Specifying Knowledge Representation Notations and Knowledge Exports in Them

Philippe A. Martin[1], Jérémy Bénard[2], and Anil Cassam Chenai[2]

[1] EA2525 LIM, ESIROI I.T., University of La Réunion, F-97490 Sainte Clotilde, France
+ adjunct researcher of the School of I.C.T. at Griffith University, Australia
Philippe.Martin@univ-reunion.fr
[2] GTH, Logicells, 55 rue Labourdonnais, 97400 Saint-Denis, France
{Jeremy.Benard, acc}@logicells.com

Abstract. This article focuses on the knowledge exporting related parts of an ontology-based approach that we designed to reduce the difficulties of automatically importing and exporting knowledge in knowledge representation languages (KRLs). KRLO, the ontology we designed to support this approach, is the first ontology allowing the representation of KRL models and notations in a simple and uniform way. It already represents KRLs of the main different families. It also specifies knowledge export functions or rules, and enables generic parsing. It can be converted in any KRL that has at least OWL2-RL expressiveness and hence be exploited by any inference engine handling it. Thus, our results can be exploited or replicated. This approach requires no or few programming tasks and allows end-users to adapt KRLs. KRLO and translation server based on it are accessible from http://www.webkb.org/KRLs/.

Keywords: Knowledge representation languages (KRLs), Knowledge sharing, KRL syntactic/structural/semantic translation.

1 Introduction

Knowledge representation languages (KRLs) permit to represent information in a logic-based way that can be exploited by inference engines. This eases precision-oriented information sharing, retrieval and problem solving. Many KRLs exist. They have different (abstract) models which follow different logics, e.g., the SHOIN(D) description logic or First-Order Logic. Each model has or may have different notations (aka concrete models or syntaxes), e.g., Notation3 or XML-based notations. In this article, *term* refers to a KRL element, *structure* refers to a term having other terms as parts, *concrete term* refers to a notation element – e.g., a *string* containing a function in a pre-fixed/in-fixed/post-fixed form – and *abstract term* refers to a model element, e.g., a quantifier, a relation, a function or a frame (class or object with its properties).

A unique KRL model (e.g., RDF+OWL) and notation (e.g., SPARQL Update+Query) is not adequate for every kind of knowledge modelling or exploitation, nor for every person or tool. Indeed, representing and sharing complex information, e.g., the content of some natural language sentences, requires a KRL that follows a very expressive logic and has a rich and concise (textual or graphic) notation. Simpler KRLs are easier to learn by people. Developing inference engines or KBMSs – Knowledge Bases (KB) Management Systems – based on simple KRLs is also easier, although extending these tools can then be very difficult. Nowadays, the W3C no longer advocates the use of RDF/XML as the sole notation to use for Linked Data and there are many KRL notations to take into account for knowledge reuse, e.g., KIF-like ones, N3-like ones and XML-based ones. Many KRL standards exist – e.g., CLIF and CGIF of the Common Logics (CL) standard – and many existing KBs (Knowledge Bases) or KB servers – e.g., those of Ontolingua and CYC – are interesting to reuse and are not written in W3C related KRLs. Thus, knowledge translation is useful.

The heterogeneity of existing KRL models and notations – or, more precisely, the absence of *shared primitives for representing their structures and semantics* make their exploitation dependant to structures specific to these KRLs. This explains the difficulty of automatically comparing knowledge from different sources and therefore performing knowledge translation, retrieval, sharing, integration, etc. In a previous article [1], we showed that this difficulty can be reduced by an ontology that i) represents some KRL models and notations in a simple and uniform way, and ii) can be extended by Web users to represent other KRLs in such a way. Thus, we also introduced some underlying ideas of KRLO, an ontology we created for supporting that goal. However, [1] is focused on the content of KRLO for abstract models. This part is summarized in Section 2. This present article, which is thus self-contained, focuses on representations for notations and how they can be used for knowledge export or translation.

E.g., with KRLO, the designers of a KBMS (KB Management System) that can handle OWL2-RL expressiveness do not have to spend weeks or months to *program* knowledge parsing, export or translation for *each* notation or model to handle: they can let their KBMS issue queries exploiting parsing or export functions of KRLO. If a target notation is not already represented in KRLO, they only have to copy and adapt the representation of an existing similar notation. For simple changes, e.g., modifying some term delimiters such as replacing "(" by "{" for the arguments of functions, or adding pre-fixed/in-fixed/post-fixed forms to structured terms, this may be done in a few minutes. These designers can then also let their KBMS end-users perform such simple changes and hence let them use notations suited to their preferences, applications or the tools they reuse. More generally, *exploiting ontologies* – here, a notation ontology – instead of less declarative or organized structures or codes has advantages: flexibility, ease of modification, automatic semantic checking, etc. Many articles have shown this for decades, e.g., since the Ontolingua based research on knowledge translation [2, 3]. Thus, given the space constraints, we do not further expand on those advantages in this article aimed at Semantic Web knowledgeable readers. However, despite those advantages, no ontology for a notation – and then *a fortiori* no ontology of KRL notations – seemed to exist before KRLO. Hence, it fills a void. This article introduces the elements of such an ontology and how they can be exploited. Section 3 explains the used conventions and gives introductory examples. Section 4 lists the underlying ideas for concrete term specification via KRLO and illustrates them. Section 5 expands on this for knowledge export. Section 6 compares our work with other ones.

2 Summary of the Representation of Abstract Terms in KRLO

Three standardizing organizations have given a *language ontology* for the models of the KRLs they advocate. The *W3C* published XML models for RDF+OWL, RIF-FLD [4], ..., as well as translation rules between some pairs of them, e.g., in [5]. The *ANSI* gave a UML model and an XML model for Common Logics (CL) [6]. The *OMG* (Object Management Group) also did so for the Ontology Definition Meta-model (ODM). Since UML and XML are not logic-based KRLs, these ontologies only *declare* some terms, they do not *define* them. ODM 1.1 [7] declares the elements of four KRL models (RDF, OWL, CL and Topic Maps). It has few semantic relations – such as generalization or equivalence relations – *between terms of different models*. These terms are still difficult to compare: the model ontologies remain globally heterogeneous. They are also internally heterogeneous in the sense that the definitions of their elements are not based on a few primitive semantic relations. Thus, for each model, different types of elements must be handled differently and a large number of relation types must be taken into account (instead of only a few combinable primitives, for obtaining the same functionality). This makes reusing these models difficult, even for designing export rules. Setting additional relations between elements of such KRL model ontologies will not solve their global or internal heterogeneity. To solve this problem, KRLO not only represents generalization and equivalence relations between the abstract terms (ATs) of different KRL models (this partially defines their semantics) but also represent their structures in a uniform way (this also partially defines their semantics). Indeed, the structure of each AT is represented like the structure of a function, i.e., as an *operator* with an optional set of arguments and a result. Thus, in KRLO, the four most important primitive relation types for relating ATs, i.e., terms of abstract models) are named r_operator, r_argument, r_result and r_part. The first two are subtypes of the last since the operator and arguments of an AT are also its parts. E.g., in KRLO, a variable or an identifier is *defined* as having for *(r_)operator* a name, no *argument* and for *result* an AT of a certain type. A relation is defined as having for *operator* a relation type, some *arguments* and for *result* a boolean. A value (e.g., a number) is defined as having no argument and having itself as *operator* and *result*. A genuine function is defined as having for *operator* a function type, some *arguments* and a *result*. A quantification is defined as having for *operator* a quantifier, some *arguments* and for *result* a boolean. Thus, in KRLO, an operator may be a function/relation/collection type, a quantifier or a value.

Besides its top-level, KRLO is composed of the specifications of KRLs – or families of KRLs – defined with this top-level. Any inference engine that exploits KRLO can perform some knowledge translation or export: no procedural code is needed for them. Thus, KRL specifications can be i) adapted at any time by the end-users, ii) compared and organized via ontologies, and iii) executed by different inference engines. This offers more possibilities or flexibility to a tool designer and its end-users than procedural programming based approaches. For the *knowledge structural translation or export* permitted by the generalization relations and function-like structural definitions of KRLO, *OWL2-RL* expressiveness is sufficient for defining the structures, including those for higher-order logic terms or equivalence relations such as those between a *frame* and a logical conjunction of binary relations from a same source node. For more complete translations, an inference engine will require – and be able to handle – more complete definitions of the semantics of each exploited AT type. Such definitions can be imported from ontologies that are complementary to KRLO, e.g., for some OWL2 like constructs, the Frame-Ontology of Ontolingua [3]. KRLO also does not include definitions for types which are not logic-related, e.g., types for scalar quantities and physical quantities or dimensions, or types for some process related concepts or relations, or types related to a particular domain. Thus, if a KRL notation has some syntactic sugar for such types, to enable other translations than structural ones, its specification has to reuse types that are not defined in KRLO but in other ontologies.

The top-level of KRLO does not have *direct* equivalence or generalization relations between *all* AT structures that are equivalent but between one expressive form and the other forms. Transitively, translations can be found between all equivalent forms. Thus, for example, to compare representations that use different KRLs, our *KRLO exploiting KBMSs* convert *each AT that uses OWL-like cardinality restrictions or non-binary relations* into an AT that only uses numerical quantifiers and binary relations, then compare it to other ATs and, when necessary for export or other inferencing purposes, translate back ATs to other forms. When the target KRL is not formally expressive enough to represent an AT – e.g., when a statement such as "in 2015, at least 78% of birds in UK could fly, according to ..." has to be translated into RDF+OWL – ad-hoc forms must be used. If the specification of the target KRL describes such forms, our inference engines use them. Otherwise, the source forms are kept and comments or annotations are used to distinguish them from genuine translations in the target KRL.

3 Terminology, Conventions and Introductory Examples

The terminology used in KRLO – and hence in this article – has been chosen to be understandable by the communities interested in information translation, e.g., those related to Model Driven Engineering (MDE), knowledge/ontology sharing/engineering or the Semantic Web. The RDF+OWL related terminology was not suited for KRLs of a higher expressiveness. It is also not used in RIF-FLD [4], the Higher-Order Logic based model recommended by the W3C. For readability and normalisation purposes, KRLO also follows many conventions or best practices. In the next examples, when UML is not expressive enough, FL (Frame Language) [8] is used since i) it is similar to the well-known Turtle or N3 notations but more concise and expressive, and ii) it was used for writing KRLO. This does *not* mean that we advocate its use for all purposes.

We call *link* an instance of a *binary relation type*. Such a type is instance of the 2nd-order type "owl:ObjectProperty" or "owl:DatatypeProperty". As illustrated by these last two identifiers, in the text of this article, as in many W3C notations, ":" is used as separator between the namespace identifier shortcut (here "owl") and the identifier with the namespace. However, in FL and hence in some examples within tables or figures below, the separator is "#" because ":" is used for separating a link and its destination node, as in many frame-based notations. A frame is a statement composed of several links connected to a same source node. A link source/destination node is called a *concept node* and either is a named individual (named term that is not a type) or refers to one or several individuals (e.g., by using a type and a quantifier; in RDF, the quantifier is implicitly the existential quantifier and an unnamed resource node is called a *blank node*). Named terms that are not individuals nor relation types are called concept types (aka *classes* in RDF). In the examples of this article, the default namespace is for the types introduced by KRLO.

For readability purposes, in KRLO and this article, each name for a concept type or individual is a nominal expression beginning by an uppercase letter, as in "Model" and "KRL_Model". To help readers distinguish relation types from concept types *in any notation*, even when they are used in concept nodes, the name of a relation type introduced by KRLO begins by "r_" (in the general case) or "rc_" if this is a type of link having a concrete term (CT) as destination – the only exception is the relation type named "rc" which connects an AT to one of its possible CTs. Thus, in the illustrations of this article, all the names not following these conventions and not prefixed by a namespace are KRL keywords. Within nominal expressions, "_" and "-" are used for separating words. When both are used, "-" connects words that are more closely associated. Since nominal expressions are used for the introduced types, the *convention for reading links in graph-based KRLs* [9] can be used, i.e., links of the form "X R: Y" can be read "X has for R Y". However, when a keyword such a "of" is used for reversing the direction of a link, the form "X R of: Y" should rather be read "X is the R of Y". The syntactic sugar of Formalized English (FE) [10] makes this reading convention explicit.

Table 1 illustrates translations of an English sentence into different KRLs. Different notation families are illustrated, e.g., prefixed notations, frame based infix ones and graphic notations. This example shows the diversity of KRL models and notations, and hence gives a feeling for the difficulty of translating between KRLs. The sentence is a simple definition and use cardinalities restrictions. For the notations not having syntactic sugar for cardinalities restrictions, the OWL2 model is used, except in KIF since this KRL allows to define the numeric quantifier "exactlyN". OWL-Lite would have been sufficient but the qualified cardinality restrictions of OWL2 make the representations more readable. The English sentence is about birds: we reused and adapted a classic example in Artificial Intelligence. It is represented in different KRLs, roughly from the representation most similar to English, to the least similar. Link type names are in italics. The names of some of these KRLs are composed of the model(s) they use, followed by the notation they use. E.g., the *RIF+OWL|RIF-PS* KRL follows the RIF-FLD model plus the OWL ontology (in order to represent cardinalities restrictions) and uses the RIF-PS (RIF Presentation style) notation. We use "|" rather than "/" as separator since "/" is needed for referring to notations, e.g., *RIF-FLD+OWL/RIF-PS* refers to the notation RIF-PS for RIF-FLD(+OWL). Indeed, such a use of "/" has been popularized by the W3C with the RDF/XML notation. This last convention is used in Fig. 1 for illustrating some relations between models and

Table 1. Translations of an English statements in 9 different KRLs.

English: By definition, a "flying_bird_with_2_wings" is a bird
that flies and has two wings.

FE: any Flying_bird_with_2_wings has for *r_type* Bird,
is *r_agent of* a Flight, has for *r_part* 2 Wing.

FL: any Flying_bird_with_2_wings *r_type*: Bird, ***//Version with the structure***
r_type: Bird, *r_agent* of : a Flight, *r_part* : 2 Wing; ***// used in FE.***

Flying_bird_with_2_wings *r_supertype*: ***//Version with normalized structure,***
{ Bird ^(Thing *r_agent* of: a Flight) ***// i.e., one close to the structures***
^(Thing *r_part*: 2 Wing) ***// followed in UML below and those***
} _[. -> complete .]; ***// followed when OWL is used below.***

CGLF: type Flying_bird_with_2_wings (*x)
[[Bird:*x]-{ ->*(r_agent)*->[Flight] ; ->*(r_part)*->[Wing:{*}@2]; }]

KIF: (defrelation Flying_bird_with_2_wings (?x) :=
(exists ((?f Flight)) (and (Bird ?x) (*r_agent* ?f ?x)
(*exactlyN* 2 '?w Wing ^(*r_part* ,?x ?w)))))

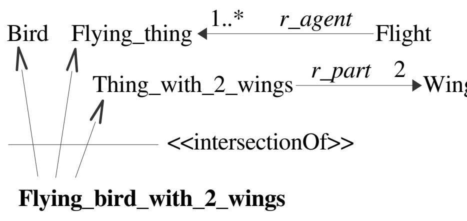
OWL Manchester: Class: :Flying_bird_with_2_wings
EquivalentTo: Bird and *r_agent* some Flight and *r_part* exactly 2 Wing

OWL Functional-style: EquivalentClasses (Flying_bird_with_2_wings
ObjectIntersectionOf (Bird ObjectSomeValuesFrom (*:r_agent* :Flight)
ObjectExactCardinality (2 *:r_part* :Wing)))

RIF+OWL/RIF-PS: Forall ?x *r_logic_implication* (
?x#Flying_bird_with_2_wings
And(Exists ?f (And(?f#Flight ?x#Bird *r_agent*(?f ?b))
Exists ?t (And(?f#?t ?t#owl:Restriction *owl:onProperty* (?t *r_part*)
owl:qualifiedCardinality (?t 2) *owl:onClass* (?t Wing)))))

RDF+OWL/N3: Flying_bird_with_2_wings owl:intersectionOf
(Bird [*rdf:type* owl:Restriction; *owl:onProperty* r_agent; *owl:someValuesFrom* Flight]
[*rdf:type* owl:Restriction; *owl:onProperty r_part; owl:qualifiedCardinality 2;*
owl:onClass Wing]) .

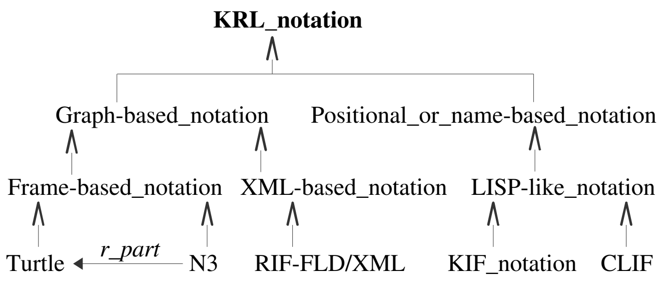
Slightly adapted UML:



Legend: the arrow —> represents a supertype (subClassOf) link while the other kind of arrow (—▸ with an associated link type in italics and a destination cardinality if different from 0..*, alias 0–N) is used for the other links. For readability purposes, the boxes around classes (types) and associations (links) are not drawn.

notations. Apart from KIF, "OWL Fct.-style" (OWL Functional Style) and RIF-PS, all notations in Table 1 are graph-based: they directly show the concept nodes – and relation nodes relating them – of a graph-based model. Apart from UML, the graph-based notations below are, at least sometimes, frame-based: the order of their concept nodes may be important for understanding them. E.g., CGLF [9] is frame-based for type definitions (as shown by Fig. 2) but not for other statements, e.g., for a definition body. A notation that is not graph-based is positional or name-based: the concept nodes appear as positional or named arguments of a relation node which looks like a function call in traditional programming languages. No example of named argument is given in Table 1: the respective positions of the arguments are therefore important.

Fig. 1 shows some subtype and r_part relations between some KRL notations. In KRLO, this is also done for KRL models. This permits to organize and compare notations or models, families of them, and hence also to modularize KRL specifications. Fig. 1 illustrates one subtype partition, i.e., one complete and disjoint set of subtypes for a type. It also gives one example of r_part link: the one between (any instance of any version of) Notation3 (N3) and Turtle.



Legend: same as in Table 1 for the *Slightly adapted UML* plus, still for readability reasons, i) since each *subclass set* is here a *subclass partition*, its "{disjoint, complete}" annotation is left implicit, and ii) the associations are in italics and their cardinalities are not displayed since they are all 0-N in both directions. Thus, the above r_part association can be read: "any (version of) N3 has for *part* 0 to many (versions of) Turtle, and conversely".

Fig. 1. *Slightly adapted UML* representation of some relations between KRL notations.

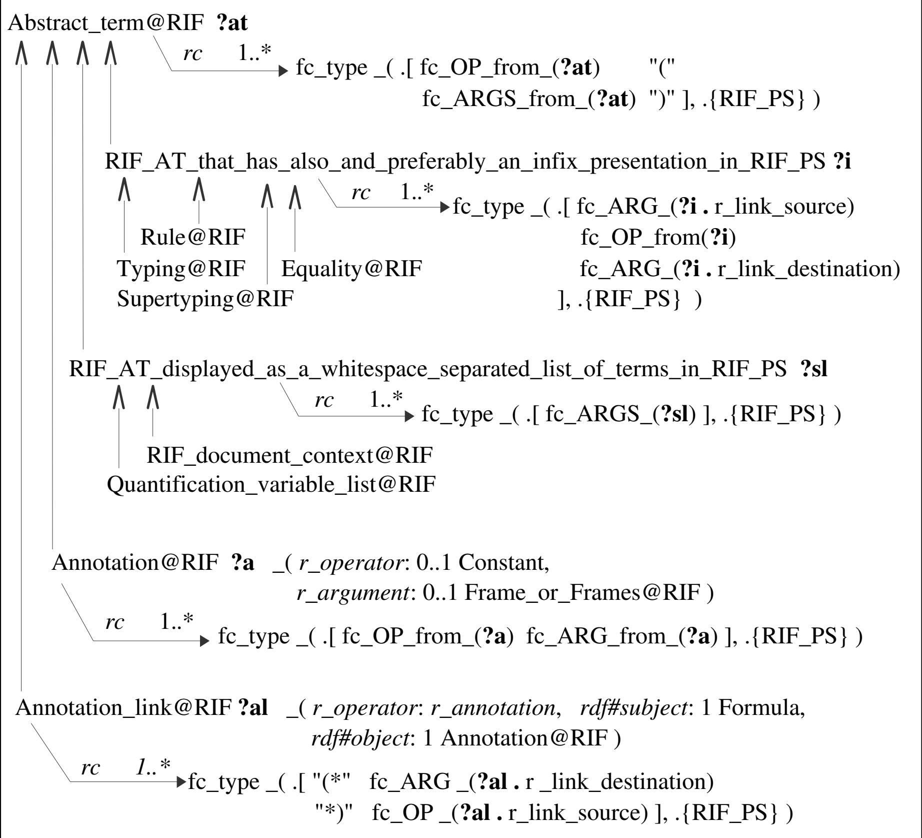
4 Concrete Term Representation in KRLO: Ideas and Examples

In all the KRLs we know, when each AT is defined as an operator with arguments, we noted that the ways to (re)present this AT – or, more precisely, the syntactic structures of the concrete terms (CTs) for this AT – could be specified in a generic way in a LL(1) or LALR(1) grammar, hence a deterministic context-free grammar that can be efficiently parsed. This was an important discovery because this meant that i) we could build one efficient generic parser for all these KRLs, and ii) the primitives of the notation ontology of KRLO could be the representations of these structures. As an example for this idea, consider an AT composed of an operator "o" with two arguments "x" and "y". If parenthesis are mandatory delimiters and if spaces are the only usable separators, this AT has only the next five possible CTs (not counting optional uses of spaces and parenthesis) in all the notations we know: "o (x y)" (function-like prefix construct as in RIF-PS), "(o x y)" (list-like prefix construct as in KIF), "(x o y)" (infix construct as in Turtle and some RIF-PS statements), "(x y o)" (list-like postfix construct), "(x y) o" (function-like postfix construct). Five rules of an LL(1) or LALR(1) grammar can be used for specifying these five possibilities and they can also 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 in the non-lexical part via terminal symbols such as Placeholder_for_begin-mark_of_the_arguments_of_a_prefix-function-like_term and Placeholder_for_end-mark_of_the_arguments_of_a_postfix-list-like_term. In the lexical part, it is also possible to specify rules for detecting various kinds of tokens and various kinds of separators rather than specific ones. Thus, using Flex&Bison (GNU variants of Lex&Yacc), we created a *generic parser for KRLs that can have an LALR(1) grammar* (which does not have to be found since it is generalized by the generic LALR(1) grammar that our parser uses). This grammar restriction did not prevent us to let FL have all the potentially useful prefix/infix/postfix constructs we could think of. It simply led us to associate explicit and unambiguous syntactic sugar to signal the beginning and end of each of these constructs, e.g., "_( )" to enclose classically prefixed function call parameters, "(_.)" to enclose LISP-Like prefixed function call parameters, "(_)" to enclose postfixed function call parameters and ".( )" or ".[ ]" to enclose genuine lists. Such marks also make these constructs unambiguous for people too.

The top-level of the *ontology of notations of KRLO* does not categorize all possible construct types: it simply contains the primitive relations permitting to describe them. Indeed, we found that there were two many possible combinations of these primitives for a categorization to be helpful. Since the primitives also proved too cumbersome to be used directly, we defined intermediary functions accepting list-based descriptions. Such functions – e.g., fc_OP and fc_ARG – will be illustrated in Fig. 2.

Given some CTs and the notation in which they are written, our parser exploits the specification of the CTs of this notation – and the specification of the ATs to which they are connected – to build data structures storing ATs that our inference engines exploit (since our implementation of this parsing is currently rather ad-hoc and not the focus of this article, it will not be further detailed in this article). Conversely, we wrote functions which exploit AT and CT specifications to generate CTs in a given KRL (model and notation) for ATs. The next section will provide more details. The functions translating ATs between different abstract models are not the focus of this article. All functions of KRLO are declarative and can be automatically converted into rules or (non-binary) relation type definitions for inference engines that do not handle functions but only rules or type expansion/contraction.

Fig. 2 gives examples of five specifications for RIF-PS CTs. Each one uses a link of type "rc" which, for the instances of a certain type of AT, defines the (default) types of CTs that can be used for presenting these instances in the RIF-PS notation.


Abstract_term@RIF ?at
rc 1..*
fc_type _(.[fc_OP_from_(?at) "(" fc_ARGS_from_(?at) ")"], .{RIF_PS})
RIF_AT_that_has_also_and_preferably_an_infix_presentation_in_RIF_PS ?i
rc 1..*
fc_type _(.[fc_ARG_(?i . r_link_source) fc_OP_from(?i) fc_ARG_(?i . r_link_destination)], .{RIF_PS})
Rule@RIF
Typing@RIF
Equality@RIF
Supertyping@RIF
RIF_AT_displayed_as_a_whitespace_separated_list_of_terms_in_RIF_PS ?sl
rc 1..*
fc_type _(.[fc_ARGS_(?sl)], .{RIF_PS})
RIF_document_context@RIF
Quantification_variable_list@RIF
Annotation@RIF ?a _(r_operator: 0..1 Constant, r_argument: 0..1 Frame_or_Frames@RIF)
rc 1..*
fc_type _(.[fc_OP_from_(?a) fc_ARG_from_(?a)], .{RIF_PS})
Annotation_link@RIF ?al _(r_operator: r_annotation, rdf#subject: 1 Formula, rdf#object: 1 Annotation@RIF)
rc 1..*
fc_type _(.["(*" fc_ARG _(?al . r _link_destination) "*)" fc_OP _(?al . r_link_source)], .{RIF_PS})

RIF-PS example of annotated phrase with 2 quantifiers, 1 frame of type Flight and 1 rule:
(* if ?ft is a Flying_thing, there is 1..* Flight with r_agent ?ft *)
Forall ?ft (Exists ?f ?f # Flight [r_agent -> ?ft] :- ?ft # Flying_thing)

Legend: i) same notes as in Fig. 1, ii) "1..*" means "at least 1", iii) the FL notation is here used for variables, functions, lists and the operators "@" and "." (→ *Extended UML*).
The FL operator "." permits to refer to the destination of a link, as with "property paths" of SPARQL, but in a concise manner as with object-oriented languages.

Fig. 2. *Extended UML* specification of RIF-PS concrete terms for some RIF abstract terms.

The first rc link starts from "Abstract_term@RIF". This is not an identifier like "rif:Abstract_term" but an FL expression referring to *the subtype of KRLO:Abstract_term for RIF*, i.e., the type that generalizes all ATs in the RIF model. The definition of the "@" FL operator is not given here because of space constraints (this would require explaining its FL syntax) and because it is only an abbreviation for a certain combination of subtype and r_member relations from a type to a set of types. However, the interested reader can find its definition in the specification of FL in KRLO. This abbreviation can be defined in any KRL having the expressiveness of OWL2-RL.

Before paraphrasing the first rc link in English, here is it global purpose. It specifies that any instance of Abstract_term@RIF has (at least, by default) RIF-PS CTs which are composed of the following sequence of elements, separated by at least one spacing character: 1) the (concrete) representation (in RIF-PS) of the operator of the AT, 2) an opening parenthesis, 3) the representation of its arguments separated by at least one spacing character, 4) a closing parenthesis. A paraphrasing of this first rc link in English is: if "?at" is an instance of the RIF type for Abstract_term (in FL, classic variable names begin by "?") then "?at" has for rc (concrete representation) at least one CT of the type returned by the function fc_type with the specified arguments.

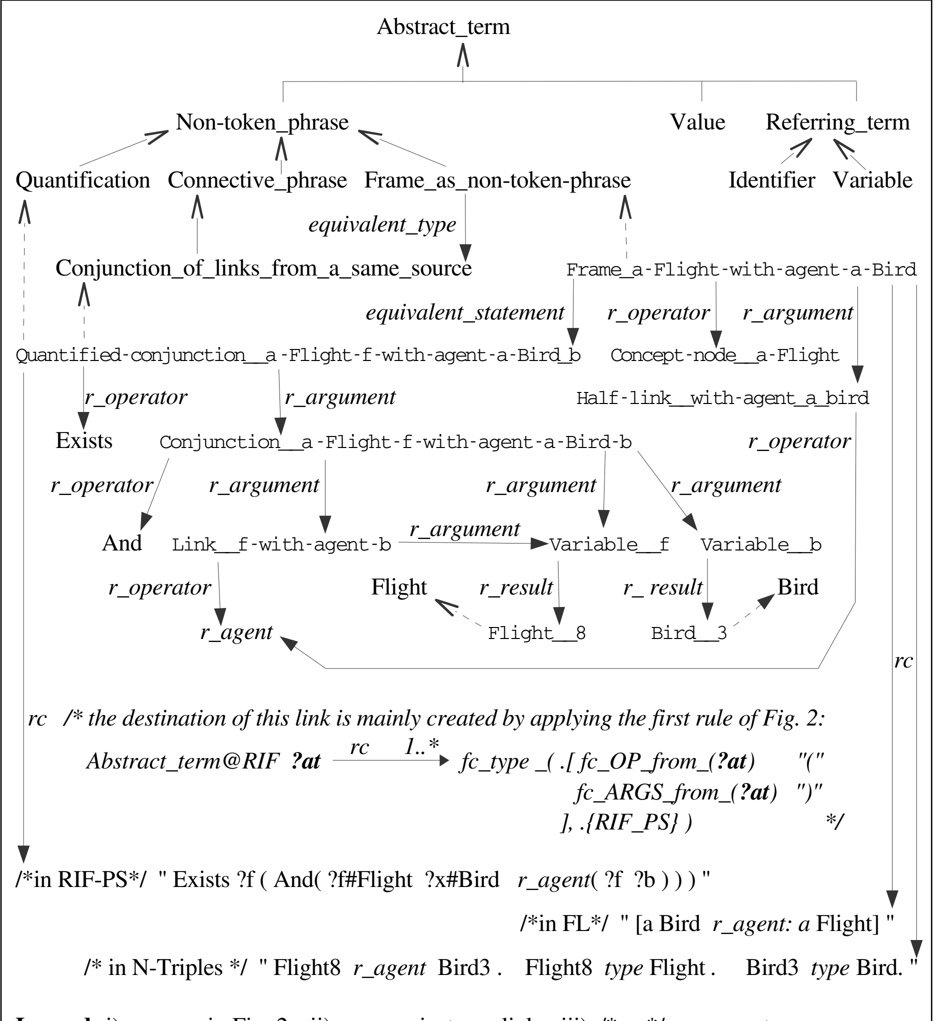
The first argument of this fc_type function is a square bracket delimited list of four terms (here, CT specifications). The first and third terms of this list are results of the functions fc_OP_from and fc_ARGS_from, defined in KRLO. They are similar to fc_type but they i) respectively work on the operator and arguments of their AT parameter, and ii) permit to know the role of each term in the specified list: operator, argument or separator. Thus, for parsing purpose, the grammar rule to use for this list of terms can be selected. The function fc_ARGS_from may have a second argument for specifying a non-space separator to use between the arguments.

The second argument of the above cited fc_type function is a set of notation types. Indeed, fc_type returns a type of CTs that are *member* of notations of such types. A set is used because an AT may have identical types of CTs in different notations.

Abstract_term@RIF has many subtypes that do not follow this first (above explained) presentation rule or not just this one. For each subtype, another overriding or complementary specification is then associated. This is why the second rc link of Fig. 2 specifies an infix presentation for some links. Fig. 2 shows that four types of links have this default infix presentation in RIF-PS: those expressing rules, supertypes, types or equality. If RIF-PS did not *also* allow the prefix presentation to be used for these four types of links, Fig. 2 would have had to specify that. In KRLs reusing OWL, this would have been translated using the owl:allValuesFrom type. The third rc link specifies the RIF-PS representation of some parts of ATs such as quantification variables and context-related parts of the Document@RIF AT. The last two rc links specify the RIF-PS representation of an Annotation link, i.e., the RIF-PS CT for the Annotation within special begin and end marks, followed by the CT for the annotated AT. Fig. 2 shows that in RIF an Annotation is composed of an optional Constant and optional Frames.

5 Export Functions and Example

In KRLO, each AT has one and only one (inherited or overriding) rc link for a given type of notation. Indeed, our knowledge server prevents the entering of ambiguities when it detects them. Thus, for a given AT and notation, the multiple possible presentations – i.e, the corresponding CTs – are declaratively and unambiguously described. This description is recursive when a *CT specification* refers to a *component AT* (via the parameter of functions such as fc_OP or fc_ARGS) since this AT has itself a *CT specification*. The spacing between CTs is similarly defined in a declarative and unambiguous way, if only via the default presentation specifications of KRLO. Thus, although KRLO uses functions for generating presentations, they can be automatically converted into rules or type definitions for inference engines that do not handle functions. The export functions of KRLO simply perform the recursive exploration of the specifications and the concatenation of the resulting CTs. The export process is



rc /* the destination of this link is mainly created by applying the first rule of Fig. 2:

Abstract_term@RIF ***?at*** $\xrightarrow{rc \quad 1..*}$ *fc_type _(.[fc_OP_from_(**?at**)* "(" *fc_ARGS_from_(**?at**)* ")" *], .{RIF_PS})* */

/*in RIF-PS*/ " Exists ?f (And(?f#Flight ?x#Bird *r_agent*(?f ?b))) "

/*in FL*/ " [a Bird *r_agent: a* Flight] "

/* in N-Triples */ " Flight8 *r_agent* Bird3 . Flight8 *type* Flight . Bird3 *type* Bird. "

Legend: i) same as in Fig. 2, ii) - - -►: instance link, iii) /* ... */ : comment,
iv) the identifiers of individuals are in courier font; here, they are all generated by concatenating a name for a type of the individual, "__", a number or a FE-like expression for what the individual describes.

Fig. 3. *Extended UML* specification of relations between some abstract terms and some concrete terms for the representation of "A bird flies" (in FL: [a Bird *r_agent: a* Flight]).

semantic preserving (and correct if the links are correct). It is complete (if what is expressed by the links is complete) since there is one and only one rc link (inherited or overriding) rc link for a given type of notation.

Fig. 3 shows relations between some (types or individuals for) ATs and CTs involved in the representation or export of "A bird flies" in RIF-PS, FL and N-Triples. This statement is represented both via a frame and a conjunction of links from a same source node (with existentially quantified nodes), two semantically equivalent forms.

6 Comparison To Other Works

KRLO and the LALR(1) parser based on it can be categorized as a *system that helps design tools handling many different models and notations*. Other ones are: classic *parser generators* (such as Lex&Yacc), *interactive programming environment* generators (e.g., Centaur [11]), structure+presentation model based tools (e.g., XML+XSLT+CSS based tools and MOF/UML based tools) and KRL model based tools. The specificity of our system is to be fully ontology based. This makes it the most i) *code-independent* in the sense that different inference engines can *run* KRLO, and ii) *extensible by its designers and end-users* (both can specify or adapt any model or notation they wish to use). In this article aimed at Semantic Web knowledgeable readers, and given the space constraints, we do not attempt to show – via formal proofs or empirical measures – how the previous sentence derives from the one preceding it. We refer the interested reader to books or proceedings related to "Semantic Web Enabled Software Engineering", e.g., [12]. However, like other systems, KRLO does not permit a tool to better exploit more expressive constructs than those it is already designed to. E.g., KRLO can permit tools to use models and notations representing transitivity in various ways but it does not permit tools to better exploit this notion than they already can. The following paragraphs examine the notation related systems from the above cited category, starting from those providing the least flexibility to the end-users.

Classic *parser generators* are given a concrete grammar (hence, a concrete model) with *actions* associated to its rules to build an abstract model. Additional functions and rules have to be created for translation or export purposes. All of them are sensitive to changes in the syntax or semantics of the languages used for inputs or outputs. They cannot be organized into an ontology, for comparison or reuse purposes. To sum up, creating and updating these functions or rules are – or are akin to – programming tasks, hence long and error-prone. Yet, this is how import/export features are implemented in the current KBMSs and KRL translators we know of.

These programming tasks are facilitated and generalized via the use of an *interactive programming environment* generator. E.g., Centaur [11] could generate a structured editor, a parser, a type checker and an interpreter or a compiler for a specified language. To that end, its concrete grammar, its abstract grammar and the translating rules between them had to be specified in the Metal language (they were then converted into Lex&Yacc). Centaur has been used mainly for programming languages but also for one

KRL [13]. However, the above cited languages are execution oriented rather than modelling oriented. They do not ease the creation and reuse of ontologies. The specifications are then still difficult to organize and reuse. Small changes in the concrete and abstract grammars often lead to important changes in the specifications. Creating them is still a bit akin to programming even though no low-level management of objects has to be done. Finally, translations rules have to be specified for each pair of languages. On the other hand, KRLO is still far from having declarative specifications for all the features that a tool such as Centaur provides via its procedural code.

Another approach is the use of notations that make the structure of a language model explicit and hence easy to parse and check via rather generic tools. This is for example the case with XML, MOF-HUTN (the Human Usable Textual Notation for the Meta-Object Facility), XMI (an XML notation for MOF models) and other structure description languages of the Model Driven Engineering (MDE) approach. However, concrete descriptions are then often not concise enough or high-level enough to be used directly by people for knowledge display/entering or by tools for knowledge handling. E.g., which KBMS or inference engine uses XML objects internally? Translations from/to other models or notations are still necessary. This approach does not eliminate the need to create parsers and export functions or rules for various notations. Some standard languages are provided for creating these export functions or rules, e.g., XSLT and CSS. However, they cannot perform logical inferences and are not ontology-based. Like this approach, ours permits to use one generic parser but the set of possible input languages is less restricted (XML is just one of many possible notations) and all KRLs having the expressiveness of OWL2-RL could be used for the model and notation specification.

For semantic translations between KRLs or KRL models, researchers soon found that pairwise translations was not a scalable approach and hence that translating to an expressive standard KRL or KRL model was necessary. This led to the creation of KIF (Knowledge Interchange Format) [14] in 1992 and then to the Ontolingua [2,3] library of ontologies. However, since KIF was the de-facto interlingua, Ontolingua does not represent various KRL models nor then relations between KRL ATs. Conversely, KRLO does not yet include the content of Ontolingua. Thus, both are complementary.

Unlike ODEDialect [15], KRLO is not a *language* permitting to express transformations between KRLs and it does not distinguish lexical, syntactic, semantic and pragmatic translations. It is an ontology of KRLs and hence a library of KRL ontologies represented via the same primitives. Using them and adding a new KRL specification to KRLO (e.g., via its Web servers) is sufficient for also specifying its translation to other represented KRLs, no transformation rules is required. No specific *language* is necessary: KRLO can be represented in any KRL that has at least OWL2-RL expressiveness. Finally, a link in KRLO (e.g., an equivalence link between two types of terms) can be used in translations that can be seen as lexical, syntactic, semantic or pragmatic.

Besides our own, we have not found works on notation ontologies – nor then on a generic parser based on them – even for a standard KRL model such as RDF. Therefore, it appears there are also no other model+notation ontology based translator. There are translators between RDF-based notations, e.g., RDF-translator and RDF2RDF. They necessarily involved a lot of programming. Their Web interfaces propose no way to parameter them. On the other hand, there have been several works on style-sheet based transformation languages or, more generally, rules to specify how RDF ATs may be presented, e.g., in a certain order, in bold, in a pop-up window, etc.: Xenon [16], Fresnel [17], OWL-PL [18] and SPARQL Template [19]. These tools were not initially meant to use a notation ontology: they initially required the use of a new template or 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.

7 Conclusion

KRLO is an ontology of KRLs, the first one representing the structure and semantics of the abstract models of different KRLs, thus aligning or organizing them and their abstract terms (ATs). It is also the first one to include a notation ontology and thus represent notations in addition to models and related to them. This article summarized the AT related *ideas and primitives* of KRLO and then presented the ideas and primitives related to concrete terms (CTs) as well as their links to ATs. Thereby, it illustrated the interest of the operator+arguments structural model: it enables the representation of ATs and CTs using a small set of primitives. This permitted us to create a *generic parser for KRLs*. The constraint is that the notations are represented using the primitives of KRLO and hence that they *can have* an LALR(1) grammar (which does not have to be found since it is generalized by the generic LALR(1) grammar that our parser uses). Conversely, this also permitted us to represent export functions or rules that can generate a unique CT given an AT and a target notation.

Combining this generic parsing and these export functions permits to perform *knowledge structural translation*, i.e., the translations that can be made given the semantic links between terms in KRLO: generalization or equivalence links, structural links (e.g., r_part and r_result) and presentation links (e.g., rc). This includes translations between ATs following different models, e.g., OWL versus CL. These translations are semantic preserving and complete w.r.t. what is expressed by the links. If needed, complementary ontologies (e.g., domain ontologies), hence additional relations, can be used to support translations for other information. Specifying a KRL into KRLO is sufficient to enable translations of this KRL to other ones, no transformation rules needs to be specified. No procedural code is required. Amongst systems helping to handle many languages, being *fully* based on a KRL ontology, ours is the most *code-independent* (different inference engines can run it) and *extensible by its designers and end-users*.

Our approach provides an ontology-based concise alternative to the use of XML as a meta-language for creating KRLs that follow KRL ontologies. It therefore also complements GRDDL and can be seen as a new research avenue (GRDDL permits to specify where a software agent can find tools – e.g., XSLT ones – to convert a given KRL to RDF/XML). This 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 (XML-based or not), and ii) let the users parameter these notations.

Our translation server (accessible from http://www.webkb.org/KRLs/) and its inference engine have recently been implemented by the second and third authors of this article, of the software company Logicells. This company will use this work in some of its applications to enable them to i) collect and aggregate knowledge from knowledge bases, and ii) enable end-users to adapt the input and output formats they wish to use or see. The goal behind these two points is to make these applications – and the ones they relate – more (re-)usable, flexible, robust and inter-operable.

We intend to further work on KRLO by integrating more abstract models and notations for KRLs as well as query languages and programming languages. We also intend to complete our notation ontology by a *presentation ontology* with concepts from style-sheets and, more generally, user interfaces.

Acknowledgments. We are grateful to Dr. Olivier Corby (INRIA & I3S, Sophia Antipolis) for our fruitful discussions on this work. E.g., they helped us clarify the presentation of the underlying ideas of this work.

8 References

1. Bénard J., Martin Ph.: Improving General Knowledge Sharing via an Ontology of Knowledge Representation Language Ontologies. In: CCIS 553, pp. 364–387 (Chapter 23). Springer, Heidelberg (2015)
2. Gruber, T. R.: A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, 5, 2, 199-220 (1993)
3. Farquhar, A., Fikes, R., Rice, J.: The Ontolingua Server: a tool for collaborative ontology construction. International Journal of Human-Computer Studies, 46, 6, Academic Press, Inc., MN, USA. (1997)
4. Boley, H., Kifer, M.: RIF Framework for Logic Dialects (2nd edition). W3C Recommendation. Boley, H., Kifer, M. (eds.), http://www.w3.org/TR/2013/REC-rif-fld-20130205/ (2013)
5. de Bruijn, J., Welty, C.: RIF, RDF and OWL Compatibility (Second Edition). W3C Recommendation, http://www.w3.org/TR/2013/REC-rif-rdf-owl-20130205 (2013)
6. Information technology – Common Logic (CL): a framework for a family of logic-based languages. ISO/IEC 24707:2007(E), JTC1/SC32 (2007)
7. ODM: Ontology Definition Metamodel, Version 1.1. OMG document formal/2014-09-02, http://www.omg.org/spec/ODM/1.1/PDF/ (2014)
8. Martin, Ph.: Towards a collaboratively-built knowledge base of&for scalable knowledge sharing and retrieval. HDR thesis (240 pages), University of La Réunion, France (2009)
9. Sowa, J.F.: Conceptual Graphs Summary. In: Conceptual Structures: Current Research and Practice, Ellis Horwood, pp. 3–51 (1992)
10. Martin, Ph.: Knowledge representation in CGLF, CGIF, KIF, Frame-CG and Formalized-English. In: ICCS 2002, LNAI 2393, pp. 77–91. Springer, Heidelberg (2002)
11. Borras, P., Clément, D., Despeyrouz, Th., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: CENTAUR: the system. In: SIGSOFT'88, 3rd Annual Symposium on Software Development Environments (SDE3), Boston, USA, pp. 148–24 (1988)
12. Pan J.Z., Zhao Y. (eds.): Semantic Web Enabled Software Engineering. IOS Press (2014)
13. Corby, O., Dieng, R.: Cokace: a Centaur-based environment for CommonKADS Conceptual Modeling Language. In: ECAI'96, Budapest, Hungary, pp. 418–422 (1996)
14. Genesereth, M., Fikes R.: Knowledge Interchange Format, Version 3.0, Reference Manual. Technical Report, Logic-92-1, Stanford Uni., http://www.cs.umbc.edu/kse/ (1992)

15. Corcho, Ó.: A Layered Declarative Approach To Ontology Translation With Knowledge Preservation. PhD Thesis (311 pages), Universidad Politécnica de Madrid (2004)
16. Quan, D.: Xenon: An RDF Stylesheet Ontology. In: WWW 2005, 14th World Wide Web Conference (Japan) (2005)
17. Pietriga, E., Bizer, C., Karger, D., Lee, R.: Fresnel: A Browser-Independent Presentation Vocabulary for RDF. In: ISWC 2006, LNCS 4273 . Springer, Heidelberg (2006)
18. Brophy, M., Heflin, J.: OWL-PL: A Presentation Language for Displaying Semantic Data on the Web. Technical report, Lehigh University (2009)
19. Corby, O., Faron-Zucker, C.: STTL: A SPARQL-based Transformation Language for RDF. In: WEBIST 2015, 11th International Conference on Web Information Systems and Technologies, Lisbon, Portugal (2015)