Top-level Ideas about Importing, Translating and Exporting Knowledge via an Ontology of Representation Languages

Philippe Martin, Jérémy Bénard

Importing, Translating and Exporting Knowledge via an Ontology of Knowledge Representation Languages

Philippe MARTIN
EA2525 LIM, ESIROI, Uni. of La Réunion,
F-97490 Sainte Clotilde, France, +262 262 48 33 30
+ adjunct researcher of the School of I.C.T.
at Griffith University, Australia
Philippe.Martin@univ-reunion.fr

Jérémy BÉNARD
GTH, Logicells, 55 rue Labourdonnais,
97400 Saint Denis, France

+262 262 20 93 85
Jeremy.Benard@logicells.com

ABSTRACT
This article introduces KRLO, an ontology of knowledge representation languages (KRLs), the first to represent KRL abstract models in a uniform way and the first to represent KRL notations, aka concrete models. More precisely, this article illustrates the content, principles and kinds of use cases for such an ontology. One kind is to help design tools handling many KRLs, and hence parsing, semantically analyzing and exporting knowledge expressed in these KRLs. Another kind is to let the end-users of these tools design or adapt KRLs. This ontology also supports translations based on equivalence or implication relations between types as well as some structural translations. They can be exploited by inference engines handling the expressiveness of RIF-BLD, i.e., of Datalog like rules.

CCS Concepts
• Artificial Intelligence→Knowledge Representation Formalisms and Methods • Representation languages

Keywords
Knowledge Representation Languages (KRLs); Ontology of KRLs; Knowledge Integration; Language Technologies.

1. INTRODUCTION

KRLs are *languages* which permit to represent information into logic-based forms – *knowledge representations (KRs*) – within knowledge bases (KBs). KRs are exploited by inference engines or KB management systems (KBMSs). They ease precision-oriented information sharing, retrieval and problem solving. The W3C has popularized the interest of using and interconnecting "KRs on the Web", aka "Linked Data". The "Semantic Web" is the set of representations that use W3C KRLs promoted by the W3C.

Many KRLs exist. A unique one would not be adequate for every kind of knowledge modelling or exploitation, nor for every person or tool. An expressive KRL with a rich and concise textual notation is useful for modelling and sharing complex information such as i) the content of some natural language sentences or ii) an ontology that precisely define complex types of concepts or relations. To such ends, it is preferable or necessary to use KRLs with a second-order notation and syntactic sugar for meta-statements and numerical quantifiers. On the other hand, less expressive KRLs can be simpler to learn and they have better computational properties that some applications require. This is why the W3C proposes different KRLs, all of which are less expressive that First-Order Logic (FOL). However, expressiveness restrictions also lead people not to represent some knowledge or to represent it in biased ways. Many applications can benefit from expressive unbiased knowledge. E.g., i) they can use powerful inference engines, ii) they use expressive KRs only for KB checking purposes, or iii) they are information retrieval applications only using structure matching techniques instead of complete and consistent deduction techniques. Thus, many KBs use expressive KRLs and many applications require handling many KRLs. More generally, for knowledge entering, reuse and interoperability purposes, *importing, exporting or translating KRs expressed in different KRLs* is needed, especially on the Web.

KRLs may have *abstract models*, e.g., RDF+OWL2 models or Common Logics (CL), the ANSI standard for KRLs based on FOL. These are abstract *data structure* models, such as the models or meta-models of Model Driven Engineering (MDE). These are *not* theory models of model-theoretic semantics. Different abstract models may follow different logics, e.g., FOL or the $SHOIN^{(D)}$ description logic. An abstract model may be *formally presented* via different *notations*, aka *concrete models or* concrete syntaxes, e.g., CLIF, Turtle or XML-based notations. From now on, unless preceded by "concrete", "model" refers to an abstract model. Models and notations *are* themselves KRLs: a KRL *is* a model and/or a notation. In this article, an *element* is a KRL element. A *concrete element* (CE) is a notation element, e.g., an infix or prefix representation, as in "3 = 2 + 1" and "= (3 +(2 1))" or "(= 3 (+ 2 1))". An *abstract element* (AE) is an element of a model. A model is a set of AEs. An AE may be i) a *formula*, i.e., something denoting a fact, ii) an abstract *term*, i.e., something denoting a logic object, e.g., a variable or function call, or iii) a *symbol*, e.g., one for a quantifier, variable or constant.

Importing KRs is done by a syntactic parser and a semantic analyser. One input of the parser is a file, e.g., containing a text or a graphic. Another input is a concrete grammar, hence a concrete model. The parser outputs CEs, generally organized into a Concrete Syntax Tree (CST). It may also outputs *syntax related AEs*, typically organized into an Abstract Syntax Tree (AST). From these CEs or AEs, the semantic analyser creates semantically structured AEs, typically organized into an Abstract Semantic Graph (ASG). If these AEs are not the ones required by the importing tool, a translation to other AEs

SEMANTiCS '16, September 12-15, 2016, Leipzig, Germany.

DOI: http://dx.doi.org/10.1145/12345.67890

occurs. When, as with XML based notations, the input notations are homo-iconic, i.e., when the structure of the CEs mirrors the structure of the AEs, the parser may also directly be a semantic analyser.

Exporting KRs expressed in a KRL goes in the reverse direction.

Translating KRs is translating their *logic or non-logic* objects. Thus, if the source and target KRLs are different, *KRL translation* first occurs, e.g., between CL AEs and RDF+OWL AEs or between CLIF CEs and RDF+OWL/XML CEs. Then, *KR content translation* occurs if the source and target *content ontologies* are different, i.e., if different non-logic objects or different names for these objects are used for expressing the source and target KRs. E.g., this is the case when translating between physical units. *Content translation e*xploits relations between objects in the source and target *content ontologies*, e.g., between one medical ontology and another. Such relations may come from ontology mapping. KR translation, i.e., the translation of KRLs *or* KR content, may be directly between CEs. More flexibility and genericity is achieved when translation between AEs is involved, i.e., when the import, translation and export processes are separated. Thus, current research works on translation focus on translation between AEs. However, they assume that the import and export processes are done separately, via other techniques.

This article is not about *KR content* translation. It is about the *KRL related part* of importing, translating and exporting KRs. The advantages of our approach first come from its exploitation of *an ontology of KRLs* in each of the last three processes. These advantages also come from the *three originalities* of the particular exploited ontology of KRLs. We named it KRLO. It is the first to represent KRL abstract models of different families in a *uniform way*, e.g., the RDF+OWL models and the CL model. KRLO is also the first to include an ontology of KRL notations. Finally, KRLO is the first to include rules and functions specifying default methods for input, translation and export purposes. Thus, this article is also about this ontology. We have designed some tools to help exploit it. Our approach, KRLO and these tools are meant to ease the implementation of applications handling many KRLs, including KRLs specified by end-users. These tools are: a generic *KRL parser and semantic analyser*, a *KRL translation Web server* based on it, and a Web server allowing its users to complete KRLO, for example with new models and notations. This last server directly reuses our shared KB server WebKB. Its KB sharing protocols have already been published [14, 15] and hence will not be presented in this article. KRLO, these servers, and this article with additions that did not fit here, are accessible from http://www.webkb.org/KRLs/.

Section 2 gives uses cases for our approach and situates it with respect to other ones. Section 3 gives examples and principles for the specification of KRLs in or via KRLO. Section 4 introduces the default rules or functions represented in KRLO for importing, translating and exporting knowledge. Section 5 concludes.

2. COMPARISONS AND USE CASES

2.1 For Importing Knowledge

Classic *parser generators*, such as Lex&Yacc, are given a concrete grammar with *actions* associated to its rules for building AEs in main memory. Implementing and updating the code for these actions is – or is akin to – a programming task, hence long and error prone. *Programming environment* generators are designed to ease this task, including in knowledge engineering [4]. E.g., Centaur [2] proposed declarative languages for specifying concrete grammars, abstract grammars and rules bridging them. Based on them, it could generate structured editors, parsers, type checkers, interpreters, compilers and translators. However, its declarative languages were execution oriented rather than modelling oriented: they did not ease the creation and reuse of ontologies. Thus, as with procedural code, i) small changes in the specified languages often led to important changes in the specifications, and ii) the specifications were difficult to organize into an ontology and hence were not as easy to compare, automatically analyze and reuse as in an ontology based approach. The API based approach, e.g., via OKBC for knowledge exchange between KBMSs, is also a procedural approach.

Languages created via *structure description* languages such as XML, MOF – the Meta-Object Facility of the *OMG* (Object Management Group) – or those used in Model Driven Engineering (MDE) are easy to parse and check via rather generic tools. E.g., XML tools also work on RDF/XML. However, i) these "rather generic tools" do not perform logical inferences, and ii) the concrete descriptions they exploit are often not concise or high-level enough to be used directly for knowledge entering/display or by tools for knowledge handling. E.g., which KBMS or inference engine uses XML objects internally? Hence, translations from/to other models or notations are still necessary. Our approach provides an ontology-based concise alternative to the use of XML as a meta-language for creating KRLs that follow given KRL ontologies. Thus, any notation can be used for the specifications and XML is not a required pivot notation.

To sum up, a first kind of use case for an approach based on an ontology like KRLO is to *ease the task of implementing tools that parse and check KRs written in many KRLs*. Our generic *KRL parser* can be reused and adapted. Alternatively, since KRLO has the expressiveness of RIF-BLD [16], any inference engine handling this expressiveness can be reused to parse KRLO. Then, such an engine can exploit the KRL specifications in KRLO for importing knowledge written in these KRLs. RIF-BLD (Rule Interchange Format - Basic Logic Dialect) is the W3C recommendation and interchange model for definite Horn rules with equality, i.e., for Datalog like KRLs. Since there is no negation, no closed world assumption is necessary. More precisely, in FL [14], i.e., in the KRL in which KRLO was originally designed, KRLO uses a bit more expressiveness than RIF-BLD. E.g., it uses subtype partitions, i.e., complete sets of disjoint subtypes for types. However, this additional expressiveness is not exploited by the import, translation and export methods we propose, and hence can be dropped out when translating KRLO to a RIF-BLD compatible KRL.

A related second kind of use case is to *enable the end users of tools to extend or adapt KRLs* in more advanced ways than what other approaches can support. E.g., even though the Model Driven Engineering tool BAM is designed to handle many models and notations, its meta-model is *predefined* and hence, for notation extension purposes, BAM only proposes macros and informal annotations. An ontology based approach lets each user specify new abstract models or notations, e.g., by adapting existing ones via menus, as in an ontology editor tailored to a particular domain. This permits application developers as well as knowledge providers or consumers to tailor notations to their tastes and needs, or to the ones of a group of persons. Indeed, in many cases, implementing or extending a KRL parser, translator, displayer or navigator is either not an option or a cumbersome one. Yet, it is also often interesting to add syntactic sugar or new structures to a particular notation, to gain conciseness, ease readability or lift some expressiveness restrictions. Each time we designed complex ontologies, to help us visualize the KRs and their relationships, we added syntactic sugar to our in-house KRL (FL), e.g., for numeric quantifiers, interpretations of relations to sets or meta-statements, and some ubiquitous functions such as the f_in and f_dest in KRLO (cf. Table 1).

Nowadays, when people want to represent knowledge that cannot be fully expressed with the KRLs they need or wish to use, they represent the knowledge in incomplete or ad hoc and biased ways. Instead, with our approach they can extend their KRLs. Conversely, our approach also provides a way to exploit KBs even if they include syntactically or semantically incorrect KRs, as long as the kinds of errors are systematic. Indeed, with our approach, the used KRL model, notation or parser can be easily adapted to interpret some *systematic* incorrect usages in special ways. This can often be useful: in the study of [1], only 37.6% of *Datahub resources for Linked Data* proved fully machine-processable. Adaptations by an end-user is hardly possible when no declarative specification is used for the KRL models. With Centaur, this was possible but complex, if only because the specifications were not ontology based. Since any structure used in a Web resource can be described in KRLO, the approach we propose for KRLs can be extended and used for generating *Semantic Web wrappers*, i.e., tools parsing structures in certain resources to extract KRs.

A third kind of use case is the checking of input files with respect to a KRL model and notation. E.g., specifications in KRLO for *profiles of OWL2* [11] can be exploited to check if a KB follows a given profile. Profile discovering can similarly be done. In KRLO, the informal presentation of KRs – e.g., the use of a certain form of indentation – can be specified too and then checked too. However, KRLO does not *yet* provide rules or functions specifying a way to do those checks or profile matching. Nevertheless, the possibilities and flexibility offered by an ontology of KRLs exceed what is possible with tools and languages solely based on XML, XSLT and CSS.

2.2 For Translating Between KRLs

KRL translation is often specified from one KRL to another, between CEs or between AEs. This is the *direct mapping* approach. E.g., in Centaur, translations rules could be specified between two abstract grammars. Tools such as ODE [6] proposed rule based *languages* for specifying lexical, syntactic, semantic and pragmatic translations between CEs or AEs, from a KRL to another. [13] proposes rules translating CEs of the OWL2 Functional-Style Syntax (FSS) into CEs of the RIF-PS notation for the RIF-BLD model. The W3C does not propose translations between AEs since i) RIF-BLD is not represented into an ontology, and ii) CEs of FSS directly represent AEs of the OWL2 Structural Specification. In [12] the W3C proposes rules to translate CEs of FSS into CEs of Triples notation for the RDF model. Thus, more generally, the W3C adopted the *layered* approach, i.e., translations based on models of increased expressiveness with direct mappings between each level of expressiveness. An alternative way to reduce the number of direct mappings between KRLs is the *pivot* approach, i.e., translations between each KRL and an expressive interlingua. In the 1990s, this led to the creation of KIF (Knowledge Interchange Format), a FOL based KRL with a second-order notation, and then to Ontolingua, a library of ontologies written in KIF. In the 2000s, these works led to the CL model and its notations (CLIF, CGIF and XCL), COLORE (the Common Logic Repository) and IKL [9]. IKL is based on CL but, like KIF, can represent notions which are important for knowledge sharing but which are usually only found in KRLs based on Higher-order Logics (HOLs), e.g., certain kinds of meta-statements and numeric quantifiers.

None of the above cited works specifies or uses an ontology of KRLs. E.g., since KIF came to be the de-facto interlingua, Ontolingua did not include *specifications for particular KRL models or notations*. Its only KRL related ontology, the "Frame Ontology", included definitions for objects similar to those of OWL. [8] showed that the mapping, layered and pivot approaches are generalized when an ontology of KRL models is used, one where AEs – and then models – are related by certain *translation relations*. Each of these relations has an associated definition for performing the translation. Each relation also represents *translation properties*, i.e., whether or not the translation preserves the model-theoretic semantics, interpretations, and logic consequences of the translated AEs. Thus, given AEs to translate, a tool exploiting this approach may propose its users to choose different target models according to what they want the translations to preserve. As a proof-of-concept, [8] used XSLT to implement about 40 translation relations between AEs belonging to 25 description logics. The LATIN (Logic Atlas and Integrator) Project (2009-2012) [3] went further by representing such translation relations between many different logics. Via HETS (Heterogeneous Tool Set), LATIN exploits several FOL KRLs and HOL KRLs, e.g., Isabelle and HasCASL. Via DOL (Distributed Ontology, Modeling, and Specification Language) [7], the OMG proposes a standard KRL for i) specifying particular kinds of translation relations between KRL models and ii) using several KRLs in a same DOL document. DOL is also implemented via HETS and the authors of DOL see some results of LATIN as avenues for future extensions [7]. Ontohub is a DOL based repository which includes KRL models and translation relations between them. DOL and HETS do not specify notations. They rely on external parsers and exporters.

None of the above cited works specifies or exploits an ontology representing AEs of different KRL models in a uniform and organized way. KRLO does so by setting as many generalization relations and partOf relations as possible and, for these last ones, using the *operator-arguments schema* detailed in Section 3. Thus, KRLO can be seen or used as a way to align and integrate other KRL translation related ontologies. When direct relations cannot be used, functions and rules are used. Translations between structures do not require much expressiveness. So far, in KRLO, the few proposed translation rules between structures only require the expressiveness of definite Horn rules with equality. This is why KRLO has a version using OWL-RL and RIF-BLD, the W3C KRL model for rules with such expressiveness. In KRLO, since the translated structures are akin to reified statements, the original statements can even follow an HOL model, this does not change the expressiveness required to convert structures. *Currently*, translations represented *via rules* in KRLO are semantic-preserving *structure translation rules.* For other translations, KRLO users have to also exploit complementary ontologies and, possibly, more powerful inference engines. *In the future*, KRLO will be added to Ontohub and made exploitable via HETS.

None of the above cited works specifies or exploits an ontology of notations. KRLO does. This permits the import, translation and export tasks to exploit an ontology of KRLs and the same one. Thus, our approach extends the one presented in [8] and may be seen a *pivot* approach based on such an ontology of KRLs instead of a KRL. As illustrated in Section 4, the use of generalization relations between AEs, models or notations can maximize modularity and reuse.

Besides KRLO and Ontohub, there is another *ontology relating KRL models not belonging to a same family*: ODM 1.1 [10], an OMG specification. It uses UML for representing four KRL models: RDF, OWL, CL and Topic Maps. It also relates a few AEs of different models via semantic relations such as generalization or equivalence relations. Finally, it gives QVT rules for direct mappings between AEs of different models. Since direct mappings are used instead of *few primitives for defining and relating the various AEs*, the heterogeneity of the various KRL models is not eliminated. This heterogeneity makes the AEs difficult to *compare* or e*xploit in a uniform way*. Finally, QVT rules are not directly usable by inference engines and translating them into KRLs may not be easy. We are not

aware of works using ODM for KR import, translation or export purposes. Similarly, we have not found ontologies for notations, even for RDF. Hence, apart from our KRLO based translator, it seems there is no other translator based on an ontology for a KRL model and a notation. There are translators between notations for the RDF model, e.g., RDF-translator, EasyRDF and RDF2RDF. Their Web interfaces or APIs propose no way to parameter their knowledge import, translation and export processes.

To sum up, regarding KRL translation, a kind of use case for our approach is an increased simplification of the *implementation of this process*. A related kind of use case is to enable end users to *parameter*, *extend or adapt* KRL translations. Adding a new KRL specification to KRLO – e.g., via its Web servers – may be sufficient to specify the structural translation of this KRL to other represented KRLs: no transformation rule may be required. Another convenient feature is that no specific KRLs such as those of ODE is necessary: any KRL with at least RIF-BLD expressiveness can be used. Finally, as detailed by the authors of DOL, LATIN and [8], it is easier to implement a tool generating proofs for the properties of its translations if the tool exploits rules or relations in an ontology.

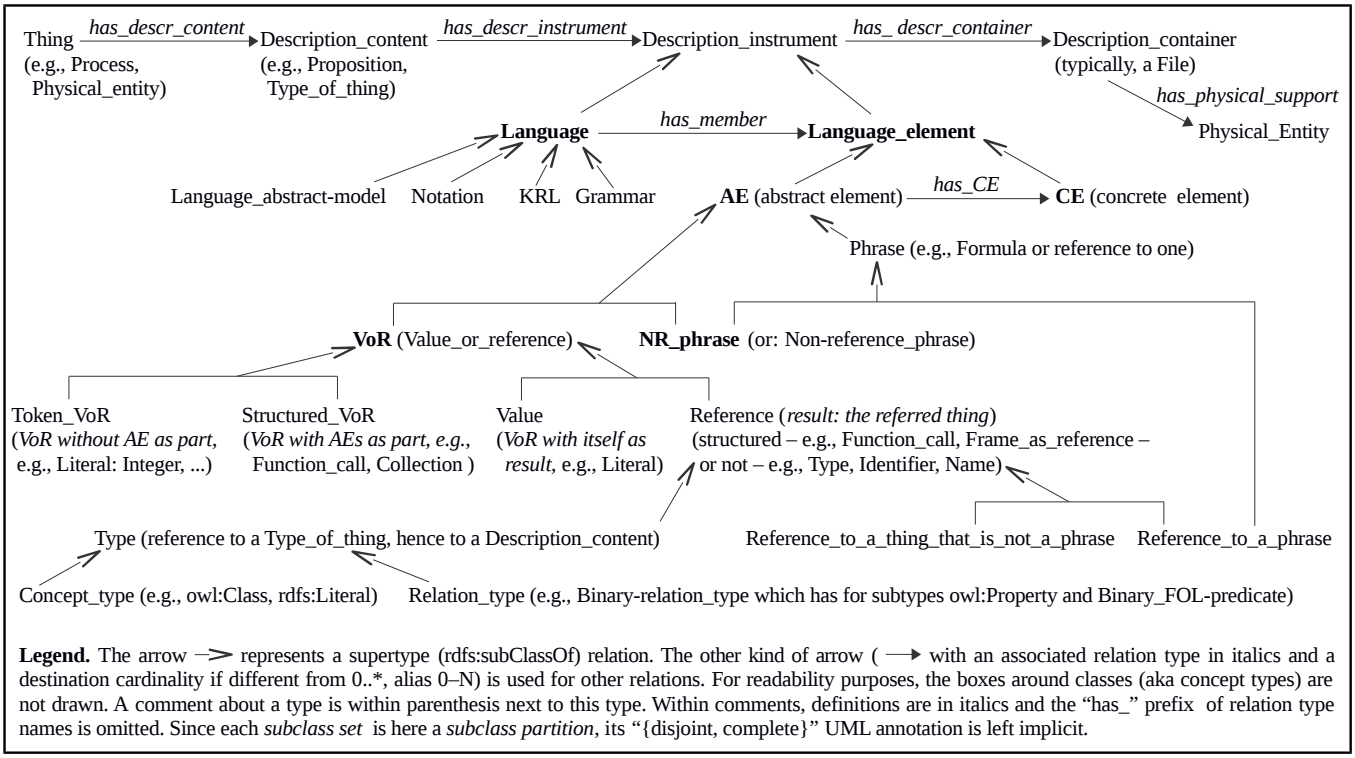
2.3 For Exporting into a KRL

Without abstract and concrete models, exporting has to be done procedurally. Centaur exploited concrete and abstract grammars for both export and import purposes. For KRL models that have a MOF or XML based notation, languages such as Mof2Text, XSLT and CSS can be used for generating KRs in other notations. However, this is not easy since these languages are not KRLs or KR query languages. This is why there have been several works on *rule based and/or style-sheet based* transformation languages *for RDF*. They specify how RDF AEs may be presented, e.g., in a certain notation, in a certain order, in bold, in a pop-up window, etc. Examples of such tools are Xenon, Fresnel, OWL-PL and SPARQL Template [5]. These tools were not initially meant to use a notation ontology: they initially required the use of a new style-sheet for each target notation. However, some of these tools – e.g., SPARQL Template – could exploit KRLO since it can be represented in RDF+OWL2-RL and SWRL, an RDF-based KRL for representing Horn rules. This is an upcoming work with the author of SPARQL template.

Here again, a kind of use case for our approach is the simplification of KR export according to each user's preferences. Exporting KRs may include the generation of hyperlinks to let users navigate from CEs in query results to other KRs of the queried KB. A related kind of use case is to enable end users to parameter, extend or adapt such exports, e.g., for knowledge pretty-printing.

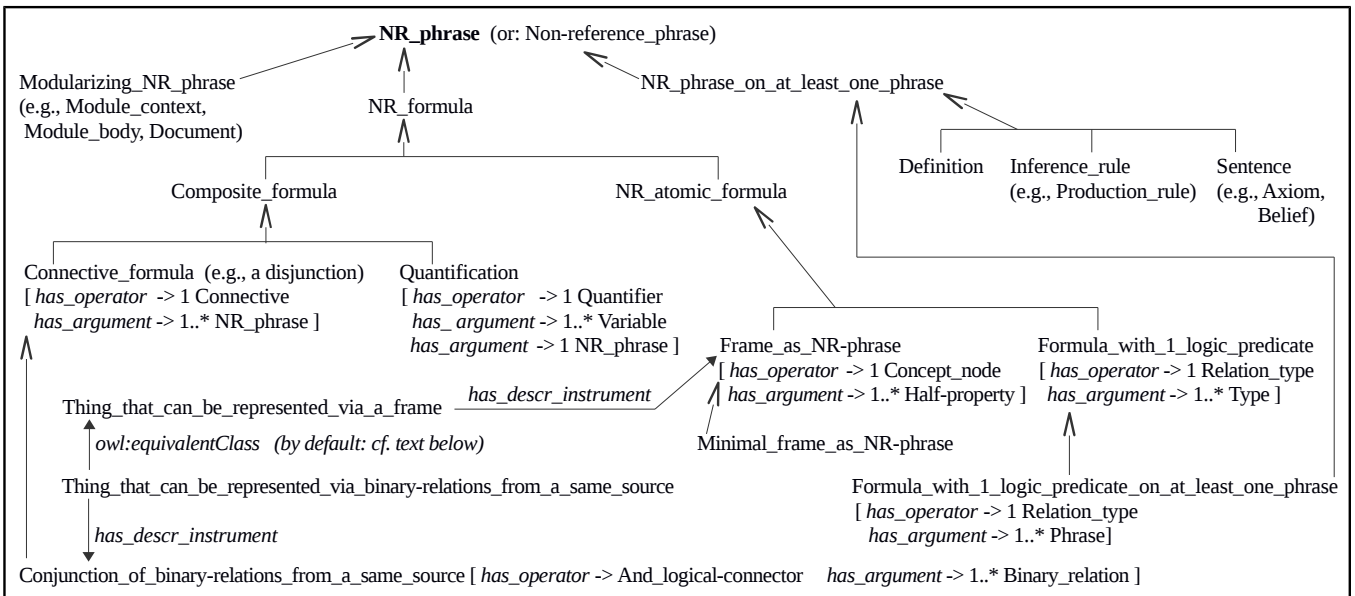
3. PRINCIPLES AND EXAMPLES FOR KRL SPECIFICATIONS

As formally represented in Figure 1, KRLO distinguishes between i) the content of a description, i.e., its meaning, what it describes, and ii) the instruments used for the description such as languages and language elements such as AEs and CEs. The notions of "types as description content", "types (as description instruments)" and "identifiers of types (as description instruments)" are distinguished. The top-level of KRLO, i.e., the part reused and specialized by specifications for particular KRLs, currently includes over 900 AE types. The structure of *each* KRL element is represented in a uniform way, like the structure of a function, i.e, as an operator with an optional set of arguments and a result. Thus, in KRLO, the six *most important primitive relation types for relating AEs* are named has_operator, has_argument, has_arguments, has_result, has_parts and has_part. The first two are subtypes of the last since the operator and arguments of an AE are also its parts. The structure of a relation is defined as having for *operator* a relation type, for *arguments* a list of AEs and for *result* a boolean. E.g., the structure



Legend. The arrow —> represents a supertype (rdfs:subClassOf) relation. The other kind of arrow (⟶ with an associated relation type in italics and a destination cardinality if different from 0..*, alias 0–N) is used for other relations. For readability purposes, the boxes around classes (aka concept types) are not drawn. A comment about a type is within parenthesis next to this type. Within comments, definitions are in italics and the "has_" prefix of relation type names is omitted. Since each *subclass set* is here a *subclass partition*, its "{disjoint, complete}" UML annotation is left implicit.

Figure 1. *Slightly adapted UML* representation of some relations between some top-level types in KRLO.



Legend. Same as in Figure 1 plus i) "1..*" means "at least 1", and ii) for readability reasons, properties of a frame are not represented via UML associations but as in RIF-PS (i.e., using"[...]" and "->") except that cardinalities are also given before the destination, as in UML.

Example of translation rule. Translating between a frame and a conjunction of relations using logic predicates requires rather complex rules, for example to take into account the quantification of variables. Hence, a simpler translation rule based on the same principle is now illustrated: the one between binary relations and non-binary one. It relates the structures of two AEs that are description instruments of something that is both a Thing_that_can_be_represented_via_a_non-binary_relation and a Thing_that_can_be_represented_via_a_binary_relation. *By default,* in KRLO, these two types are related by an equivalence relation. The next rule can be used for translations in both directions. It is in RIF-BLD/RIF-PS, i.e., it follows the RIF-BLD model and is linearized with the RIF-PS notation. ":-" means "<=". It can be read as follows: "*if a thing ?t is, because of its types, known to be representable both via a binary relation and a non_binary one, then i) these two representations have the same first argument, and ii) the second argument of the binary relation is the list of the non-first argument of the non-binary relation*". This rule only works when the relation types for the two kinds of relations are already known in the ontology. Indeed, no generation for relation type names is *here* specified. This rule also relies on the fact that, *by default*, it is stated in KRLO that any proposition can be represented via both a binary relation and a non_binary one. *The user can modify this to control the translations.*

```
Forall ?t ?binRel  ?binRelArg1  ?binRelArg2     ?nonBinRel  ?nbrArg1  ?nbrArgsExcept1st   (
  And ( ?binRelArg1 = ?nbrArg1      ?binRelArg2 = ?nbrArgsExcept1st )
  :-  And(  ?t [ rdf:type -> Thing_that_can_be_represented_via_a_non-binary_relation      rdf:type -> Thing_that_can_be_represented_via_a_binary_relation
                 r_descr_instrument -> ?nonBinRel      r_descr_instrument ->?binRel ]
            ?binRel [ rdf:type -> Binary_relation    r_1st_arg -> ?binRelArg1     r_2nd_arg -> ?binRelArg2 ]
            ?nonBinRel [ rdf:type -> Non-binary_relation     r_args -> List(?nbrArg1 | ?nbrArgsExcept1st) ] ) )
```

Figure 2. *Slightly adapted UML* representation of some important types of phrases and example of structure translation rule in KRLO.

for the relation "has_part (USA Iowa)" has for operator has_part, for arguments USA and Iowa, and for result True. The structure of a quantification is defined as having for *operator* a quantifier, some *arguments* and for *result* a boolean. A function call is defined as having for *operator* a function type, some *arguments* and a *result*. A variable or an identifier is (partially) *defined* as having for *operator* a name, for *arguments* an empty list of AEs and for *result* an AE of a certain type. Thus, in KRLO, an AE operator may be a function/relation/collection type, a quantifier or a value. Being an operator is only a structural role that different kinds of elements may have. When the AE is a formula, representing its structure is not representing its meaning directly. E.g., a relation does not actually have a boolean as result. In RDF, this is illustrated by the difference between a statement and its reification.

Since RIF distinguishes between two relation types – logic predicate and properties – so does KRL0, as shown at the end of Figure 1. In RIF, all RDF triples are represented as ternary relations of a same type named "T". In RDF terminology this time, the three arguments are the predicate (an instance of rdf:Property), the subject and the object. Thus, in such a triple, a variable can be used for the predicate without this implying the use of an higher-order logic.

Figure 2 shows some important subtypes of NR_phrase, i.e., types for statements, not references to statements. In RIF and KRLO, a *frame* is a conjunction of triples sharing a same source. Since many KRLs allow either *frames* or *conjunction of relations using logic predicates* but not both, KRLO proposes a way to translate between them, via rules based on relations of type *owl:equivalentClass* and *has_descr_instrument*, as shown in Figure 2. By modifying the list of subtypes of the sources of these relations, a user can choose the kinds of KRs to which this translation is applied. In KRLO, most translation rules use the same principle. The end of Figure 2 gives an example. The authors of RIF do not propose a way to support the above cited translation but note that it is always correct when the translated KR does not use more expressiveness than RIF-BLD [16].

Table 1 illustrates the way that particular models and notations for them are specified in KRLO. The notation related part is discussed in Section 4.2. Table 1 shows how the *AE types for a model* specialize

Table 1. The second row partially specifies RIF-M/RIF-PS, a KRL having for model RIF-M, an *invented* simple submodel of RIF-BLD, and for notation the RIF Presentation Syntax for this submodel. The first row gives a grammar for this KRL and examples of KRs in this KRL.

W3C EBNF grammar for RIF-M/RIF-PS:

Document ::= Formula * //"X*": "0 or more X"; "X+": "1 or more X"
Formula ::= Rule | AND-Formula
Rule ::= AND-Formula ':-' AND-Formula //implicitly universally quantified
AND-Formula := AtomicF | 'And' '(' AtomicF + ')'
AtomicF ::= Relation | Frame Relation ::= Const '(' Term* ')'
Frame ::= Term '[' (Term '->' Term)* ']' Term ::= Var | Const
Var ::= '?' Const //for Const, see the RIF-PS grammar

Four examples of RIF-M/RIF-PS representations (one alternative per line) for the sentences "Paris is a city. Every city is a place":

1) City (Paris) Place(?x) :- City(?x)
2) Paris [rdf:type -> City] ?x [rdf:type -> Place] :- ?x [rdf:type-> City]
3) Paris [rdf:type -> City] Place [rdf:subClassOf -> City]
4) rdf:type (Paris City) rdf:subClassOf (City Place)

Figure 2 has illustrated the way structure translation rules can be written to deal with such alternatives.

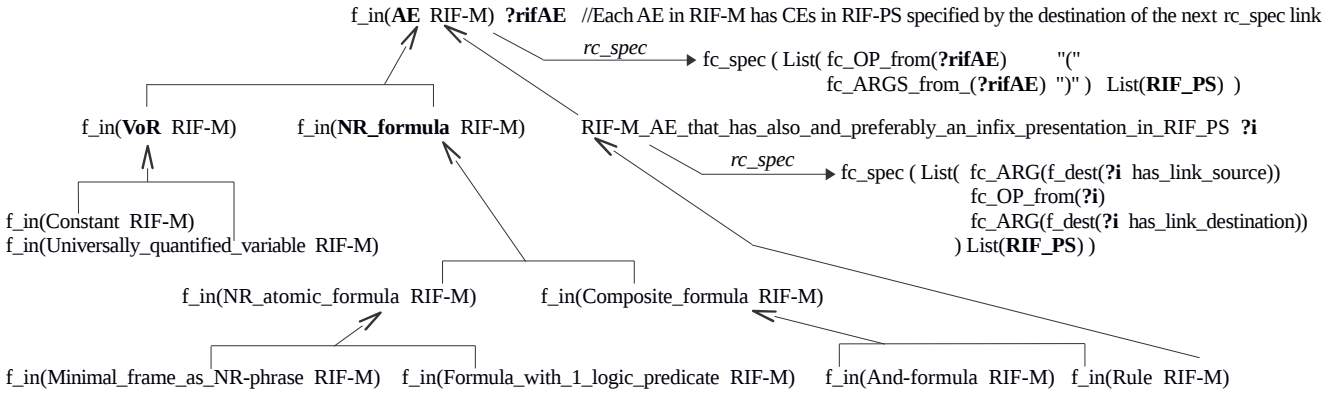
Below, the specifications for AEs use the function f_in. The specifications for CEs use the functions f_dest, fc_OP_from, fc_ARGS_from and fc_ARG.

f_dest is used for referring to the destination of a property of given type, from a given source. Here are definitions of f_dest in RIF-BLD/RIF-PS:

Forall ?pSource ?pType ?pDest (**f_dest(?pSource ?pType) = ?pDest** :- ?pSource [?pType -> ?pDest])
Forall ?pSource ?pType ?pDest (?pSource [?pType -> ?pDest] :- **f_dest(?pSource ?pType) = ?pDest**)

The other functions also have definitions in RIF-BLD/RIF-PS. The functions beginning by "fc_" are explained after the graphic.

The function **f_in(?elemType ?modelType)** returns a type ?elemTypeInLanguage that is "the specialization in the model ?modelType" of the type ?elemType. This means that any instance of ?elemTypeInLanguage is instance of ?elemType, belongs to ?modelType and any of its parts that is of type ?elemType belongs to ?modelType. This function cannot be *directly* fully expressed in RIF-BLD but, when KRLO is translated into RIF-BLD and whenever a call to this function appears, it can be replaced by a generated type defined with a RIF-BLD definition. The use of such a function is one of the ideas that led to simpler and better specifications in KRLO compared to its early versions. Given the meaning of f_in, the type f_in(**AE** RIF-M) below is subtype of AE. It can also be set as a subtype of f_in(AE RIF-BLD) if RIF-M is defined as a part of RIF-BLD.



Note. The CE specifications for frames and variables are not shown but are similar to the two shown above. For more details, see in KRLO the specifications for the models of RIF and for their presentation in RIF-PS. The difference with the RIF-PS ones is that universal quantification is not implicit in RIF-PS.

Legend. Same as in Figure 2 except that i) next to some defined types is a(n implicitly) universally quantified variable of that type (see the comment beginning by "//"), ii) the word "link" is used as a synonym for "binary relation", and iii) the type of the destination of each *rc_spec* link is specified by a call to the fc_spec function and this call is represented in RIF-BLD/RIF-PS, i.e., RIF-BLD linearized with RIF-PS.

Explanations for the *rc_spec* relations and the functions beginning by "fc_". They can for example be read when referred to at the end of Section 4.1.

Each *rc_spec* link relates an AE to a structural specification of CEs for this AE in a given notation. Above, this notation is always RIF-PS.

The first above *rc_spec* relation defines the prefixed functional presentation that *all* RIF-M AEs may have in RIF-PS, one that *most* RIF AEs must have.

The second *rc_spec* relation defines the infix presentation that *some kinds of AEs (rules, equalities, ...)* should *preferably* have an in RIF-PS. This "preferably" is represented via the following convention in KRLO: inherited specifications are *overridden* by more specialized specifications.

The second parameter of fc_spec is a list of notation types since i) an AE may have identical types of CEs in different notations, and ii) fc_spec returns a type for CEs that are member of notations of such types.

The first parameter of fc_spec is an ordered list of CE specifications:syntactic sugar (delimiters or separators) or results of functions such as fc_OP_from and fc_ARGS_from. These last functions are similar to fc_spec but they i) respectively work on the operator and arguments of their AE parameter, and ii) permit to specify the role of each CE in the specified list: operator, argument or separator. The fc_ARGS and fc_ARGS_from functions have an optional second parameter to enable the specification of a non-space separator to use between the *argument CEs* generated by these functions.

To conclude, the first rc_spec relation specifies that any instance of AE in RIF-M has (at least, by default) RIF-PS CEs composed of the following sequence of elements, separated by at least one spacing character: i) the RIF-PS representation of the operator of the AE, ii) an opening parenthesis, iii) the representation of the argument CEs separated by at least one spacing character, and iv) a closing parenthesis. This specifies a notation form called "prefix functional form" in Section 4.1. Simply modifying the order of the elements in this list permits to generate the other notation forms.

AE types from a more general model or the top-level types of KRLO. It also shows how, when needed, the structures of these AE types can be *restricted*, i.e., re-defined with more specialized parts or results. In Table 1 this is done via subtype partitions and the use of a function named f_in explained in this table. This model restriction approach is modular and concise. It exploits subtype and has_part relations between AE types and between model types as well as has_part relations from models to AE types. It worked for very different kinds of models, e.g., CL, the RIF models, RDF, JSON-LD, OWL and its profiles. With our FL notation, only 65 lines were necessary to define CL, 300 lines for the RIF models (without RIF-PRD) and about 500 lines for RDF, OWL and its profiles.

4. METHODS SPECIFIED IN KRLO

4.1 For Importing Knowledge

A CE is the presentation of an AE in a given notation. Since the structure of each AE is known, the presentation of CEs for a given AE can be specified by composition of the presentation of CEs for parts of this given AE. For textual notations, this composition is often a simple ordering – e.g., in a prefix, infix or postfix way – plus some syntactic sugar to delimit some parts. Thus, like the structure of an AE, the structure of a CE can be represented like the structure of a function. This led us to note that, like models, notations can be represented in a uniform way using few primitives. This also led us to note that a *generic* analyser could be built for all KRLs that can have a deterministic context-free grammar, e.g., an LL(1) or LALR(1) grammar. Furthermore, this analyser would be efficient since these grammars that can be efficiently parsed. As an example for the underlying idea, consider an AE composed of an operator “o” with two arguments “x” and “y”. If single parenthesis are mandatory delimiters and if single space characters are the only usable separators, this AE has only the next five possible CEs *in all the notations we know*: “o (x y)”, the prefix functional form, as in RIF-PS; “(o x y)”, the prefix list-like form, as in KIF; “(x o y)”, the infix form, as in Turtle and some RIF-PS formulas; “(x y o)”, the postfix list-like form; “(x y) o”, the postfix functional form. Five rules of an LL(1) or LALR(1) grammar can be used for specifying these five forms and they can be generalized for any number of arguments, not just two. Furthermore, if – as with the Lex&Yacc parser generators – the grammar can be divided into a lexical grammar and a non-lexical grammar, the separators can be made generic via terminal symbols with names such as Placeholder_for_begin-mark_of_arguments_of_a_prefix-function-like_element. Based on these ideas and using Flex&Bison, variants of Lex&Yacc, we created a *generic analyser for any KRL that can have an LALR(1) grammar*. This grammar for this KRL does not have to be found since it is generalized by the generic grammar that our analyser uses. Given a CE and its KRL, based on the specifications for the notation and model of this KRL in KRLO, this analyser directly generates an *abstract semantic graph*. Its genericity illustrates the usefulness of our fully ontology based approach. However, this analyser is not in-line with our approach since it is not declaratively specified. Only programmers can reuse it, via its API, or adapt it by modifying its source code. This is why, we have begun specifying two complementary *knowledge import methods* in KRLO.

The first method generates rules that directly parse a given KRL. The manual creation of such rules is not uncommon in Prolog. E.g., [17] discusses the rules, uses and good performances of an “RDF compiler on top of the sgml2pl package” in SWI-Prolog.

For the second method, we have begun i) representing models and notations for grammars and their associated actions, and ii) extending our knowledge translation and export methods for generating procedural code for these actions. These generations will allow KRLO based systems to reuse any parser generator.

Both methods exploit specifications. Both are being implemented via RIF-BLD rules *and* via functions. These two implementations maximize the possibilities of reuse by different inference engines. Furthermore, we are making a Javascript module for these functions to be executable by a browser. Thus, they will be usable in client-side scripts and adaptable by Javascript aware end-users.

The top-level of the *ontology of notations of KRLO* does not categorize all possible prefix/infix/postfix notation forms for an AE. Instead, it contains primitive relations permitting to describe them and, for usability purposes, functions to combine them. Their names begin by "fc_". Table 1 illustrates and explains them. They permit to specify i) the structure of CEs for a given type of AE in a given type of notation, and ii) the order and roles of these CEs: operator, argument or separator. Thanks to these roles, relevant grammar rules can be selected to be executed, e.g., rules belonging to the grammar of our generic analyser. For exporting into textual CEs, only the order of the terms in this list is important. The spacing between CEs, e.g., for indentation purposes, is similarly defined.

In KRLO, each AE has *one* and *only one* rc_spec relation for a given type of notation, hence one CE specification. This relation may be inherited or overriding an inherited one. The *“only one”* part is ensured by our knowledge server which prevents the entering of ambiguities when it detects them. The other *“one”* part comes from the default presentation specifications of KRLO, e.g., for the spacing between CEs. Thus, for a given AE and notation, all possible CEs are unambiguously described. This is why the import and export methods of KRLO can be represented not only as rules but also as functions, or as functional relations for the KRLs that do not handle functions.

4.2 For Exporting into a KRL

Given an AE and specifications related to a target KRL, the default export method specified in KRLO generates a CE for this AE by i) recursively navigating the parts of that AE, ii) translating each of these parts *when* the target model requires it and *when*, to do so, the method can find relations, rules or functions that satisfy the *translation properties* selected by the user (as noted in Section 2.2), iii) for each translated or original part, applying *the* CE specification associated to this part in the target KRL specification, and iv) concatenating the resulting CEs. The translation and export processes are *complete with respect to what is expressed by these relations, rules or functions* since there is one rc_spec relation for each AE in a given type of notation. Each user can control the translation and export processes. Indeed, she can *select* not only the translation properties but also the target notation and the target model or the target AEs. She can also *extend* KRLO or *adapt* her copy of KRLO, e.g., to adapt some translation rules as explained at the end of Figure 2. In the general case, knowledge export and translation are arbitrary in the sense that knowledge can be translated and exported in various ways. However, this is not the case with KRLO in the sense that *particular ways* can be represented and then selected by the user. The default presentation choices represented in KRLO permit the user not to do this work if she does not want to. However, KRLO does not represent *all* information necessary for *any* export or translation to be semantically complete with respect to *any* application. E.g., KRLO does not yet represent any inference strategy and hence the order of statements generated by translation and export processes may not be adequate for a particular inference strategy. As an example, rules may be generated in an order that lead to infinite loops if they are used with a Prolog inference engine.

Ad-hoc forms can be used when the target KRL is not formally expressive enough to represent an AE , e.g., when a statement such as “in 2015, at least 78% of birds in UK were able to fly, according to ...” has to be translated into RDF+OWL. If the specification of a target KRL describes such forms, our default export method uses them. Otherwise, the source forms are kept but within comments or annotations to isolate them from formal translations.

4.3 For Translating Between KRLs

For KRL translation, in addition to equivalence or implication relations between types of AE, KRLO currently proposes some equivalences or implications via functions and rules. These rules

and functions are for *translations between structures*, e.g., between i) non-binary relations and binary ones, ii) different structures for meta-statements (formulas about formulas), and iii) some kinds of definitions and some uses of universal quantification with implication or equivalence relations. These *structural translations* are simple: they can be expressed via RIF-BLD rules and do not require the complex strategies of general *term/graph rewriting techniques*. Backward chaining is sufficient to exploit them. Thus, KRLO does not specify a default *translation method* for combining these translation rules or functions.

For translations not yet supported by KRLO, the user has to import complementary ontologies. E.g., a RIF-BLD enabled inference engine that does not hard-code the special semantics of the types of OWL2-RL can import the *RIF-BLD definitions of these types* [13]. KRLO does not define types which are not logic-related, e.g., types for physical quantities or dimensions. Thus, *if* a KRL notation has some syntactic sugar for such types, the notation specification has to reuse types that are not defined in KRLO but in other ontologies. Using them for translation may require special translation methods.

5. CONCLUSION

One contribution of this article was to present the *interest of exploiting an homogeneous ontology of KRL models and notations* when implementing tools importing, translating and exporting knowledge in many KRLs. The beneficiaries of this exploitation are the tool designers and the tool end-users. Homogeneous *domain ontologies* are commonly used instead of rules or procedural code to reduce coding effort and increase the possibilities of *reuse or parameterization*. KRLO permits to do so *for the handling of KRLs*. It can also be used to ease the definition and comparison of KRLs.

Another contribution was to show *how we created such an ontology*, which main ideas we used, and which first tools we designed to help its exploitation. A related underlying contribution is the resulting ontology, KRLO, which i) represents and relates several models and notations into a unified framework, ii) declaratively specifies some import and export methods, and iii) can be extended with additional specifications by Web users.

Our approach provides an ontology-based concise alternative to the use of XML as a meta-language for creating KRLs. Thus, it is also a complement to GRDDL, the W3C recommendation for specifying where a software agent can find tools – e.g., XSLT ones – to convert a given KRL to RDF/XML. This new avenue is important given the frequent need for applications to i) integrate or import and export from/to an ever growing number of models and notations, ii) exploit ad hoc uses of them, and iii) let application end-users adapt them.

Our translation server and its inference engine have recently been implemented by the second author of this article, employee of the software company Logicells. This company will use this work in some of its software products for them to i) collect and aggregate knowledge from knowledge bases, and ii) enable end-users to adapt the input and output formats to their needs. The goal behind these two points is to make these products – and the other ones they interconnect – more (re-)usable, flexible, robust and inter-operable. By itself, our translation server is currently only a proof of concept, not a claimed contribution. Our generic *KRL analyser* is a contribution but, as noted, is not fully in-line with our approach. We shall continue our work on KRLO to i) implement declarative import methods, ii) integrate more abstract models and notations for KRLs as well as query languages and programming languages, and iii) complement our notation ontology by a *presentation ontology* with concepts from style-sheets and, more generally, user interfaces.

6. REFERENCES

[1] Beek, W., Groth, P., Schlobach, S. and Hoekstra, R. 2014. A web observatory for the machine processability of structured data on the web. In *Proceedings of the 2014 ACM conference on Web science*, 249-250.

[2] Borras, P., Clément, D., Despeyrouz, Th., Incerpi, J., Kahn, G., Lang, B. and Pascual, V. 1988. CENTAUR: the system. In *Proceedings of SIGSOFT'88, 3rd Annual Symposium on Software Development Environments* (Boston, USA), 14-24.

[3] Codescu, M., Horozal, F., Kohlhase, M. , Mossakowski, T. and Rabe, F. 2011. Project Abstract: Logic Atlas and Integrator (LATIN). In *Intelligent Computer Mathematics* 2011, LNCS 6824, 287-289. See also http://trac.omdoc.org/LATIN/

[4] Corby, O. and Dieng, R. 1996. Cokace: a Centaur-based environment for CommonKADS Conceptual Modeling Language. In *Proceedings of ECAI 1996* (Hungary), 418-422.

[5] Corby, O. and Faron-Zucker, C. 2015. STTL: A SPARQL-based Transformation Language for RDF. In *Proceedings of WEBIST 2015* (Lisbon, Portugal).

[6] Corcho, Ó. 2004. *A Layered Declarative Approach To Ontology Translation With Knowledge Preservation*, PhD Thesis (311 pages), Universidad Politécnica de Madrid.

[7] DOL 2016. *The Distributed Ontology, Modeling, and Specification Language (DOL).* OMG document. http://www.omg.org/spec/DOL/

[8] Euzenat, J. and Stuckenschmidt, H. 2003. The 'family of languages' approach to semantic interoperability. *Knowledge transformation for the semantic web* (eds: Borys Omelayenko, Michel Klein), IOS press, 49-63.

[9] Hayes P.J. 2006. *IKL guide.* http://www.ihmc.us/users/phayes/IKL/GUIDE/GUIDE.html

[10] ODM 2014. *ODM: Ontology Definition Metamodel, Version 1.1.* OMG document formal/2014-09-02. http://www.omg.org/spec/ODM/1.1/PDF/.

[11] OWL2profiles 2012. *OWL 2 Web Ontology Language: Profiles (Second Edition).* W3C Recommendation. http://www.w3.org/TR/owl2-profiles

[12] OWLinRDF 2012. *OWL 2 Web Ontology Language: Mapping to RDF Graphs (Second Edition)*. W3C Recommendation. http://www.w3.org/TR/owl2-mapping-to-rdf

[13] OWLinRIF 2013. *OWL 2 RL in RIF (Second Edition).* W3C Working Group Note, http://www.w3.org/TR/rif-owl-rl. See also http://www.w3.org/TR/2013/REC-rif-rdf-owl-20130205/

[14] Martin, Ph. 2009. *Towards a collaboratively-built knowledge base of&for scalable knowledge sharing and retrieval*. HDR thesis (240 pages; "Habilitation to Direct Research"), University of La Réunion, France.

[15] Martin, Ph. 2011. Collaborative knowledge sharing and editing. *International Journal on Computer Science and Information Systems*, Vol. 6, Issue 1, 14-29.

[16] RIF-BLD 2013. *RIF Basic Logic Dialect (Second Edition).* W3C Recommendation. Editors: H. Boley, M. Kifer. http://www.w3.org/TR/2013/REC-rif-bld-20130205/

[17] Wielemaker, J., Schreiber, G. and Wielinga, B., 2003. Prolog-Based Infrastructure for RDF: Scalability and Performance. In *Proceedings of ISWC 2003*, LNCS 2870, 644-658.