Improving General Knowledge Sharing via an Ontology of Knowledge Representation Language Ontologies

Jéremy Bénard[1] and Philippe Martin[2,3(✉)]

[1] GTH, Logicells, 3 rue Désiré Barquisseau, 97410 Saint-Pierre, France
jeremy.benard@logicells.com
[2] EA2525 LIM, ESIROI I.T.,
University of La Réunion, 97490 Sainte Clotilde, France
Philippe.Martin@univ-reunion.fr
[3] School of ICT, Griffith University, Queensland, Australia

Abstract. Via a comparison of the currently used language-based components for knowledge sharing, this article first highlights the difficulties caused by the inexistence – and hence absence of exploitation – of a shared core ontology of knowledge representation languages (KRLs), i.e., (i) an ontology of KRL abstract models which represents, aligns and extends standards, and (ii) an ontology of KRL notations. For programmers, these are the difficulties of importing, exporting or translating between KRLs; for end-users, the difficulties of adapting, extending or mixing notations. This article then shows how we have built this shared core ontology plus a tool for exploiting it. We use them for specifying, parsing and translating KRLs, thus allowing their use without additional programming. This ontology can be represented in any KRL that has at least OWL-2 expressiveness. Thus, the results can easily be replicated. A Web address for the tool and the full specifications is given.

Keywords: Language ontology · Meta-modelling · Syntactic translation · Knowledge representation languages · General knowledge sharing

1 Introduction

The term "knowledge representation language" (KRL) may refer (at least) to one of the next notions or their combination: (i) *a semantic model*, i.e., a set of types (alias, an ontology) specifying semantic and/or logical notions, e.g., those of the SHOIN(D) description logic, (ii) *an abstract "data type" (ADT) model*, i.e., an ontology for ADTs such as abstract syntax trees or abstract semantic graphs (e.g., most types in the OWL-DL ontology form an ADT model since they permit to store a graph that only contains binary relations and follows a semantic model for SHOIN(D)), and (iii) a *concrete model*, i.e., a textual/graphical/… KRL *notation*, e.g., Turtle and OWL-Functional-Style. Knowledge sharing (KS) involves many tasks, some of which are notation related: knowledge editing, parsing, importing, exporting, translating, etc. Section 2 compares "approaches implementing these tasks" depending on the language they offer for specifying the ADT model: (i) a notation grammar, e.g., EBNF, (ii) a

© Springer International Publishing Switzerland 2015
A. Fred et al. (Eds.): IC3K 2014, CCIS 553, pp. 364–387, 2015.
DOI: 10.1007/978-3-319-25840-9_23

generic language for creating ADTs, e.g., XML and MOF-HUTN, and (iii) an ADT ontology.

The first approach implies creating a concrete grammar with *actions* associated to its rules, and giving it to a parser generator, e.g., Lex & Yacc. The actions contain functions or rules to build a less syntax-oriented ADT and build a semantic model (directly or not: a parser/translator exploiting rules and a grammar for the semantic model may be used). Even though an "interactive programming environment" generator (e.g., Centaur) may provide parsers, editors, checkers and interpreters or compilers for specified languages, creating these specifications involves – or is akin to – programming. These specifications (grammars and building/translation rules or functions) are difficult to compare, extend and re-use: they cannot be organized by specialization relations. Small changes in the concrete/ADT/semantic models often lead to important changes in the specifications. Translations rules or procedures have to be specified for each pair of languages.

With the second approach, specifying an ADT grammar (e.g., an XML schema) permits to use a concrete parser or editor and specify various presentations (e.g., via CSS and XSLT). However, since the concrete descriptions must then have an explicit structure, they are too bulky to be used *directly* for building or displaying (parts of) programs or knowledge bases, especially with standards in this approach (e.g., XML). Thus, more adapted textual/graphic KRL notations are required and parsers for them are still needed. Furthermore, since these languages are not based on logic, the tools based on them cannot perform logical inferences, hence cannot exploit knowledge representations.

With the third approach – i.e., the use of ontologies on notations, ADTs and semantic elements instead of just grammars on them – the difficulties of the previous two approaches can be reduced. So far however, (i) there were no ontologies for notations, and (ii) the ontologies on KRL abstract models (ADTs and/or semantics) were implicit (i.e., informally or insufficiently described) or about KRLs of insufficient expressiveness for representing many other KRLs (hence for general knowledge representation and sharing). Thus, these last ontologies were also not inter-related. This article shows how (i) we created a core for an "ontology of KRL ontologies" by representing, aligning and extending the major "KRL abstract models", and then extending it with a KRL concrete model ontology, and (ii) we have begun to use/extend it for building an organized library of declarative concise "KRL specifications". This "ontology of KRL ontologies" supports – and also specifies (i.e., provides the declarative code for) – a *generic tool for parsing and exporting KRLs,* hence also performing (many) translations between them. Since programming is avoided and since KRLs or KRL modifications can be specified in a concise way, even the end-users of applications using such a tool can adapt the format of input and output KRLs to their needs or preferences. This ontology, as well as a Web server interface to use this tool, are available from http://www.webkb.org/KRLs/.

Section 3 introduces the notation used in this article for the illustrations.

Section 4 shows some relations between top-level "language elements" and more general concepts, as well as between notation models and abstract models.

Section 5 explains the main primitive concepts and relations which permit to represent and relate the various models in a uniform and concise way.

Section 6 illustrates their use for the general "abstract model" parts of our ontology.

Section 7 illustrates their use for specifying particular KRLs and even grammars. Representing grammars shows the generality of the approach and permits to re-use existing parser generators. However, our tool currently only uses the LALR(1) parser generator "Gold"; thus, it cannot parse KRLs requiring a more expressive grammar.

2 Comparison of the Various Language-Based Approaches for KS

The model-related terminology used in this article is compatible with the terminology used in syntax or semantic related works. To be compatible with the terminology used in Model Driven Engineering (MDE) related works, especially the one related to MOF (the Meta-Object Facility of the OMG: Object Management Group), the prefix "meta-" would need to be systematically added before "model" since in MDE a document or code is a "model" and it follows the specifications of a "meta-model", e.g., an XML schema. For clarity purposes, in this article, a "meta-model" – i.e., a language to specify a language – refers to what is called a "meta-meta-model" in MDE.

The introduction noted that a model may be an "*abstract model*" (e.g., OWL-DL or an XML schema; its *specifies abstract structures* of a certain type) or a "*concrete model*" (e.g., a formal grammar or a CSS script; it *specifies concrete structures* of a certain type). A concrete model, alias a presentation model, specifies either a formal presentation (it is then a notation) or an informal one (e.g., that certain kinds of ADT elements should be presented with bold characters). CSS and EBNF are therefore meta-models for concrete models. MOF is a meta-model for – and a subset of – the UML model. Since UML also refers to a notation, it is both an abstract and concrete model. Even graphical notations implicitly or explicitly follow a grammar [1]. XML is a meta-model for certain ADT models and concrete models. Meta-models are also models.

This terminology permits to compare language-based approaches based on the kind of meta-model they exploit: a notation-dependent one, a structure-dependent one and a logic-based one. In each case, the next subsections show that problems comes from a lack of expressiveness of the meta-model: there are some notions that it allows to *declare* (in a model) but not to *define*. Thus, tools based on this meta-model cannot exploit the semantics of these notions. Other tools that know this semantics are needed for exploiting it. Furthermore, these notions are represented in different and ad-hoc or implicit ways across models, thus making knowledge sharing and re-use more difficult. As can be concluded at the end of the next subsections, the use of an ontology of KRL ontologies, based on higher-order logics, is necessary to avoid problems.

2.1 Exploiting Notation-Dependent Meta-models

The grammar directed parsing of a textual/graphical description leads to a concrete structure – e.g., a concrete syntax tree if a context-free concrete grammar is used – and an ADT – e.g., an "abstract syntax tree" or an "abstract semantic graph". To derive the abstract structure from the concrete one, an “action” – containing a transformation rule or function – is generally associated to all or most concrete grammar rules. Then, static semantic checking (type checking, context-dependent interactions), dynamic semantic checking (interpretation, debugging) and “un-parsing” (pretty-printing of the abstract structures, translation to other notations) can be performed or their code/specifications can be generated. E.g., in the generic “interactive programming environment” generator Centaur [2] – which may generate a parser, a structured editor, a type checker and an interpreter or a compiler for a formal language – the concrete and abstract grammars and building rules can be specified in the Metal formalism (which is then for example transformed into a format on which Yacc is called), while the semantic related specifications are in Typol which is then transformed and executed via Prolog, Lisp and, via the Minotaur system [3], attribute grammars. Centaur has been used for numerous programming languages and one KRL [4].

Tools such as Centaur ease the task of writing parsers, translators, etc. They can be seen as programming focused MDE tools. They could be extended to use ontologies or MOF and XML models. If such a tool allowed the re-use of an ontology of abstract and concrete models (such as the one we propose for KRLs), the various specifications that must be provided by people for each language would be much lighter, comparable and re-usable since the ontology permits to share and categorize them. This is much harder otherwise, even with KRLs such as Prolog which are oriented toward execution rather than modelling. Our KRL ontology includes sufficient information to allow people – programmers as well as application end-users – to specify the peculiarities of their notations for them to be usable as input or output KRLs: programers or users do not need to specify conversions (except for complex ones between abstract elements referring to elements of different logics) since they can be automatically derived.

To avoid the use of multiple structures or models, and thus also allow languages to be directly extended, other (and often earlier) avenues have been proposed. They all imply that extensions can be defined with the language and that it embeds a parser (e.g., via an “eval” function). This is eased when the language is *homo-iconic*, i.e., when its abstract structure can be directly derived from the text (because they have hom same structure). In other words, this is eased when new functions or rules can be built like other data structures and then evaluated, as in Lisp and Prolog. Lithe [5] is a class-based programming language looking like an EBNF grammar with semantic actions containing C-like code; the classes are the non-terminal alphabet of the grammar. Similarly, XBNF [6] is an extension of EBNF that is a KRL since it permits to define some functions, logical relations and sets on each class of objects defined by a rule. However, in all these other avenues, extensions to the language are restricted by its core concrete and abstract models. In other words, these extensions do not allow to represent and follow other models than the predefined ones. Yet, they show that using a unique language to represent abstract and concrete specifications has advantages (concision, …). A very expressive modelling-oriented KRL has those advantages without the restrictions since it can represent different models (because of it expressiveness, the *defined* models are just specializations of the core models). It also permits to categorize their elements.

2.2 Exploiting Structure-Dependent Meta-models

As explained in the introduction and the previous sub-section, using EBNF and other "languages to write concrete grammars" them implies programming-like tasks to build the abstract structures (ADT and semantic ones) from the concrete ones. The building of the ADT structure can be avoided when homo-iconic ADT meta-models are used, e.g., XML, MOF-HUTN (the Human Usable Textual Notation for MOF) or XMI (the XML-based notation for MOF models). However, the concrete descriptions are then not enough concise and high-level or user-friendly [7] to be used *directly* for developing or displaying (parts of) programs or KBs. Indeed, (i) these descriptions have a very explicit structure, and (ii) current meta-models – and hence their notations – *declare* and allow the use of few *structural* notions (e.g., the notion of object, attribute and – in MOF – association/relation), not logical notions (quantifier, meta-statement, etc.) nor programming notions (parallelism, succession, class inheritance, parameter evaluation, etc.). Although such "structural meta-models" permit to *declare* these additional notions in models, they do not permit to *define* their semantics. In other words, the tools which exploits structural meta-models (e.g., the XML parser and CSS pretty-printer) can only understand the meaning – and hence exploit – the structural elements. This is why, since 1999, for the Semantic Web and, more generally, for knowledge sharing and exploitation purposes, the W3C advocates the use of RDF instead of XML *as meta-model*. Indeed, XML is a tree-based structural meta-model while RDF is a graph-based meta-model which follows a simple logic and can be extended with language ontologies, e.g., those of OWL. However, like KRL models represented with a structural meta-model, extensions by language ontologies are just *declarations* of KRL components (not *definitions*): inference engines must handle them in special ways to take into account their semantics. The OMG followed this approach by proposing ODM (Ontology Definition Metamodel) which, in its version 1.1 [8], *declares* the elements of four KRL models in MOF (with a few relations between them) – RDF, OWL, Common Logic and Topic Maps – with a UML profile for the first two. No integrated ontology is provided. To conclude, since XML and MOF-HUTN cannot be used *directly* (as notations and ways to *define* semantics) for knowledge sharing or programming, tools supporting those tasks include parsers and semantic-handling modules in addition to – or replacement for – XML and MOF related tools.

Some MDE tools are described as having extensible *input notations*, e.g., BAM [9] in process modelling. Actually, they handle an expressive meta-model which includes the primitives for various (already existing) process modelling languages and thus can handle each of them. The authors of these tools also count textual annotations – both formal and informal ones – as a way to extend existing graphical notations.

2.3 Exploiting a Logic-Based Meta-model

As above noted RDF is an ADT meta-model following a simple logic. Its structures can be presented with concrete models specific to RDF (e.g., RDF/XML: RDF linearized with XML) or not (e.g., Turtle). These structures can be used for storing ADTs with more semantics (e.g., SPIN structures are RDF representations of the SPARQL query language). The generic parser and translator specified in this article also works for RDF. No generic parser just for RDF or another model seems to have been undertaken. On the other hand, there have been several works on style-sheet based transformation languages and ontologies for specifying how RDF abstract structures can be *presented*, e.g., in a certain order, in bold, in a pop-up window, etc.: Xenon [10], Fresnel [11], OWL-PL [12] and SPARQL Template [13]. Since these tools do not use a notation ontology, they require a new template or style-sheet for each target notation.

KRLs of low expressiveness ensure good properties for knowledge exploitation, typically speed and completeness. This is why KRL models of the OWL family have different degrees of expressiveness, all inferior to First Order Logic (FOL). For knowledge modeling and sharing purposes, the more expressive the used KRLs the better. Indeed, more expressiveness permit more definitions (instead of declarations) and thus permit to set more relations between different notions (logic ones, programming ones, …) from one or various sources. In other words, a more expressive KRL permits to represent knowledge in more precise (or less biased and ad-hoc) ways and in more generic, high-level, normalized and concise ways, hence in easier to develop and re-use ways. This is clear with the representation of cardinalities (or, more generally, numerical quantifiers; e.g., see the part in italic bold in Table 1), meta-statements and set interpretations. Thus, for knowledge modeling and sharing purposes – and also, as explained below, for knowledge exploitation purposes – a meta-model needs to represent "higher-order logic" (HOL). RIF-FLD [14], the W3C "Framework for Logic Dialects", is based on HOL. To ease its re-use, in our KRL ontology we represented the RIF-FLD elements and organized them via subtype and exclusion links (this organization was left implicit by the W3C, only a grammar and informal descriptions were provided). KIF (Knowledge Interchange Format) [15] is a KRL – with a FOL model but a HOL notation – which reached its purpose: being a de-facto KRL interlingua by allowing KRL authors to *define* elements of their KRLs in KIF and thus ease the translations of these KRLs into KIF.

A HOL model does not necessarily require a HOL inference engine to be handled. One reason is that, interpreted with Henkin semantics, it is equivalent to (many-sorted) FOL. This is how KIF has a HOL notation and a FOL model. Another reason is that conversions to less-expressive models (via losses of information) can be performed for applications, depending on their needs. E.g., a knowledge-retrieval application gain speed and does not loose much precision and completeness by ignoring meta-statements (modalities, …) as long as results are displayed with their associated meta-statements. Since HOL does not restrict possible exploitations – as opposed to knowledge modelled with KRLs of lower expressiveness – it is good for knowledge exploitation purposes too.

For business-to-business KS where the used structural/logic meta-models are well-known and sufficient for both businesses, KRLs of reduced expressiveness may be used and tailored knowledge conversion procedures may then be developed. For general knowledge sharing and re-use, or for making business-to-business KS more efficient, knowledge representation should not be restricted and, to allow the use of various KRLs, a generic parser-translator for KRLs is needed. This requires a shared ontology of KRLs (abstract models and notations) and, more generally, of general concepts. The Ontolingua server [16] was a first step in that direction. It proposed a structured library of interconnected fundamental ontologies, some of which formalized concepts related to KRL models, especially frame-based ones, i.e., concepts similar to those of OWL. It also hosted ontologies of its users in a structured library. However, this server did not have protocols to detect and help avoiding implicit redundancies between the ontologies and hence encouraging numerous relations between the various represented concepts. In other words, the various ontologies did not form an integrated one. The WebKB server [17] provides protocols solving this problem within a KB and between different servers, without restrictions on the content or on the used KRLs, nor forcing the users to agree on terminology or beliefs. The ontology proposed in this article is hosted by a WebKB server and thus can be extended by Web users.

This ontology integrates the main standards for KRL models: RDF + OWL + RIF-FLD from the W3C, Common Logic [18] (a subset of the KIF model) from *ISO/IEC* and the "Semantics of Business Vocabulary and Business Rules" (SBVR [19]) from the OMG. These standards have similar or complementary components which, previously, were not semantically related. Another originality is that our ontology includes a notation ontology.

3 Notation and Conventions Used in this Article for Illustrations

To allow the display and understanding of its numerous required illustrations, this article needs a concise and intuitive notation for KRL models of OWL-2 like expressiveness.

To that end, this article uses the FL notation [20] (it does not advocate the generalized use of FL since it proposes a way for people to use any notation they wish). Indeed, graphical notations are not concise enough and common notations such as those of the W3C are not sufficiently concise and "structured" enough. Here, "structured" means that all direct or indirect relations from an object can be (re-)presented into a unique tree-like statement so that the various inter-relations can readily be seen. Table 1 illustrates this by representing the same statement – or set of statements – in English and then in six formal notations: FE (Formalized-English [21]: it looks like English but it is actually very similar to FL), FL, UML, Turtle (or Notation 3), OWL Manchester notation and OWL Functional-style. This last notation is "positional relation" based. The first five are graph-based notations: they are composed of concept nodes and relation nodes.

The above textual graph-based notations are frame-based. A frame is a statement composed of a first "object" (alias "node": individual or type, quantified or not) and several links associated to it (links from/to other objects). In this article, "link" refers to an instance of a "binary relation type". In OWL, such a type is instance of "owl: Property" ("owl#Property" in FL: the namespace identifier is before the "#"). What is not an individual is a type: relation type or concept type (an instance of owl#Class).

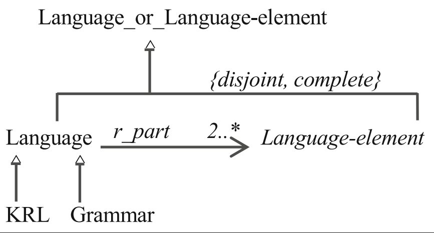
In this article, the default namespace is for the types we introduce via our ontology. Each name for a concept type or individual is a nominal expression beginning by an uppercase letter. The name of a relation type we introduce begins by "r_" (or *"rc_" if this is a type of link with destination a concrete term*). Thus, names not following these

Table 1. The same statement – or set of statements – in English and six different KRL notations: FE, FL, UML, Turtle, OWL Manchester, OWL Functional-Style. In all other tables, FL is used.

The type Language_or_Language-element is defined by its subtype partition composed of Language_element and Language. Any instance of Language has for (r_)part at least 2 instances of Language_element. This last type has for subtypes (at least) KRL and Grammar.

/* In **FE**: */ Language_or_Language-element has for partition {Language, Language_element}. Language has for r_part at least 2 Language_element.
Language has for subtype KRL and has for subtype Grammar.

```
Language_or_Language-element       //Notes: in FL, ">" is an abbreviation for the "subtype" link
 = exclusion                                //  as in some other notations).  "<" is its inverse.
   {  (Language                            //  "exclusion{...}" specifies a union of disjoint types :
        r_part:  2..* Language_element,  //  a real "subtype partition" of T if "T = exclusion{...}",
         >    KRL    Grammar  )            //   an "incomplete partition" if "T > exclusion{...}".
      Language_element       // A "," separates 2  links of different types. For consecutive links of
   };                        //  the same type, this type needs not  be repeated and there is no ",".
```



//In this **UML** representation, no box is drawn
// since no attribute or method needs to be
// represented.

// Here, an association/relation of type r_part
// is used instead of the special arrows used
// in UML for aggregations or compositions

:Language_or_Language-element //**Turtle + OWL** ***(which is a low-level KRL model, e.g., the***
owl:equivalentClass [rdf:type owl:Class; // ***part in italic bold below translates "2..*")***
owl:unionOf (:Language :*Language_element*)].
[ ] rdf:type owl:AllDisjointClasses; owl:members (:Language :*Language_element*).
Language ***rdfs:subClassOf [a owl:Restriction; owl:onProperty : r_part;***
owl:minQualifiedCardinality "2"^^xsd:nonNegativeInteger;
owl:onClass *Language_element* ***]´***
KRL rdfs:subClassOf :Language. Grammar rdfs:subClassOf :Language.
/ Here is a translation in "RDF/XML with OWL" of the 3rd line in this Turtle example:*
<owl:AllDisjointClasses><owl:members rdf:parseType="Collection">
<owl:Class rdf:about="Language"/>
*<owl:Class rdf:about="Language-element"/></owl:members></owl:AllDisjointClasses> */*

Class: Language_or_Language-element EquivalentTo: Language or *Language-element*
DisjointClasses: Language, *Language-element* //**OWL Manchester**
Class: Language EquivalentTo: r_part min 2 *Language_element*
Class: KRL SubClassOf: Language Class: Grammar SubClassOf: Language

---

EquivalentClasses(:Language_or_Language-element //**OWL Functional-Style**
ObjectUnionOf(:Language *Language-element*))
DisjointClasses(: Language :*Language_element*)
EquivalentClasses(:Language ObjectMinCardinality (2 :r_part :*Language_element*))
SubClassOf (:KRL : Language) SubClassOf (:Grammar : Language)

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 common convention for reading links in graph-based KRLs can be used, i.e., links of the form "X R: Y" can be read "X has for R Y". If "of" is used for reversing the direction of a link, the form "X R of: Y" can be read "X is the R of Y". The syntactic sugar of FE makes this reading convention explicit. Following this convention reduces the use of verbs and adjectives (which are more difficult to categorize and awkward to use with quantifiers) and thus normalizes knowledge.

In FL, if a link is not a subtype link nor another "link from a type", its first node is quantified and its default quantifier is "any". This one is the "forall" quantifier for definitions (in other words, the type in the first node is defined by this link). FL allows different links with the same first node to quantify this node differently. Indeed, in FL, the quantifiers of the source node and destination node of each link may also be specified in its relation node or in its destination node. This original feature permits FL to gather any number of statements into one visually connected graph. However, in this article, the quantifier for the first node is always "any" and left *implicit*. A destination node can also be source of links if they are delimited by parenthesis.

Given these explanations, the content of the tables in this article can now be read. Every keyword not introduced above will be explained via a comment near it. Comments use the C++ and Java syntax. In these tables, bold and italic characters are only for highlighting some important types and for readability purposes.

4 Situating Top-Level Language Elements in a Top-Level Ontology

Table 2 shows how types for KRL models and notations can be organized and inter-related. E.g., RIF-FLD includes RIF-BLD, both are part of the RIF family of models, and both have a Presentation Syntax ("PS") and an XML linearization.

Table 3 relates Language_element and some of its direct subtypes to important top-level types, thus adding precisions to these subtypes. Such a specification is missing in RIF-FLD but is well detailed in SBVR. This is why Table 3 includes many top-level SBVR types, although indirectly: *the types with names in italics* are still types that *we introduce* but they *have the same names as types in SBVR* and are equal to them or slight generalizations of them. This approach is for readability reasons and flexibility: if the SBVR authors disagree with our interpretation of their types, only some links to SBVR types will have to be changed, not our ontology. As illustrated by Table 3, to complement and organize types from other ontologies, ours includes many new types.

In RIF-FLD, depending on the context, the word "term" has different meanings. In our ontology, *Gterm* generalizes all these meanings of "term": it is identical to Language_element and sbvr#Expression. In RIF-FLD, an "individual term" is an abstract term that is not a Phrase (see Table 3), although it may refer to one. Individual_gTerm – or, simply "Iterm" – generalizes this notion to concrete terms too. This distinction was very useful to organize types of language elements, especially those from the

Table 2. Examples of relations between KRLs.

```
KRL   r_part: 1..* Language_element,    > exclusion { KRL_notation  KRL_model },
      r_grammar_head_element_type:  Grouped_phrases;

KRL_notation > (S-expression_based_notation  >  LISP_based_KIF)
               (Function-like_based_notation  >  (RIF_PS  >  RIF-FLD_PS   RIF-BLD_PS) )
               (Graph-based_notation > (Markup_language_based_notation
                                           > (XML_based_notation
                                                > (RIF_XML  > RIF-FLD_XML) )
                                        (Frame_based_notation  >  FL   JSON-LD  Turtle) )  );

KRL_model   //KRL abstract model (for an ADT and/or a logic)
 > (First-order-logic_with_sets_and_meta-statements
      > (KIF_model   r_model_type of:  LISP-based_KIF),   r_part: 1..* First-order-logic )
   (First-order-logic > (CL  r_model_type of: CLIF) )
   (RIF > (RIF-FLD  r_model_type of: RIF-FLD_PS,    r_part: RIF-BLD )
          (RIF-BLD  r_model_type of: RIF-FLD_PS) )
   (Graph-based_model
      > (JSON-LD_model  r_model_type of: JSON-LD)
        (RDF   r_part: 1..* JSON-LD_model,     r_model_type of: JSON-LD  RDF/XML),
        (Frame_model_with_closed_world_assumption  > F-Logic_classic_model )
        (Frame_model_with_open_world_assumption
           > (Description_logic_model
                > (OWL_model
                     > (OWL-1_model  >  OWL-Lite     OWL-DL      OWL-1-Full)
                       (OWL-2_model  >  OWL-2_EL   OWL-2-RL   OWL-2-Full),
                     r_part: 1..* (OWL-1-Full   r_part:  1..* RDFS    1..* RDF    1..* OWL-DL )
                             1..* (OWL-2-Full   r_part:  1..* OWL-2_EL     1..* OWL-2_RL ) ) ) ) );
```

implicit ontology of RIF-FLD (this framework uses different vocabulary lists, including one for signatures; in our ontology, all these terms are inter-related). Used in this context, the informal word "individual" is not equivalent to "something that is not a type". Indeed, since an Iterm may refer to a Phrase, an Iterm identifier may be a Phrase identifier. Thus, Table 3 uses the construct "near_exclusion" instead of "exclusion". This construct has no formal meaning (it does not set exclusion links). It is only useful for readability purposes. Table 3 also uses it to group and distinguish types for abstract and concrete terms. Indeed, a (character) string may be seen by some persons as being both abstract and concrete. Our ontology is – and must be – compatible with many visions, when this can be achieved without information loss.

5 Ontology of a Core Meta-model for KRL Languages

RIF-FLD distinguishes *three* types of generic structures for a Gterm that is a function or a phrase. We dropped their RIF-related restrictions and named them Positional_gTerm, Gterm_with_named_arguments and Frame. Table 1 gave examples for positional and frame terms. A term with named arguments is similar to a frame except that,

Table 3. Situating Language_element w.r.t. other types (note: names in italics come from SBVR).

```
Thing  = owl#Thing,   r_identifier: 0..* Individual_gTerm,
 = exclusion
   { (Situation  =  exclusion {State  Process},    r_description: 0..* Phrase )
     (Entity   //thing that can be involved in a situation
       > exclusion
         { Spatial_entity   //e.g., Square, Physical_Entity
           (Non-spatial_entity   //e.g., Justice, Attribute, ...
             > (Description_content  = Meaning,
                  > Proposition   Question
                    (Concept >  Noun_concept /*e.g., types */   (Verb_concept = Fact_type) ) )
               (Description_container  >  (File  >  RDF_file) )
               (Description_instrument
                  > (Language_or_Language-element
                       = exclusion { (Language  >  KRL   Grammar,
                                                   r_part: 1..* Language_element )
                                     Language_element   //see below
                                   } ) ) )
         }  )  };

Language_element  =  Gterm   Expression,    r_representation of: 1 Meaning,
 > near_exclusion     //String is both abstract and concrete
   { (Representation > Statement,   rc_type: Concrete_term )
     (Concrete_term > (Expression > Text)  (Concrete_iTerm < Iterm) )  //see tables 10 and 11
   }                                                                   // (for some subtypes)
   near_exclusion  //a reference to a phrase is an Iterm
   { (Phrase > Statement  Definition  Frame)  //+ see Table 6
     (Individual_gTerm  = Iterm,  > Place_holder,   r_identifier  of: 1 Thing)  //see tables 10-11
   }
   near_exclusion { Positional_gTerm   Frame   Gterm_with_named_arguments  }
   near_exclusion  //subtyping these types is KRL dependent
   { Non-referable_gTerm  //e.g., a predefined term
     (Referable_gTerm  //via constant/variable/function/phrase      //referable→linkable
        r_variable: 0..* Variable,     r_result of: 1..* Function,   r_annotation: 0..* Annotation,
        >  (Gterm_that_cannot_be_annotated_without_link    r_annotation: 0 Annotation )
           Termula )  };
```

as in object-oriented languages, local attribute names are used instead of link types (types are global). It could be argued that a same term could be presented in any of these three forms and hence that these three distinctions should rather be syntactic. However, the authors of RIF-FLD have not formalized the equivalence or correspondence between (i) "classes and properties" (or frames "interpreted as sets and binary relations") and (ii) "unary and binary predicates", *in order to have* a "uniform syntax for the RIF component of both RIF-OWL 2 DL and RIF-RDF/OWL 2 Full combinations" [22]. According to this viewpoint, each person re-using ontologies must decide if, for its applications, stating such an equivalence is interesting or not. RIF rules

Table 4. Main links for defining a structure for abstract terms and specifying concrete terms.

```
Gterm   r_identifier_or_description of: 1 Thing,   //these link types are used in tables 6 to 10
   r_operator:  0..1 Operator ,   //see Table 10
   r_part:  0..* Gterm,  /* object parts or fct/relation arguments */      r_parts: 1 List,  //ordered
   r_result: 1 Gterm,  /* e.g., a phrase has for r_result a boolean */    rc_type:  Concrete_term,
   r_variable: 0..* Variable,     r_result of: 1...* Function;

rc_link_to_concrete-term   //and from an abstract term but also often from a concrete term
     _[/*from:*/Gterm, /*to*/Concrete_term]  //signature of this relation type
 >  (rc_begin-mark >  rc_operator_begin  rc_parts_begin)    (rc_separator  > rc_parts_separator)
     (rc_end-mark  > rc_operator_end  rc_parts_end )      rc_operator_name
     rc_infix-operator_position      rc_annotation_position; //-1: before

r_gTerm_part   r_type:  Transitive_relation_type,
  > (r_operator >  r_frame_source  rdf#predicate)   (r_phrase_part > rdf#subject  rdf#object )
     (r_parameter = r_part,  //"r_part" used for concision
        > (r_link_parameter   >  (r_link_source        >  rdfs#domain)
                                   (r_link_destination > rdfs#range) ) );

//r_parts permits to order the parts, this is sometimes needed for abstract terms and
// this also permits to give a default order for presentation purposes.
r_parts _[?e,?list] :=>  [any  ^(Thing  r_member of: ?list)  r_part of: ?e];
r_parts _[?e,?list] :<=  [any  ^(Thing  r_part of: ?e)  r_member of: ?list];

/* Notes:  in FL,  ":=' permits to give a full definition,
 ":=>"  gives only "necessary conditions",   ":<=" gives only "sufficient conditions",
 "^(" and ")" delimit a  lambda-abstraction (a construct defining and returning a type),
"_(" and ")" delimit the parameters of a function call,  "_[" and "]" delimit those of a definition,
 ".[" and "]" delimit  the elements of a list,   ".{" and "}" delimit the elements of a set.   */

rc_type _[?t,?rct] := [any ?t  rc_: 1..* ?rct]; //people who see concrete terms as specializations
                // of abstract terms can still state:    rc_type <  subtype;       rc_  r_type:  instance;
                 //in the next function signature (i.e., in [...]), the variables are untyped

f_link_type  _[ ?operatorName,  ?linkType,  ?linkSourceType,  ?linkDestinationType ]
 :=  ^(Link   rc_operator_name: ?operatorName,    r_parts: .[ ?linkSource ?linkDestination],
                r_operator: ?linkType,  r_result: 1 Truth_value,
                r_link_source: 1 ?linkSourceType,   r_link_destination: 1 ?linkDestination );
```

or a macro language such as OPPL can certainly be used for such structural translations [23]. However, to avoid imposing this exercise to most users of our KRL ontology, and to avoid limiting its use for specifying KRLs, it formalizes relations between a frame and a Conjunction_of_links_from_a_same_source (this is done in the last 17 lines of Table 9; reminder: a link is – or can also be seen as – a binary relation).

We found that a small number of link types are sufficient for defining a structure for abstract terms and specifying their related concrete terms. Table 4 lists and explains

Table 5. Important structured concrete term types and definition of their default presentation.

```
Structured_concrete_term_that_is_not_a_string        //the delimiters in comments below permit a
 > exclusion             // KRL to have all these structures  and still only requires an LALR(1) parser
   { (List_cTerm >  Enclosed_list_cTerm  /* e.g.,  .[A B C]  */
                    Fct-like_list_cTerm    /* e.g.,  A ..[B C]  */  )
     (Set_cTerm  >  Enclosed_set_cTerm  /* e.g.,  .{A B C}  */
                    Fct-like_set_cTerm     /* e.g.,  A ..{B C} */  )
     (Positional_cTerm  //e.g., with operator "f" and parts/parameters A, B and C
        rc_operator-name: "",   rc_operator_begin: "",   rc_operator_end: "",   //declared in Table 4
        rc_parts_begin: "(",     rc_parts_separator: "",   rc_parts_end: ")",
        rc_infix-operator_position: 0,  //if not 0, it indicates the operator position within the parts
        > exclusion
          { (Fct-like-cTerm
               = exclusion { (Prefix_fct-like-cTerm  rc_parts_begin: "_(")        //e.g.,   f _(A B C)
                             (Postfix_fct-like-cTerm  rc_parts_begin: "(_") } )     //e.g.,   (_ A B C)f
            (List-like_fct_cTerm
               = exclusion { (List-like_prefix-fct_cTerm   rc_parts_begin: ".(")    //e.g.,   .(f A B C)
                             (List-like_infix-fct_cTerm   rc_parts_begin: "(.",
                                                          rc_operator_begin: ".")  //e.g.,   (. A B .f  C)
                             (List-like_postfix-fct_cTerm rc_parts_begin: "(..")   //e.g.,   (.. A B C f)
                           } ) } )
     (Frame_cTerm  //e.g., with operator type "f" and with parts two half-links of type r1 and r2
        rc_operator-name: "",   rc_operator_begin: "",  rc_operator_end: "",
        rc_parts_begin: "{",  rc_parts_separator: ",",  rc_parts_end: "}",   //as in JSON-LD
        rc_parts: 1..* Half-link_cTerm,
        > exclusion { (Prefix_frame_cTerm  rc_parts_begin: "_{")  //e.g.,  f_{ r1: A,   r2: B }
                      (List-like_frame_cTerm   rc_parts_begin: "{.",
                         > List-like_prefix-frame_cTerm                   //e.g.,  {. f   r1: A,   r2: B}
                           List-like_infix-frame_cTerm )                  //e.g.,  {. r_id: f,  r1: A,   r2: B}
                      (Postfix_frame_cTerm rc_parts_begin: "{_") //e.g.,   {_  r1: A,  r2: B } f
                      Alternating-XML_cTerm   //Frame in the Alternating-XML style where
                    } )                // concept nodes alternate with link nodes, as in RDF/XML
     Cterm_with_named_arguments  //quite rare in KRLs, hence not detailed in this article
   };

fc_prefix-fct-like_type _[?notationSet, ?operator_name, ?begin_mark, ?separator, ?end_mark]
 :=  ^(Prefix_fct-like-cTerm   r_direct-or-indirect_part of: ?notationSet,  //uses of this function are
                               rc_operator-name: ?operator_name,          // illustrated in Table 13
        rc_parts_begin: ?begin_mark,  rc_parts_separator: ?separator,  rc_parts_end: ?end_mark )

Phrase  //any phrase has at least these presentations in these 2 kinds of notations (see Table 2),
        // e.g., in RIF-PS and RIF-XML:
  rc_type:  ^(fc_prefix-fct-like_type _(.{Function-like_based_notation},"","(","",")")
                 rc_annotation-position: -1 )
            ^(fc_alternating-XML_type_(.{XML_based_notation},"") rc_annotation-position: 0 );

List   rc_type:  fc_list_type _( .{Notation}, "[", "," ,"]" );  //by default, in any notation, a list has
            // for representation a comma separated list of element delimited by square brackets;
            // note that fc_list_type has no argument for an operator name
```

the main link types. They can be seen as a representation and *extension* of the signature system of RIF-FLD. The *ideas* are that (1) every composite term can be decomposed into a (possibly implicit) *operator* (e.g., a predicate, a quantifier, a connective, a collection type) and a *list of parameters* (alias, "parts"), and (2) *many non-binary relations can be specified as links to a collection of terms*. Table 6 and the subsequent tables use the link types of Tables 4 and 5 directly or via functions which are shortcuts for specifying such links. This is highlighted via bold characters in those tables. The end of Table 4 specifies one of these functions (f_link_type). In the Tables 6, 7, 8, 9, 10 and 11, which illustrate the organization of subtypes of Phrase and Iterm, f_link_type is used to define some abstract terms *as links* and hence *enables to store them or present them as such when necessary*. To illustrate the way these ADTs are instantiated by knowledge representations, Table 12 shows a *phrase in different notations and then a part of its abstract structure*.

Some links are used for both abstract and concrete terms. E.g., rc_operator_name is often also associated to an abstract term for specifying a default name for its operator. If no such link is specified or if the empty string ("") is given as destination, the operator type name (without its namespace identifier) is used as default operator name.

Table 5 lists major kinds of structured concrete terms and thus also the main presentation possibilities for structured abstract terms (see the 14 names in italics). Based on the five main categories for these concrete terms (see the names in bold and not in italics), it is easy to find the five categories of abstract terms they correspond to, even though such links are not shown in Table 5. We found that each of these concrete term types can be defined with only a few types of links, those that begin by "rc_" and that were listed in Table 4. We defined some functions to provide shortcuts for setting those links when defining a particular concrete term, e.g., fc_prefix-fct-like_type (*its use is illustrated by Table 13 for the definition of RIF-PS and JSON-LD*).

In our ontologies, links from a type do not specify that the given destination is the only one possible (to do so in FL, "=>" would need to be used instead of ":" after the link type name; in OWL-based descriptions, owl#allValuesFrom can be used). Thus, such links represent "default" relationships: if a link from a type T specializes a link from a supertype of T, it overrides this inherited link. This is also true when the link type is functional (i.e., when it can have only one destination) and its destination for T does not specialize the destination for a supertype of T. The links beginning by "rc_" look functional but are not: in FL, multiple destinations can be stated to indicate different presentation possibilities. However, by convention, such links override inherited links of the same types. Table 5 shows how different kinds of "default presentations" can be represented in concise ways.

6 Ontology of KRL Content Models

Table 6. Important top-level types of phrases.

```
//Note: names in italics come from RIF-FLD, names in bold italics are used in RIF-FLD signatures,
// bold is for highlighting, "cl#" prefixes terms from Common Logics

Phrase <  ^(Gterm r_operator: 1 (Operator_type > (owl#Property  r_instance: r_binary_relation) ),
                    r_result: 1 (Truth_value  r_instance:  True  False  Indeterminate_truth-value) ),
  > (Phrase_not_referable_in_RIF-FLD   //→ cannot have an annotation in RIF-FLD
      > (Annotation  >  cl#Comment   (Formal_annotation > RIF_annotation) )   Module_directive
        (Annotating_phrase =  f_link_type_("",r_annotation,Gterm,Annotation))  Attribute  ),
 = exclusion
   { (Modularizing_phrase       // Phrases is the head element of a KRL grammar
       > (Phrases =  Grouped_phrases,   r_part: 0..* Phrase,   > cl#Text,
          > (Module  > (Document  r_part:  0..1 Document_Directive   0..1 Phrases)  cl#Module,
              r_part: 0..1 (Module_parts_that_are_directives  <  Module,
                          >  (Module_header = f_link_type_("",r_header,Module,
                                                          .[0..* Module_directive] ) ) )
                    0..1 (Module_parts_that_are_not_directives =
                                   f_link_type_("Group",r_group,Module,.[0..* Phrases],
                          <  Module,    >  Module_body   Group_of_phrases ),
              r_parts: .[0..1 Module_header, 0..1 Module_body]  ) )
        (Module_directive  = f_link_type_("",r_relation,Module,Thing),
          > (Module_name_directive = f_link_type_("Name",r_name,Module,Name) )
            (Excluded_Gterm-reference_directive = f_link_type_("",r_excluded_gTerm,Module,
                                                             .[1..* Gterm_reference]) )
            (Document_directive
              > (Dialect_directive =  f_link_type_("Dialect",r_dialect,Module,Name))
                (Base_directive    =  f_link_type_("Base",r_base,Module,Document_locator))
                (Prefix_directive   =  f_link_type_("Prefix",r_prefix,Module,
                                              NamespaceShortcut-DocumentLocator_pair))
                (Import-or-module_directive  > cl#Importation,
                  > (Import_directive = f_link_type_("Import",r_imported-doc,Document,
                                                 Imported_document_reference) )
                    (Remote_module = f_link_type_("Module",r_imported-module,Module,
                                                 Remote_module_reference) )
                ) ) ) )
     (Non-modularizing_phrase   //including non-monotonic phrases: queries, removals, ...
       > (Formula  >  Positional_formula   Formula_with_named_arguments
                         Phrase_of_a_grammar   cl#Sentence,
          = exclusion  //the 3 following distinctions come from KIF
            { (Definition = exclusion { Non_conservative_definition  Conservative_definition } )
              (Sentence  //fact in a world: formula assigned a truth-value in an interpretation
                 >  Belief  //the fact that someone believes in a certain thing
                    Axiom ) //sentence assumed to be true, from/by which others are derived
              (Inference_rule> Production_rule) //like an implication but the conclusion is "true"
            }                                     // only if/when the rule is fired
            near_exclusion{Composite_formula  Atomic_formula_or_reference_to_formula} ) )
        Termula_phrase /*RIF function/atomic_formula parameter; not detailed in this article*/  )
   };
```

Table 7. A way to restrict this general model for specifying particular KRLs.

```
r_only_such_part_of_that_type _[?x ?pt]  //The source ?x has some parts of type ?pt but no other
  <  r_part _(?x ?pt),   // parts with type the genus of ?pt. The definition in the next line requires that
  := [?x  r_part:  1..* ?pt    0 ^(?t != ?pt,  < (?gpt  r_genus_supertype of: ?pt))];     // relations of type
 // r_genus_supertype are set by definitions. Thanks to this link type, to our general model for KRLs
// and to the default presentation associated to its abstract terms, KRLs can be defined in a very concise
// way.  Below are examples for some abstract terms of some KRLs. The next section gives examples
// for some concrete terms of some KRLs. For the Triplet_notation, nothing else is required.

RIF   r_only_such_part_of_that_type:   //RIF models have for part terms defined by these 6 lambdas:
  ^(Gterm_that_can_be_annotated_without_link > Phrase)   //-> only a phrase can be annotated
  ^(Grouped_phrases  r_part: 0..* Document)
  ^(Quantification >  Classic_quantification)    ^(Frame > Minimal_frame)    ^(Collection > List)
  ^(Delimited_string >  Delimited_Unicode_string);

RIF-BLD   r_only_such_part_of_that_type :      //the next two lambdas se are just  examples,
  ^(Rule_conclusion > rif-bld#Formula)             // RIF-BLD has other restrictions
  ^(Rule_premise  >  Connective_phrase_on_atomic_formulas   Conjunction_phrase);

Triplet_notation = ^(KRL  r_only_such_part_of_that_type:
                                    ^(Phrase > Link)    ^(Individual_gTerm > Constant_or_variable) );
```

Table 8. Important types of composite formulas.

```
//Names in italics come from RIF-FLD, names in bold italics are used in RIF-FLD signatures

Composite_formula = f_relation_type_("",r_relation,.[1..* Formula]),  //-> r_part: 1..* Formula
  > exclusion
    { (Formula_connective   r_operator_type:  1 connective_operator,   >  cl#Boolean_sentence,
        > exclusion  //e.g., Negative_formula, Production_rule, Logical_implication, ...
          { (Unary_connective_phrase  = f_relation_type_("",r_unary_relation,.[1..* Formula]) )
            (Binary_connective_phrase  = f_relation_type_("",r_binary_relation,.[1..* Formula]) )
            (Variable-n-ary_connective_phrase
              = f_relation_type_("",r_variable-ary_relation,.[1..* Formula]),
              >  exclusion { (Disjunction_phrase = f_relation_type_("Or",r_or,.[1..* Formula]))
                             (Conjunction_phrase = f_relation_type_("And",r_and,.[1..* Formula]),
                               > (Conjunction_of_links = f_relation_type_("And",r_and,Link),
                                   > Frame_as_conjunction_of_links_from_a_same_source ) )   } )
          } )
      (Quantification = f_quantification_type_("",Quantifier,.[1 Type],Constant-or-variable,Formula),
        > (Classic_quantification = f_quantification_type_("",Quantifier,.[],Variable,Formula) )
          exclusion                                        //classic: no guard, no constant
          { (Universal_quantification
               = f_quantification_type_("Forall",q_forall,.[1 Type],Constant_or_variable,Formula),
               > (Classic_universal_quantification
                    = f_quantification_type_("Forall",q_forall,.[],Variable,Formula) ) )
            Existential_quantification //defined the same way as for Universal_quantification
          } ) };
```

Table 9. Important types of relations between frames, links and positional formulas.

```
Atomic_formula_or_reference_to_formula  > exclusion
 { (Formula_reference  /* this is also an Individual_gTerm  */
     > exclusion { Variable_for_a_formula
                   Reference_to_formula_in_remote_module  //with the same KRL
                   Reference_to_externally_defined_formula
                 } )              //external: not in a module and not with the same KRL
   (Atomic_formula   //names in bold italics are used in RIF-FLD signatures
     > { Constant_for_a_formula
         (Atomic_formula_that_is_not_a_constant
           > near_exclusion  //possible shared subtypes: subclass_or_equal, link
             { (Positional-or-name-based_formula
                  r_operator: 1 Termula,  >  cl#Atomic_sentence,
                  > exclusion { (Positional_formula  r_part: 1..* Termula)
                                (Name-based_formula  r_part: 1..* Name-Termula_pair) } )
               (Equality_formula = f_link_type_("=",r_equal,Termula,Termula),  >  cl#Equation)
               (Class-membership = f_link_type_("#",r_type,Termula,Termula) )
               (Subclass_formula = f_link_type_("##",r_supertype,Termula,Termula) )
               (Frame= (Frame_as_conjunction_of_links_from_a_same_source  ?f
                          r_frame_head: 1 Termula ?fh,   r_part: (1..* Link  r_link_source: ?fh) )
                       (Frame_as_head_and_half-links_from_head  ?f
                          r_operator: (1 Termula ?fh  r_frame_head of: ?f),
                          r_part: (1..* Half_link  r_link_source: ?fh),
                          > (Minimal_frame  r_part: 1..* Minimal_half-link) ) )
             } )
             (Binary_atomic_formula_that_is_not_a_constant
               > (Link
                   =  (Link_as_positional_formula  < Positional_formula,
                         <  f_link_type_("",r_binary_relation_type,Termula,Termula),
                         r_part of: (1 Frame ?f  r_frame_head: 1 Termula ?fh),  r_link_source: ?fh )
                      (Link_as_frame_part   r_part of: (1 Frame ?f  r_frame_head: 1 Termula ?fh),
                         r_operator: ?fh,   r_link_source: ?fh,   r_link_destination: 1 Termula ?ld,
                         r_part: (1 Half_link   r_link_source: ?fh,   r_operator: ?fh,  r_parts: ?ld )
                  ) ) )
       } ) };
```

Table 10. Important top-level types of “individual terms” (not phrases unless referring to one).

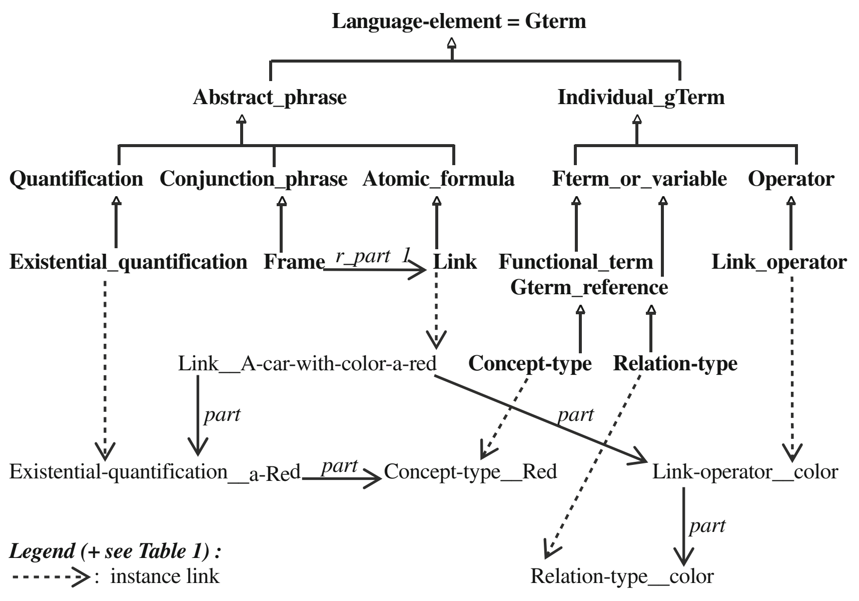
```
Individual_gTerm = near_exclusion          //names in italics come from RIF-FLD
 { (Individual_concrete_term
     > Concrete-term_for_constant_or_name   Lexical-grammar_character-set   Character
       Concrete_list-like_term     (String >  (Delimited_string > Delimited_Unicode_string) ))
   (Individual_abstract_term
     > Abstract_individual_gTerm_not_referable_in_RIF-FLD   //Quantifier, Half_link, ...
       (Fterm_or_variable  >   (Functional_term   r_operator: 1 (Function_type <  Type) ) )
       Individual_abstract_term_of_a_grammar
       (Operator   >  r_relation  f_function   Operator_not_referable_in_RIF-FLD )
       (Symbol_space > rif#iri  rif#local  xs#string  xs#integer  xs#decimal  xs#double)  )   };
```

Table 11. Important subtypes of one subtype of Individual_gTerm (see Table 10).

```
Concrete-term_for_constant_or_name              //examples to show that the same approach
 > (Variable_name   r_identifier of: 1..* Variable,   //  also applies for concrete terms
      < ^(f_string_type_("?","","")  r_part: 1 Undelimited_variable-name)  )
    (Constant_concrete_term  r_identifier: 1..* Constant_gTerm,
      > (Constant_concrete_term_without_symbol-space
           > (Constant_IRI  r_part: 1 IRI_reference )
             (Constant_short-name_via_compact_URI  r_part: 1 Compact_URI)
             (Literal_or_datatype_concrete_term  r_identifier of: 1..* Literal_or_datatype,
               > (Double_quoted_string
                    < ^(f_string_type_("","",""")
                          r_part:  1..* f_character_type_with_escape_for_(Character,"\\","") ))
                 (Numeric_literal
                    > (Positive_integer  <  ^(f_string_type ("+","","") r_part: 1..* Digit))
                      (Negative_integer <   ^(f_string_type_("-","","") r_part: 1..* Digit)) ) ) ) );
```

Table 12. One phrase in different notations; part of its abstract structure represented in UML.

In English: "There exists a **car** which is **red** (one shade of red; it may also have other colors)".
In FE: `a **Car** with *color* a **Red**´.   In RIF-PS: Exists ?car ?red (*color*(?car#**Car** ?red#**Red**))
In FL: a **Car** *color :* a **Red** ;   In RIF-PS: Exists ?car ?red ?car#**Car** [color -> ?red#**Red**].
In N-Triples: **Car8** *color* **Red3** .   **Car8** *type* **Car** .   **Red3** *type* **Red** .



7 Ontology of Particular KRLs and Grammars

In a KRL that is "perfectly regular *with respect to a particular kind of abstract/concrete term*" allows all the terms of this kind to be (re-)presented in the same way, e.g., all the terms which in our approach have an operator (the "operator based terms"). A *perfectly regular KRL* is then one which is perfectly regular for all the kinds of terms it allows. The "Triplet notation" is perfectly regular. To be so, a more expressive KRL would have to be fully based on an ontology and be HOL based. Since KIF re-uses the LISP notation, it is perfectly regular with respect to "operator based concrete terms" and "concrete terms for collections". Most KRLs have some *ad hoc* abstract and concrete terms. E.g., in RIF-XML, document directives are presented in different ways: some via links, some via XML attributes. In RIF-PS, they are presented as positional terms but not links. Thanks to the fact that our general model represents the directives both as parts and links (see Table 6), these RIF predefined directives can be represented within/via frames as well as via positional terms. The first part of Table 13 shows how *ad hoc* concrete terms of particular types of KRLs can be specified in a concise way. The approach used to do so for abstract terms (see Table 7) is here re-used. Thus, both abstract and concrete terms of a KRL – or a family of KRLs – can be specified at the same time and in a concise way. Furthermore, since (families of) KRLs and their specifications can be organized by specialization relations, they can be formally and visually compared.

The second part of Table 13 shows how it is possible to declaratively specify all the presentations of a type of abstract term by an ordered list of concrete terms, given a type of presentation and the list of usable notation types. Since the function fc_r_parts is recursive and, in turn, uses the same kinds of specifications (via links of type rc_parts or, for non-structured terms, links of type rc_), the specified ordered list only contains strings. Finally, given the value of rc_separator between tokens in the given notation (i.e., the kinds of space characters separating them), the kinds of strings that can be associated to this collected list are specified. Thus, the whole specification is fully declarative. However, for concrete term generation purposes, choices have to be made, e.g., about space indentation. In our system, this is implemented via generation functions (also included in our ontologies) which recursively navigate the abstract and concrete specifications to find the most precise relevant specifications. Since our system rejects the entering of ambiguous knowledge – e.g., the entering of different concrete term specifications for a same type of abstract term and the same type of notation – finding the most precise relevant specifications was easy to implement.

Specifying parsing rules and generating them – given an abstract term and a grammar notation – can be represented using the same techniques. The first part of Table 14 shows the beginning of an ontology for grammars. The second part shows an example of grammar rule (and its connection to a grammar but this part actually needs not be generated). Once the grammar rules are *generated* – in a way *similar to presentation generation* – the generation of *their presentation is then done exactly as for any other statement*, according to the given grammar notation.

Our ontologies can be represented with KRLs having at least OWL-2 expressiveness. To that end, r_parts links with "lists with cardinalities" (e.g., [0..1 Y, 1..* Z]) as

Table 13. Ways to specify concrete terms for particular kinds of terms in particular notations.

```
//Thanks to the default values in our specifications for abstract and concrete terms, only the
// following lines are needed for  defining the presentation in RIF-PS of the abstract terms shared
// by the  KRLs of the RIF family. For instance, the order and operator names of the directives of
// a document can be found in Table 6. Since these directives follow the default presentation for
// phrases in RIF-PS, nothing needs to be specified about them here. The abstract term restrictions
// can be specified here (as  illustrated below for "Frame" or separately, as illustrated by the
// second part of Table 6.

RIF    r_only_such_part_of_that_type:   //default values make the next lines sufficient
 ^(Phrase  rc_type: fc_prefix-fct-like_type_(.{RIF-PS},"","(","",")") )  //default style in RIF_PS
 ^(RIF_annotation  rc_type: fc_list_type_(.{RIF-PS},"(*","","*)"))   //override for  annotations
 ^(Quantification_bound_list  rc_type: fc_list_type_(.{RIF-PS},"","",""))
 ^(Rule rc_type: fc-like_infix-fct_type_(.{RIF-PS},":-","","",""))
 ^(Externally_defined_term  rc_type: fc_prefix-fct-like_type_(.{RIF-PS},"External","(","",")"))
 ^(Equality_formula  rc_type: fc_list-like_infix-fct_type_(.{RIF-PS},"=","","",""))
 ^(Subclass_formula rc_type: fc_list-like_infix-fct_type_(.{RIF-PS},"##","","",""))
                              //e.g., "?t1 ## ?t2";  in FL: "?t1 < ?t2"
 ^(Class-membership_formula  rc_type: fc_list-like_infix-fct_type_(.{RIF-PS},"#","","",""))
 ^(Frame > Minimal_frame,   rc_type: fc_infix_list-like_frame_type_(.{RIF-PS},"","[","","]") )
 ^(Half_link rc_type: fc_half-link_type_(.{RIF-PS},"","","->","",""))
 ^(Name-Termula_pair  rc_type: fc_list_type_(.{RIF-PS},"","->",""))
 ^(Open_list    rc_type: fc_prefix-fct-like_type_(.{RIF-PS},"List","(","|",")"))
 ^(Open-list_rest rc_type: fc_list_type_(.{RIF-PS},"","","",""))
 ^(Aggregate_function rc_type: fc_prefix-fct-like_type_(.{RIF-PS},"","{","|","}"))
 ^(Aggregate_function_bound_list rc_type: fc_fct-like_list_cTerm_(.{RIF-PS},"[","",]""));

RIF-FLD    r_only_such_part_of_that_type:    //only 1 example: a document in RIF-FLD/XML
  ^(Document  rc_type:  (1 fc_alternating-XML-cTerm_type_(.{RIF-XML},"Document")
                                   rc_annotation-position: 0,
                                   rc_XML-attribute_type:  r_dialect  xml#base  xml#prefix,
                                   rc_XML-link_types: .[rif#directive rif#payload] ) );

JSON-LD_model   r_only_such_part_of_that_type:   //specifies also the JSON-LD notation
 ^(Phrase  rc_type:  fc_list-like_infix-frame_type_(.{JSON-LD},"","{",",","}"))
 ^(Half_link rc_type: fc_half-link_type_(.{JSON-LD},"",":","",""))
 ^(Module_header  rc_type: fc_list-like_infix-frame_type_(.{JSON-LD},"'@context:'","{",",","}"))
 ^(Module_body   rc_type: fc_list_type_(.{JSON-LD},"","",",",""))
 ^(Formula  >  ^(Minimal_frame  r_operator: 1 Constant_gTerm))   //only 1 destination per link
 ^(Fterm_or_variable >  Constant_or_set_or_closed_list)
 ^(Set  rc_type: fc_list_type_(.{JSON-LD},"[",",","]"))  //in classic JSON, this would be for a list
 ^(Closed_list  > ^(Frame  r_part: 1 .[r_container, Closed_list],
                           rc_type: fc_half-link_type_(.{JSON-LD},"","@container",":","@list","") )
                     ^(Frame  r_part: .[r_list, 1 Set],  //2nd way to represent a closed list in JSON-LD
                                rc_type: fc_half-link_type_(.{JSON-LD},"","@list",":","","") ) );
```

```
//Another kind of specification ("^?" prefixes variables that are implicitly universally quantified):
^(Thing ?t    rc_:  (a Enclosed_list_cTerm ?c   r_KRL-set:  ^?notationSet))
  rc_parts:  f_remove_empty_elements_in_list_( .[ (^?cb  rc_begin_mark  of: ?c),
                      fc_r_parts_(?notationSet, (^?tp r_parts of: ?t), (^?cs rc_parts_separator of: ?c))
                      (^?cb  rc_end_mark of: ?c) ]  );
```

Table 14. Important links from Grammar_element, followed by an example of grammar head rule.

```
Grammar_element   //specifications mainly only for  EBNF-like and Lex&Yacc-like grammars
  r_part  of: 1..* Grammar,      //and conversely:    Grammar   r_part:   1..* Grammar_element;
  > exclusion
     { (Phrase_of_a_grammar   >  Head_grammar-rule,
           = exclusion{Non-lexical-grammar_rule   Lexical-grammar_rule} )
       (Individual_gTerm_of_a_grammar
           = exclusion{  Lexical-grammar_individual-gTerm  //what Lex grammars handle
                         Non-lexical-grammar_individual-gTerm } ) };

Non-lexical-grammar_rule  =  NLG_rule,        //this is a beginning but the representation
  r_part:   1 NLG_rule_left-hand-side    1 NLG_expression           // of the whole grammar
            0..1 (Parsing_action_phrase < Phrase),                   // is similar
  rc_:  (1 fc_list_type_(.{W3C-EBNF,XBNF,Grammar},"","",""")
            //fc_list_type is like fc_prefix-fct-like-cTerm_type but without operator as parameter
           rc_parts: .[NLG_rule_left-hand-side "::=" NLG_expression] )
                                        //→"A::=B"  ("Grammar "→default presentation)
        (1 fc_list_type_(.{ISO-EBNF},"","","")
           rc_parts:  .[NLG_rule_left-hand-side "=" NLG_expression] )
                                        //→ "A = B" in ISO-EBNF
        (1 fc_list_type_(.{Yacc, Bison},"","","")
           rc_parts:  .[NLG_rule_left-hand-side ":" NLG_expression] );
                                        //→ "A : B" in Yacc or Bison (without parsing actions)
```

```
Grammar_for_RIF_FLD_in_RIF-PS  <  Grammar,
  r_description of:  1..* (RIF-FLD < (KRL_model  r_part of: 1..* KRL)),
  r_part: 1 (fc_NLG_rule_type_( .{RIF-PS},  "RIF-FLD_document",
                              .[ 0..1 Annotation   "Document"   "("    0..1 Dialect_directive
                                 0..1 Base_directive    0..* Prefix_directive
                                 0..* Import_directive   0..*  Remote_module_directive
                                 0..1 Group  ")"  ] )
             <   Head_grammar-rule );
```

destinations can be replaced by lists without cardinalities (e.g., [Y, Z]) as long as r_part links are also used for specifying the cardinalities (e.g., X r_part: 0..1 Y, 1..* Z). The use of functions may also be avoided via macros, i.e., by expanding function definitions.

Replicating our work does not require details on the implementation of our system: our ontologies *are* the required *declarative code*. The used inference engine is irrelevant as long as it can handle the specifications. However, some readers might be interested to know that our translation server exploits the parser available at http://goldparser.org while its inference engine was implemented in Pascal Object (for portability purposes) and exploits “tableaux decision procedures” [24]. This server and its inference engine have recently been designed by GTH (http://www.mitechnologies.net). This work on a generic approach for handling KRLs comes from the many problems encountered to handle various versions of FL and other KRLs in the knowledge sharing servers WebKB-1 [25] and WebKB-2 [17, 20].

Our ontology of KRL ontologies (i.e., its core and the specifications of particular KRLs) and our translation server are accessible from http://www.webkb.org/KRLs/. Its interface is similar to Google Translate except that the input and output languages are KRLs and, instead of KRL names, KRL specifications can also be given by the users.

8 Conclusions

One contribution of this article is a generic model for structured abstract or concrete terms. It is simple: only a few types of links and a few distinctions (Tables 4 and 5). This operator + parameters based model permits to define terms in a concise and flexible way, and thus also their presentation and parsing.

A second contribution is the design of a KRL model ontology by representing, aligning and extending various KRL models, and defining their elements via the above cited few links, as illustrated by Tables 3, 6, 7, 8, 9, 10 and 11. Thus, the merged models are also easier to re-use.

A third one is the design of a KRL notation ontology – to our knowledge, the first one – based on the above two cited contributions, as illustrated by Tables 5, 13 and 14.

These three contributions permit to avoid or reduce the problems listed in the introduction and Sect. 2: those of KRL syntactic translation, KRL parser implementation, dynamic extension of notations, etc. Thus, we provide an ontology-based concise alternative to the use of XML as a meta-language for easily creating KRLs that follow KRL ontologies. Therefore, this also complements GRDDL and can be seen as a new research avenue (GRDDL permits to specify where a software agent can find tools – typically XSLT ones – to convert a given KRL to RDF/XML). This avenue is important given the frequent need for applications to (i) integrate or easily 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.

Previous attempts (by the second author of this article) based on directly extending EBNF – or directly representing or generating concrete terms in a KRL or transformation language – required much lengthier specifications that were also more difficult to re-use.

Besides its translation server, the GTH company will use this work in its applications for 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.

One theme of our future work on this approach will be the *generation of parsing actions in parsing rules*, given particular ADTs to use. A second theme will be the representation and integration of more abstract models and notations for KRLs as well as *query languages and programming languages*. A third theme will be the extension of our notation ontology into a full *presentation ontology* with concepts from style-sheets and, more generally, user interfaces.

References

1. Golin, E., Reiss, S.: The specification of visual language syntax. J. Vis. Lang. Comput. **1**(2), 141–157 (1990)
2. Borras, P., Clément, D., Despeyrouz, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: CENTAUR: the system. In: SIGSOFT 1988, 3rd Annual Symposium on Software Development Environments (SDE3), Boston, USA, pp. 14–24 (1988)
3. Attali, I., Parigot, D.: Integrating Natural Semantics and Attribute Grammars: the Minotaur System. INRIA Research Report no. 2339 (1994)
4. Corby, O., Dieng, R.: Cokace: a centaur-based environment for CommonKADS conceptual modeling language. In: ECAI 1996, Budapest, Hungary, pp. 418–422 (1996)
5. Sandberg, D.: Lithe: a language combining a flexible syntax and classes. In: ACM Sigplan-Sigact Symposium on Principles of Programming Languages, POPL 1982, pp. 142–145 (1982)
6. Botting, R.: How Far Can EBNF Stretch? http://cse.csusb.edu/dick/papers/rjb99g.xbnf.html
7. Lapets, A., Kfoury, A.: A user-friendly interface for a lightweight verification system. Electron. Notes Theoret. Comput. Sci. **285**, 29–41 (2012)
8. ODM: Ontology Definition Metamodel, Version 1.1. OMG document formal/2014-09-02 (2014). http://www.omg.org/spec/ODM/1.1/PDF/
9. Feja, S., Witt, S, Speck, A.: BAM: a requirements validation and verification framework for business process models. In: 11th Quality Software International Conference, QSIC 2011, pp. 186–191 (2011)
10. Quan, D.: Xenon: an RDF stylesheet ontology. In: 14th World Wide Web Conference, WWW 2005, Japan (2005)
11. Pietriga, E., Bizer, C., Karger, D.R., Lee, R.: Fresnel: a browser-independent presentation vocabulary for RDF. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 158–171. Springer, Heidelberg (2006)
12. Brophy, M., Heflin, J.: OWL-PL: A Presentation Language for Displaying Semantic Data on the Web. Technical report, Department of Computer Science and Engineering, Lehigh University (2009)
13. Corby, O., Faron-Zucker, C, Gandon, F.: SPARQL template: a transformation language for RDF. In: 25th Journées francophones d'Ingénierie des Connaissances, IC 2014, Clermont-Ferrand, France (2014)
14. RIF-FLD: RIF framework for logic dialects, 2nd edn. In: Boley, H., Kifer, M. (eds.) W3C Recommendation (2013). http://www.w3.org/TR/2013/REC-rif-fld-20130205/
15. Genesereth, M., Fikes, R.: Knowledge Interchange Format, Version 3.0, Reference Manual, Technical Report, Logic-92-1, Stanford University (1992). http://www.cs.umbc.edu/kse/
16. Farquhar, A., Fikes, R., Rice, J.: The ontolingua server: a tool for collaborative ontology construction. Int. J. Hum. Comput. Stud. **46**(6), 707–727 (1997). Academic Press, Inc., MN, USA
17. Martin, P.: Collaborative knowledge sharing and editing. Int. J. Comput. Sci. Inf. Syst. **6**(1), 14–29 (2011)
18. Common Logic: Information technology – Common Logic (CL): a framework for a family of logic-based languages. ISO/IEC 24707:2007(E), JTC1/SC32 (2007)
19. SBVR: Semantics of Business Vocabulary and Business Rules (SBVR), Version 1.0, OMG document formal/08-01-02 (2008). http://www.omg.org/spec/SBVR/1.0/

20. Martin, P.: 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 (2009)
21. Martin, P.: Knowledge representation in CGLF, CGIF, KIF, Frame-CG and formalized-English. In: Priss, U., Corbett, D.R., Angelova, G. (eds.) ICCS 2002. LNCS (LNAI), vol. 2393, pp. 77–91. Springer, Heidelberg (2002)
22. RIF-FLD-OWL: RIF, RDF and OWL Compatibility, 2nd edn. W3C Recommendation, 5 February 2013. http://www.w3.org/TR/2013/REC-rif-rdf-owl-20130205/
23. Šváb-Zamazal, O., Dudáš, M., Svátek, V.: User-friendly pattern-based transformation of OWL ontologies. In: ten Teije, A., Völker, J., Handschuh, S., Stuckenschmidt, H., d'Acquin, M., Nikolov, A., Aussenac-Gilles, N., Hernandez, N. (eds.) EKAW 2012. LNCS, vol. 7603, pp. 426–429. Springer, Heidelberg (2012)
24. Horrocks I.: Optimising Tableaux Decision Procedures for Description Logics. Ph.D. thesis, University of Manchester (1997)
25. Martin, P., Eklund, P.: Embedding knowledge in web documents. Comput. Netw. Int. J. Comput. Telecommun. Netw. **31**(11–16), 1403–1419 (1999)