the reference model. In this sense, the mapping instructions establish a partial conceptual commitment of the implementation model to the reference model. Formalizing the mapping instructions is performed at design time and by an expert knowing the implementation model as well as the reference model. The mapping instructions are processed by a *mapping execution engine*. The mapping instructions are *executable*, i.e., they can be applied at run-time for providing flexible mappings.

For each IT system, two separate mapping execution engines are established, one for mapping from the implementation model to the reference model, and one for mapping from the reference model to the implementation model. Each of the two engines consists of a *model instance inspector* for the source model, and a *model instance factory* for the target model (Fig. 3 only depicts one mapping execution engine for each IT system, which is needed to illustrate the data flow from S1 to S2).

The information exchange between S1 and S2 is implemented as follows: Instances according to the implementation model of system S1 are in use. If the information carried by one of these instances is to be transferred to system S2, this instance is sent to the mapping execution engine via the API of IT System 1 (in Fig. 3 an particular instance $I_{S1\text{-}A}$ is used to demonstrate the data flow). First, the instance inspector identifies from which class the instance was derived and selects the appropriate mapping instruction. Then, the reference model instance factory creates one or more instances of a class (or different classes) of the reference model in such a way that all information of the instance $I_{S1\text{-}A}$ is represented appropriately, including the relations between the classes. This "instance" is sent to S2.

Note that the reference model can be represented in more than one representation language. Hence, the mapping instructions can be written for more than one representation language. However, the language in which the reference model is written serves as exchange format for the information between S1 and S2. The *mapping execution engine* of S2 receives the previously generated instance and identifies via the *reference model instance inspector* to which class or classes in the reference model the arrived information belongs to. In the final step the *implementation model instance factory* creates one or several instances according to the implementation model of S2. The created instance is handed over to S2 via the API of S2.

It is noteworthy that, since arbitrary mappings and implementation models are possible, the number of objects in S1 and S2 does not have to be the same. An object from S1 (and potentially a set of other objects related to that object) are serialized in a data structure which conforms to the reference model, and deserialized into an object or a set of objects for S2. Since those object models can be conceptually different, the set of objects created for S2 can be substantially different from the original set of objects in S1, and may also be implemented with a different language. Thus, the approach is able to bridge both *conceptual* as well as *technological* heterogeneities.

5 Case Study

The previous section introduced an architecture that can be implemented differently, depending on the language of the reference model, the implementation models, and the IT system landscape. For example, we have discussed an instantiation in the area of emergency management in [22]. In the following, we introduce an instantiation in an industrial case study in the outlined Oil and Gas domain and show how the generic elements of the architecture are implemented in that concrete scenario. The role of the reference model is played by the ISO 15926 Oil and Gas ontology [8] specified in OWL/RDF. For the sake of brevity, Figure 4 shows the instantiated architecture with *facility monitoring* as an existing IT system and *production optimization* as a new composite application. Further existing IT systems are in place (not shown in Figure 4), e.g., rotating equipment monitoring, engineering systems, or asset management, which exchange information with production optimization. With respect to the implementation languages, we assume that the facility monitoring is Java-based, and the production optimization is Flex-based, with class models (implemented in Java and Flex, respectively) as implementation models. The latter provides an interface for exchanging objects with JSON [32] to facilitate data exchange.

5.1 Mapping Specification

To facilitate an executable mapping, we use *rules* for expressing mappings between class models and the ISO 15926 ontology (and vice versa). These rules can be evaluated at runtime on Java and JSON objects to create the desired RDF graph describing the object, and on an RDF graph to create the corresponding set of objects. Fig. 5 shows how the mapping rules are used to transform a Java object into an RDF graph and back, using the example depicted in Fig. 1. We have employed a set of simple rule-based languages, which re-use elements from common querying languages, such as XPath [33] and SPARQL [34]. The next sections explain the different rule syntaxes in detail.

Mapping Class Models to ISO 15926. Our mapping approach uses tests on the objects to be mapped as rule bodies, and a set of RDF triples to be produced as rule heads. For each Java and Flex class, one rule set is defined. In cases where objects have relations to other objects, the rule sets of the corresponding related classes are executed when processing a related object. Rule sets defined for super classes are inherited to sub classes, however, the developer may also override inherited rules explicitly.

For defining dynamic mappings, XPath queries are used. Utilizing such queries, RDF representations for objects can be realized dynamically. Thus, our rules have the following form: the body consists of a test to be performed on an object. The head is a set of RDF triples, each consisting of a subject, a predicate, and an object, all three of which may depend on the object to transform. For defining tests and dependent values, we use XPath expressions. If the test is evaluated positively, one or more triples are generated, consisting of a subject,

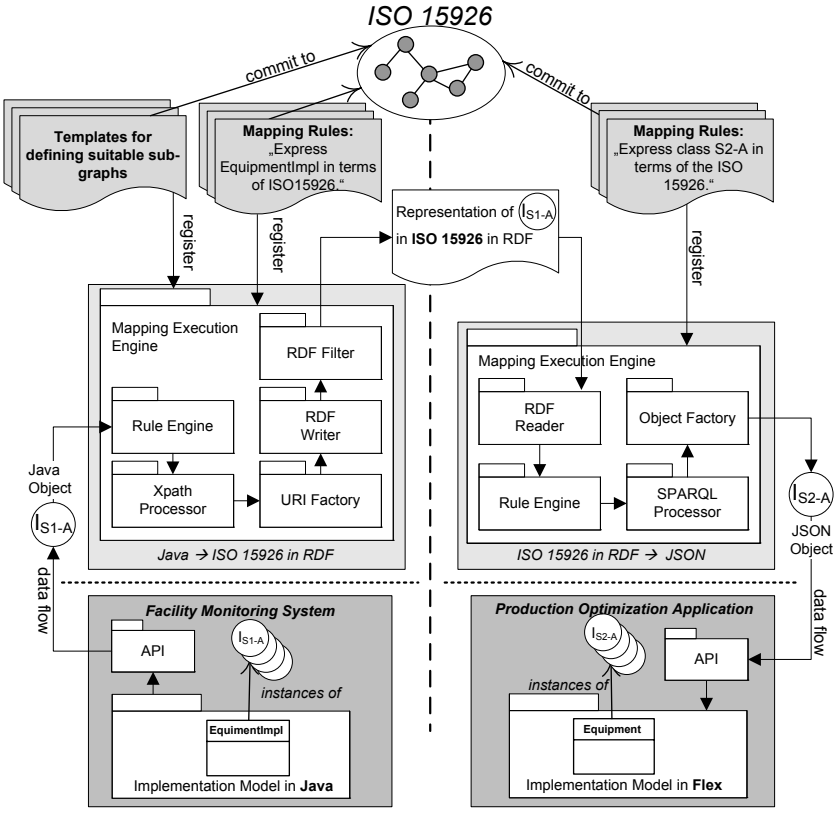

Fig. 4. Instantiated architecture in the Oil and Gas scenario

predicate, and object. The subject, predicate, and object may be either constants or XPath expressions as well. Thus, the syntax of our rules looks as follows:[4]

```
Rule ::= XPathExpr "→" Triple {"," Triple } "." ;        (1)
Triple ::= 3 * (Constant|XPathExpr) ;                    (2)
```

In this syntax, `Constant` denotes an arbitrary sequence of characters enclosed in quotation marks, and `XPathExpr` denotes an XPath expression following the XPath standard [33], enhanced by the following extensions:

- The function `regex()`, called on a Java attribute, evaluates a regular expression [36] on that object and yields `true` if the regular expression matches the attribute value, false otherwise.
- The function `repeat(XPathExpr)`, called with an XPath expression as an argument in the rule body, causes the rule head to be executed as many times as there are results for the given XPath expression.

[4] Represented using the Extended Backus Naur Form (EBNF) [35].

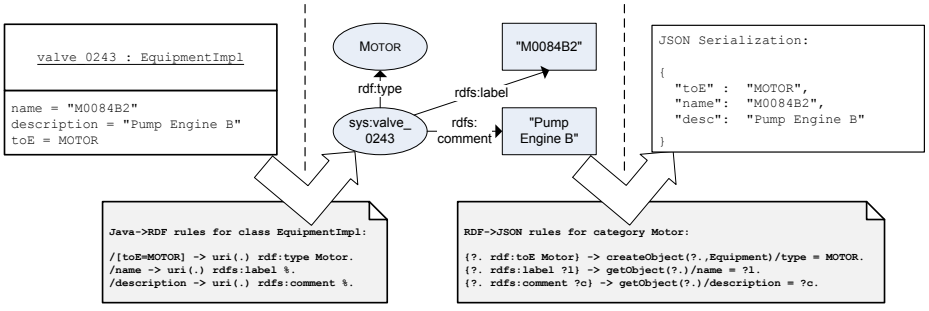

Fig. 5. Example rules for transferring Java to RDF and RDF to JSON

- The `%` symbol used in the head refers to the result of the XPath test performed in the body.
- The `.` symbol used in the head refers to the currently serialized object.
- The function `uri(XpathExpr)` assigns a unique URI to a Java or Flex object. The argument of the function is again an XPath expression, which may also use the `%` and `.` constructs, as described above.

The result of an XPath test in the body can be used as a variable in the head within the RDF triple to be generated (referred to with the `%` sign), as well as the current object (referred to with the `.` sign). The `uri` function used in a triple generates a unique URI for an object. The triples may also contain blank nodes, which are needed, e.g., to cope with shortcuts.

The XPath expressions may also contain regular expressions to deal with implicit background knowledge and non-atomic data types. Those can be used for conditions or for splitting data values. A `repeat` function can be used to cover object counting deviations (e.g., produce a set of n blank nodes for an attribute value of n). The left-hand side of Fig. 5 shows a set of rules for mapping two Java objects to a subset of the ISO 15926 ontology.

Mapping ISO 15926 to Class Models. The mapping rules from ISO 15926 to class models are similar. Again, rule sets are defined per ontology category, are inherited to sub categories, and rules can be explicitly overridden by the developer.

For rule bodies, SPARQL expressions are used. In the rule heads, objects are created by using `createObject`, and attribute values for created objects are set (by using `getObject` and an XPath expression identifying the attribute to be set). The execution order of rules is defined such that all `createObject` statements are executed first, assuring that all objects are created before attempting to set attribute values.

Typically, Java or Flex objects will be created when a condition is fulfilled, and values are set in these objects. This leads to the following rule syntax for mapping rules from ISO 15926 to class models:

```
Rule ::= SPARQLExpr "→" ObjFunction { "/" SetObjValue } "." ;
```
(3)

```
ObjFunction ::= "getObject(" SPARQLVariable{,SPARQLVariable} ","
                              ClassName ")" ;
```
(4)

```
SetObjValue ::= XPathExpr "=(ObjFunction|ValueFunction|Constant) ;
```
(5)

```
ValueFunction ::= "getValue(" (SPARQLVariable|BuiltinFunction) ","
                              ClassName ")" ;
```
(6)

```
BuiltinFunction ::=  "count("SPARQLVariable")"
                   | "concat("SPARQLVariable,{SPARQLVariable}")" ;
```
(7)

Like in the rules for creating RDF representations from objects, `XPathExpr` denotes an XPath expression. `SPARQLExpr` denotes the WHERE clause of a SPARQL expression, and `SPARQLVariable` denotes a variable defined in that WHERE part and is used for referencing the query's results. `ClassName` is the name of a Java class which is used when creating objects and object values (for handling primitive types, the corresponding wrapper classes are used as a class name). The right-hand side of Figure 5 shows how to use the rules to map RDF representations to Flex objects.

The built-in functions `count` and `concat` are used for counting results and concatenating strings, respectively. Both functions are on the feature list for the next version of SPARQL [37], thus, our proprietary support for those functions may be removed from the rule language once that new version becomes a standard with adequate tool and API support.

For determining which categories an RDF instance belongs to, and for executing SPARQL statements, reasoning on the RDF graph and the domain ontology can be used. To cover non-atomic data types, the rules may use a `concat` function for concatenating different results of a SPARQL query. For coping with object counting, a `count` function can be used to produce the corresponding attribute values (once SPARQL version 1.1, which supports counting, becomes a standard, this will be obsolete).

Although the rules for both directions look similar, there is one subtle difference. The XPath expressions used on the object model are executed with closed world semantics, while the SPARQL expressions used on the RDF model are

executed with open world semantics[5]. The rationale is that the set of objects in an information system is completely known (and therefore forms a closed world), whereas an RDF graph representing a set of objects will typically only represent a subset of the original information.

5.2 Template-Based Filtering for Data Exchange

The rules discussed above are typically evaluated in a recursive manner. This may lead to problems when creating the data structure for an implementation model instance. When creating the RDF representation for an object, each object that is encountered underway is queued and processed. For very large connected object graphs, this means that the resulting RDF graph can grow fairly large. Especially when using that graph for data exchange between applications, such a large graph can be undesirable for reasons of performance.

A straight forward way would be defining different rule sets for each class, depending on which kind of object is currently serialized. Such a solution would lead to n^2 rule sets for n classes and thus be rather costly. If we would want to take arbitrary paths into account (e.g., include the address of a person's employer into the annotation, but not the addresses of that person's friends' employers), the complexity would even be exponential.

A better alternative is to use *templates* which define the sub graph that is to be generated for an object of a certain class. While rules define *the whole* possible graph that can be produced for an object and are thus universal, templates specifically restrict that graph to a sub-graph. This alternative reduces the complexity to n rule sets and n templates. In our approach, the templates can be written in plain RDF, which allows for a straight forward definition and re-use of existing tools. Furthermore, since the rules are universal, they may be reused for different information transmission use cases by only applying a different set of templates.

5.3 Non-intrusive Implementation

For our case study, we have implemented the solution sketched above in a prototype capable of exchanging objects between Java and Flex applications. In our integrated prototype, the Flex applications run encapsulated in Java containers, and their API provides and consumes Flex objects in JSON notation [38].

Fig. 4 shows the architecture as it was implemented for the case study. The left hand side depicts an equipment fault protection application, implemented in Java. The mapping execution engine shown in the figure produces RDF graphs from Java objects which are obtained from the application through its API. The rule engine processes the mapping rules discussed above and uses an object inspector implemented with *JXPath*[6] for performing tests on the Java objects.

[5] The `count` function discussed above counts *results* in the result set of a SPARQL query (which forms a closed world), not in the underlying graph.

[6] `http://commons.apache.org/jxpath/`

The rule engines processing our rule languages have been implemented using parsers generated from abstract grammars using *JavaCC*.[7]

By using Java's reflection API [39] and relying on the Java Beans specification [40] (a naming convention for object constructors and for methods for accessing property values), the implementation is non-intrusive and does not require changes to the underlying class model in Java. A *URI factory* keeps track of the RDF nodes created for each object and assigns unique URIs. An RDF writer, implemented with *JENA* [24], creates RDF files which conform to the ISO 15926 ontology. These files are the cetral elements for semantically correct informatione exchange in the proposed architecture.

As discussed above, it is desirable to reduce the set of transmitted RDF data as far as possible. Thus, we have implemented a filter based on templates expressed in RDF. Thus, the mapping rules from the Java model to the ontology have two parts: the rules themselves, generating the whole possible graph for an object, and the template reducing that graph to the desired subset.

On the right-hand side, a production optimization application is shown, which is implemented in Flex, and which is supposed to consume data from the Java-based equipment fault protection application. To that end, it receives RDF data based on the ISO 15926 ontology. This data is processed using the mapping rules described above, which are executed in a rule engine. The RDF data is analyzed using JENA as a SPARQL engine, and corresponding JSON data is produced and enriched using a Java-based reimplementation of *JSONPath*[8] as an object factory. The JSON objects created are then handed to the Flex application's API. For Flex-based applications, the transformation between RDF and JSON is done entirely in the Java container, and the Flex application is only addressed by using its JSON-based API, the implementation is also non-intrusive with respect to Flex-based applications.

6 Scalability and Performance Evaluation

As the examples in Section 2 show, information exchange between IT systems require a flexible approach for transforming information from an IT system into a representation that follows a reference model and back. When using Java and Flex-based applications in the Oil and Gas domain, this means mediating between Java and Flex-based class models and the ISO 15926 reference ontology.

The instantiation of the architecture shown in Section 5 is capable of handling all the typical deviations introduced. To build useful solutions, especially real-time systems, this implementation has to be able to handle larger amounts of data in short times. Therefore, we have run several performance tests on our approach.

For these performance tests, we have used artificially created objects graphs consisting of up to 10,000 interconnected objects and transformed them to RDF and back to Java and Flex with our mapping engine. Fig. 6 shows the processing

[7] `https://javacc.dev.java.net/`
[8] `http://goessner.net/articles/JsonPath/`

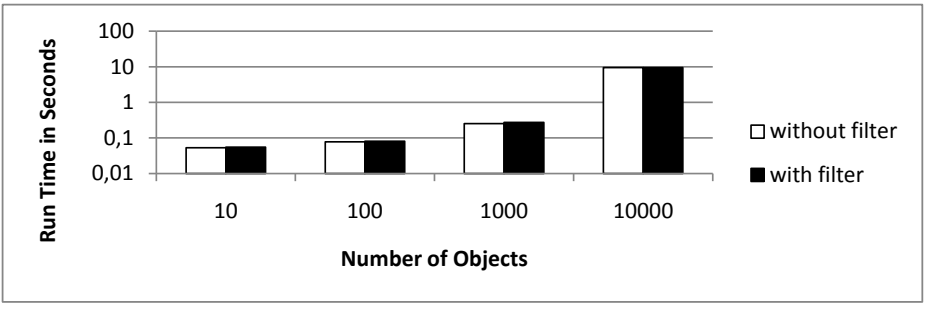

Fig. 6. Runtime behavior for serializing Java objects as RDF

time for serializing Java objects in RDF, once with and once without applying the template-based filtering mechanism. It shows that the time required per object is below one millisecond, and that the processing time scales linearly with a growing number of objects. The figures for transformation from Flex objects look very similar.

For deserializing Java objects from RDF graphs, different reasoning mechanisms can be used for evaluating the SPARQL queries. For the evaluation, we have used three different built-in reasoners in the JENA framework: a simple transitive reasoner only working on subclass and subproperty relations, an RDF(S) reasoner, and an OWL reasoner. Except for the latter, the processing time for each object is below ten milliseconds. In either case, the approach scales linearly with a growing number of objects. Again, the figures for transforming to Flex-based models look very similar.

The trade-off for using more powerful reasoning is a more complex definition of mapping rules for the developer, since certain information which may be required for the mapping (e.g., class membership of RDF instances) can be either inferred automatically by the reasoner, or encoded explicitly in a mapping rule.

Figures 6 and 7 also demonstrate the impact of the template-based filtering mechanism on performance: while applying the template during the serializing step does not lead to a significant performance impact, smaller RDF structures may be deserialized much faster than larger ones. Thus, it is beneficial to reduce the RDF structures before the transfer to another system as far as possible. For example, if the RDF structure to be transferred can be reduced by 50%, the total processing time is also decreased by 50%, as the time for creating Java objects from RDF decreases linearly, while there is no overhead in applying the filter.

In summary, the evaluations show that the dynamic, rule-based mapping algorithm can be implemented in a fast and high-performance manner, which does not add any severe run-time overhead to the information exchange between IT systems, and which also scales up to larger object graphs.

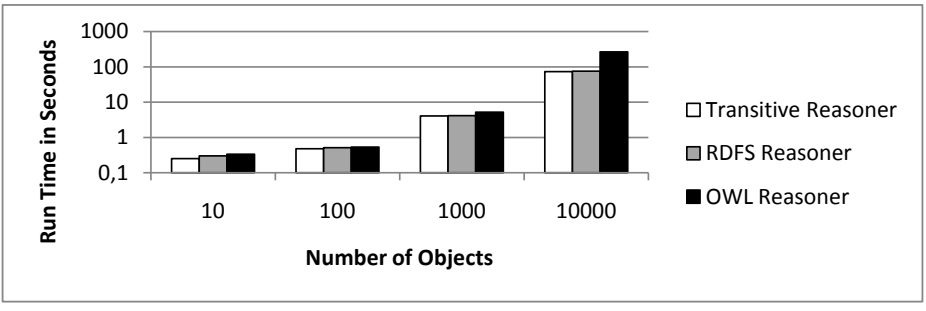

Fig. 7. Runtime behavior for deserializing RDF graphs as Java objects

7 Conclusion and Future Work

We have introduced a flexible, bidirectional, and non-intrusive approach for mapping between reference and implementation models on the instance level. The mappings are used at runtime while the mapping instructions can be specified after the implementation of the applications and the reference model. The work is motivated by conceptual deviations between implementation models and reference models that can be frequently observed in existing software systems. We discussed these domain independent deviations in detail in order to draw attention to an often ignored problem that occurs when reference models are used in software engineering. The central conclusion of this discussion is that 1:1 mappings between implementation models and reference models will not lead to the expected effect of syntactically and semantically correct information exchange between independent applications.

With a case study from the oil and gas domain, we have discussed and shown in an implementation how our approach can be used to bridge both conceptual and technological heterogeneities between applications, and to facilitate semantically correct information exchange between Java and Flex-based applications, using the ISO 15926 ontology. The central advantage of the presented approach is that not classes of the implementation models are mapped 1:1 to the reference model but only its instances. *Explaining* the instances of the implementation model in terms to one or more classes of the reference models allows for a great flexibility for the implementation model. Legacy systems can be annotated without making compromises in terms of ontological or conceptual soundness, and due to the non-intrusive approach, can also be used with modern enterprise buses without having to be modified to comply to a newly created reference model. Furthermore, newly developed implementation models can be specified having only computational efficiency and elegance in mind. Compliance to a domain vocabulary or standards can be established via the presented approach. This opens the possibility for establishing composite applications by reusing already existing systems and information sources.

The paper has focused on the use case of information integration and exchange. However, the mechanism of mapping different implementation models

to one common reference model may also be used to access the information in different applications available as a unified linked data set, allowing for reasoning and for unified visualization [41], and for other purposes that require run-time access to a system's data in a form that can be processed by a reasoner, such as self-explaining systems, self-adapting user interfaces, or semantic event processing in integrated applications [42].

Currently, the developer has to specify the mapping rules by hand. A straight forward improvement is the provision of a tool set for assisting the developer in creating the mapping rules. In the future, techniques developed, e.g., in the field of ontology matching [31] and schema matching [43] may be employed to suggest mappings and rules to the developer in an interactive manner. However, this poses several challenges, since our rules are more complex (combining the full expressive power of XPath, SPARQL, and regular expressions) than those that can be discovered with state of the art tools. A possible solution would be to suggest an approximation of a mapping rule to the user, and let her refine that approximation to a complete mapping rule.

Acknowledgements. The work presented in this paper has been partly funded by the German Federal Ministry of Education and Research under grant no. 01ISO7009 (SoKNOS), 01IA08006 (ADiWa), and 13N10711 (InfoStrom).

References

1. Brachman, R.J., Schmolze, J.G.: An Overview of the KL-ONE Knowledge Representation System. Cognitive Science 9(2), 171–216 (1985)
2. Booch, G., Rumbaugh, J.E., Jacobson, I.: The Unified Modeling Language User Guide. J. Database Manag. 10(4), 51–52 (1999)
3. Stipp, L., Booch, G.: Introduction to object-oriented design (abstract). OOPS Messenger 4(2), 222 (1993)
4. Staab, S., Studer, R. (eds.): Handbook on Ontologies. International Handbooks on Information Systems. Springer (2009)
5. Rebstock, M., Fengel, J., Paulheim, H.: Ontologies-based Business Integration. Springer (2008)
6. Pletat, U., Narayan, V.: Towards an upper ontology for representing oil & gas enterprises. In: Position paper for W3C Workshop on Semantic Web in Energy Industries; Part I: Oil and Gas (2008)
7. Verhelst, F., Myren, F., Rylandsholm, P., Svensson, I., Waaler, A., Skramstad, T., Ornæs, J., Tvedt, B., Høydal, J.: Digital Platform for the Next Generation IO: A Prerequisite for the High North. In: SPE Intelligent Energy Conference and Exhibition (2010)
8. Kluewer, J.W., Skjæveland, M.G., Valen-Sendstad, M.: ISO 15926 templates and the Semantic Web. In: Position paper for W3C Workshop on Semantic Web in Energy Industries; Part I: Oil and Gas (2008)
9. Credle, R., Akibola, V., Karna, V., Panneerselvam, D., Pillai, R., Prasad, S.: Discovering the Business Value Patterns of Chemical and Petroleum Integrated Information Framework. Red Book SG24-7735-00, IBM (August 2009)
10. Bernstein, P.A., Melnik, S.: Model Management 2.0: Manipulating Richer Mappings. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 1–12 (2007)

11. Doan, A., Halevy, A.Y.: Semantic Integration Research in the Database Community: A Brief Survey. AI Magazine 26(1), 83–94 (2005)
12. Choi, N., Song, I.-Y., Han, H.: A survey on ontology mapping. SIGMOD Record 35(3), 34–41 (2006)
13. Manola, F., Miller, E.: RDF Primer. W3C Recommendation (February 2004), http://www.w3.org/TR/rdf-primer/
14. Berners-Lee, T.: Notation3 (N3) A readable RDF syntax (1998), http://www.w3.org/DesignIssues/Notation3
15. ANSI/X3/SPARC Study Group on Data Base Management Systems: Interim Report. FDT – Bulletin of ACM SIGMOD 7(2), 1–140 (1975)
16. Chen, P.P.: The Entity-Relationship Model - Toward a Unified View of Data. ACM Trans. Database Syst. 1(1), 9–36 (1976)
17. Hettel, T., Lawley, M., Raymond, K.: Towards Model Round-Trip Engineering: An Abductive Approach. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 100–115. Springer, Heidelberg (2009)
18. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 16-18, pp. 4–16. ACM, Portland (2002)
19. Halevy, A.: Information Integration. In: Encyclopedia of Database Systems, pp. 1490–1496. Springer (2009)
20. Sahoo, S.S., Halb, W., Hellmann, S., Idehen, K., Thibodeau Jr., T., Auer, S., Sequeda, J., Ezzat, A.: A Survey of Current Approaches for Mapping of Relational Databases to RDF (2009), http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf, (accessed July 16, 2010)
21. Puleston, C., Parsia, B., Cunningham, J., Rector, A.: Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T.W., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 130–145. Springer, Heidelberg (2008)
22. Paulheim, H., Plendl, R., Probst, F., Oberle, D.: Mapping Pragmatic Class Models to Reference Ontologies. In: DESWeb 2011 - 2nd International Workshop on Data Engineering Meets the Semantic Web. In Conjunction with ICDE 2011, Hannover, Germany, April 11 (2011)
23. Hillairet, G., Bertrand, F., Lafaye, J.Y.: Bridging EMF applications and RDF data sources. In: Kendall, E.F., Pan, J.Z., Sabbouh, M., Stojanovic, L., Bontcheva, K. (eds.) Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering, SWESE (2008)
24. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: Implementing the Semantic Web Recommendations. In: Feldman, S.I., Uretsky, M., Najork, M., Wills, C.E. (eds.) Proceedings of the 13th International Conference on World Wide Web - Alternate Track Papers & Posters, pp. 74–83. ACM (2004)
25. Bechhofer, S., Volz, R., Lord, P.W.: Cooking the Semantic Web with the OWL API. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 659–675. Springer, Heidelberg (2003)
26. Völkel, M., Sure, Y.: RDFReactor - From Ontologies to Programmatic Data Access. In: Posters and Demos at International Semantic Web Conference (ISWC 2005), Galway, Ireland (2005)

27. Kalyanpur, A., Pastor, D.J., Battle, S., Padget, J.A.: Automatic Mapping of OWL Ontologies into Java. In: Maurer, F., Ruhe, G. (eds.) Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2004), Banff, Alberta, Canada, June 20-24, pp. 98–103 (2004)
28. Polleres, A., Scharffe, F., Schindlauer, R.: SPARQL++ for Mapping Between RDF Vocabularies. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 878–896. Springer, Heidelberg (2007)
29. Bouquet, P., Giunchiglia, F., van Harmelen, F., Serafini, L., Stuckenschmidt, H.: C-OWL: Contextualizing Ontologies. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 164–179. Springer, Heidelberg (2003)
30. Akhtar, W., Kopecký, J., Krennwallner, T., Polleres, A.: XSPARQL: Traveling between the XML and RDF Worlds – and Avoiding the XSLT Pilgrimage. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 432–447. Springer, Heidelberg (2008)
31. Euzenat, J., Shvaiko, P.: Ontology Matching. Springer, Heidelberg (2007)
32. json.org: Introducing JSON (2010), http://www.json.org/
33. W3C: XML Path Language (XPath) 2.0 (2007), http://www.w3.org/TR/xpath20/
34. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (January 2008), http://www.w3.org/TR/rdf-sparql-query/
35. International Organization for Standardization (ISO): ISO/IEC 14977: Information technology – Syntactic metalanguage – Extended BNF (1996), http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.html?csnumber=26153
36. Friedl, J.: Mastering Regular Expressions. O'Reilly (2006)
37. W3C: SPARQL New Features and Rationale (2009), http://www.w3.org/TR/sparql-features/
38. Paulheim, H.: Seamlessly Integrated, but Loosely Coupled - Building UIs from Heterogeneous Components. In: ASE 2010: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 123–126. ACM, New York (2010)
39. Foreman, I.R., Forman, N.: Java Reflection in Action. Action Series. Manning Publications (2004)
40. Sun Microsystems: Java Beans API Specification (1997), http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html
41. Paulheim, H., Meyer, L.: Ontology-based Information Visualization in Integrated UIs. In: Proceedings of the 2011 International Conference on Intelligent User Interfaces (IUI), pp. 451–452. ACM (2011)
42. Paulheim, H., Probst, F.: Ontology-Enhanced User Interfaces: A Survey. International Journal on Semantic Web and Information Systems 6(2), 36–59 (2010)
43. Bonifati, A., Mecca, G., Papotti, P., Velegrakis, Y.: Discovery and Correctness of Schema Mapping Transformations. In: Bellahsene, Z., Bonifati, A., Rahm, E. (eds.) Schema Matching and Mapping, pp. 111–147. Springer (2011)