electronics

MDPI

Article

Integration Strategy and Tool between Formal Ontology and Graph Database Technology

Stefano Ferilli †

Department of Computer Science, University of Bari, 70125 Bari, Italy; stefano.ferilli@uniba.it; Tel.: +39-080-5442293

† Current address: Via E. Orabona 4, 70125 Bari, Italy.

Abstract: Ontologies, and especially formal ones, have traditionally been investigated as a means to formalize an application domain so as to carry out automated reasoning on it. The union of the terminological part of an ontology and the corresponding assertional part is known as a Knowledge Graph. On the other hand, database technology has often focused on the optimal organization of data so as to boost efficiency in their storage, management and retrieval. Graph databases are a recent technology specifically focusing on element-driven data browsing rather than on batch processing. While the complementarity and connections between these technologies are patent and intuitive, little exists to bring them to full integration and cooperation. This paper aims at bridging this gap, by proposing an intermediate format that can be easily mapped onto the formal ontology on one hand, so as to allow complex reasoning, and onto the graph database on the other, so as to benefit from efficient data handling.

Keywords: knowledge representation; formal ontologies; graph databases

Citation: Ferilli, S. Integration Strategy and Tool between Formal Ontology and Graph Database Technology. *Electronics* **2021**, *10*, 2616. https://doi.org/10.3390/electronics10212616

Academic Editor: Agnieszka Konys

Received: 24 September 2021
Accepted: 24 October 2021
Published: 26 October 2021

1. Introduction

Two main perspectives, very different from each other, have been adopted in Computer Science for information storage and handling. The 'Knowledge Base' (KB) perspective is interested in high-level reasoning on the available information, so as to infer implicit information or check the consistency of the information with respect to the reference domain. It is pursued by the Knowledge Representation (KR) branch of Artificial Intelligence (AI) and includes the research field of formal ontologies. The 'Data Base' (DB) perspective is a traditional branch of research in Computer Science interested in developing optimal data organizations aimed at efficient storage, management and retrieval. While clearly complementary, these two perspectives have traditionally been investigated separately. However, due to the increasingly pervasive use of AI solutions in many applications, it would be extremely relevant to take advantage of both.

A new opportunity for cooperation comes from the recent development of *Graph Databases*, a kind of NoSQL DB aimed at optimizing element-driven data browsing rather than batch processing as in traditional relational DBs. Another difference between graph and relational DBs is that the former do not have a pre-defined schema to describe and organize the data, which obviously affects the interpretability and accessibility of the data by the applications and their interoperability. A *graph* is a data structure consisting of nodes (usually representing things) and arcs connecting these nodes (usually representing relationships between things). The arcs may be directed, if they have a direction, and may have attributes or labels qualifying or quantifying the relationship. Interestingly, when the terminological part of an ontology (Tbox, reporting definitions and axioms) is considered in conjunction with the assertional part (Abox, specifying individuals or instances) the result is a so-called *Knowledge Graph* (KG, a kind of KB) [1]. Whilst the literature on ontologies often defines them as encompassing both parts, the relevant literature adopts this very definition for KGs, equating the ontology to the data model only:

- "A knowledge graph is created when you apply an ontology (the data model) to a dataset of individual data points (the [...] data). In other words:

  ontology + data = knowledge graph" [2].
- "Ontologies represent the backbone of the formal semantics of a knowledge graph. They can be seen as the data schema of the graph" [3].
- Knowledge graphs derive from "the core idea of using graphs to represent data, often enhanced with some way to explicitly represent knowledge" [4].
- "In general, a knowledge graph describes objects of interest and connections between them. [...] Many practical implementations impose constraints on the links in knowledge graphs by defining a schema or ontology' [5].

It is clear that graph representation can be the missing link to join the two perspect ives/technologies and take the best from each. Unfortunately, formal ontologies and graph DBs refer to different graph models which cannot straightforwardly be combined together. This paper proposes a technology, called *GraphBRAIN*, aimed at bridging the gap between them through the following contributions:

- Defining a formalism for expressing graph DB schemas, so as to allow data interpretability and applications interoperability;
- Defining the mapping between the graph DB model expressed by this formalism and the standard ontological model adopted in the literature;
- Defining the basics for the operational connection between graph DBs and ontologies, through the two mentioned standards;
- Implementing a software library (intended to act as a wrapper for the DB, permitting only interactions that are compliant to the schema) and tools for the practical exploitation of the proposed formalisms and methodologies.

It would allow graph DB developers to carry out high-level reasoning on their data. Indeed, formal, automated reasoning is much more powerful than the DB's query language, e.g., using ontological reasoning one may check consistency, correctness or completeness of the data. Using rule-based reasoning one may infer information that is not explicitly expressed in the data, possibly defined by complex patterns (as expressible in Logic Programming). Even more, multiple inference strategies (e.g., abduction, argumentation, etc.), not just deduction, can be carried out.

We already developed prototypes of the library, of a tool for building and maintaining the schema and of a tool for handling and consulting the DB based on the schema. This preliminary implementation of GraphBRAIN [6] is currently in use as part of a larger ongoing project [7], aimed at building an integrated system for AI-supported tourism, providing advanced support to end-users, entrepreneurs and institutions involved in touristic activities. It currently includes schemas describing the inter-related domains of 'tourism' (concerning history, cultural heritage items, points of interest, logistics and services, etc.), 'food' (concerning typical dishes and beverages from specific regions), 'computing' (concerning computing devices and their history) [8] and 'lam' (concerning libraries, archives and museums) [9].

Original contributions of this paper are:

- For the first time, a detailed specification of the proposed formalism with a complete account and explanation of its components;
- An extension and refinement of the formalism's components proposed in the previous papers;
- A description of its use as a schema for graph DBs;
- A full mapping of it on a standard ontological format.

This paper is organized as follows. After discussing in Section 2 the basic concepts and related works about formal ontologies and graph databases, Section 3 describes our proposed formalism for interfacing the two technologies. Then, Section 4 shows how schemas expressed in our formalism can be mapped onto graph DBs on one hand and onto a standard ontological format on the other. Finally, Section 5 concludes the paper.

2. Basics and Related Work

According to one of its many definitions in Computer Science, an *ontology* is "a formal, explicit specification of a shared conceptualization" [10]. Therefore, building an ontology requires a conceptualization step, by which: (1) the relevant entities, relationships and their attributes in a domain of interest are identified; (2) names are defined for them; (3) possibly (in the case of *formal* ontologies) axioms are stated expressing what is mandatory, permitted or prohibited in that domain. Explicit or implicit ontology building is pervasive in Computer Science (e.g., when designing E-R diagrams in DBs, or class diagrams in Object-Oriented systems, or predicates, functions and constants in KBs), to determine what can be represented in a (family of) application(s) and to define the rules driving their operation. Indeed, ontologies are key to improving communication among agents, foster systems interoperability and support reuse. Formal Ontologies specifically focus on automated reasoning aimed at making inferences on the available knowledge (concerning both the concepts and their instances) expressed according to the ontology. The main reasoning tasks include KB satisfiability, axiom entailment, concept satisfiability, instance retrieval, classification, query answering [11].

A standard formalism for expressing ontologies and KGs is the Web Ontology Language (OWL) [12]. In fact, a number of reasoners based on OWL are available [13] that provide implementations for all or part of the inferences. OWL is based on the Resource Definition Framework (RDF) [14], originally developed for describing resources on the Web but amenable to knowledge representation in general. RDF graphs are based on a directed graph data model in which nodes are Uniform Resource Identifiers (URIs). A Named Graph is an RDF graph named by a graph URI. An RDF Graph is a collection of RDF Triples, representing arcs, i.e., units of RDF Data of the form:

(Subject, Predicate, Object)

where the Subject and Predicate are URIs and the Object may be a URI or a literal value. Triplestores (or 'Semantic Graph Databases') are DB Management Systems (DBMSs) specifically focusing on RDF Data. Sometimes they need to extend Triples to store extra information, thus actually becoming Column Stores. A common extension are Quads, useful to add context or provenance to triples. Another NoSQL semantic graph database is GraphDB, which may work schema-free or exploiting an RDF ontological schema. Triplestores are specialized for RDF knowledge graphs and thus not optimized for generic data handling, like standard DBMSs. Since data representation constrained to using URIs does not necessarily make sense out of the Automated Reasoning applications (e.g., the Semantic Web), we aim at working with 'normal' DBs but still adopting the graph approach and still being able to carry out formal reasoning on their contents.

A more general structure than Triplestores is provided by graph DBs, based on the Labeled Property Graphs (LPGs) model [15]. In LPGs, both nodes and arcs may have names (called *labels* for nodes and *types* for arcs) and can store *properties* represented as key/value maps. Many arcs, possibly labeled with the same type, may exist between the same pair of nodes. Operationally, nodes and arcs are associated with unique identifiers. The most relevant differences between RDF graphs and LPGs are [16]:

- Nodes are atomic in RDF graphs while they carry information in LPGs; this ensures a much more compact structure in the latter (the estimated decrease in number of nodes is of up to one order of magnitude), which means that not only the former are much less readable, they also cause a significant decay in efficiency, especially in browsing-intensive tasks such as Social Network Analysis or Graph Mining algorithms;
- RDF cannot distinguish different occurrences of the same relationship between the same pair of entities; this is possible in LPGs thanks to the unique identifiers of relationships instances;
- RDF cannot attach properties to instances of relationships; the reification solution (transforming a relationship instance into an object which has relationships to the

original Subject and Object and to the additional properties) worsens readability; another partial solution is via annotations;
- RDF admits multivalued properties (triples with same subject and predicate but different object); these are recovered in LPGs by using arrays as property values;
- The notion of Quad has no equivalent in LPGs, but LPGs have labels, types and properties to carry additional information;
- There is only one kind of node in LPGs, but two in RDF graphs (URIs or literal values for objects of triples).

Whilst not directly related to data storage and management, and seemingly irrelevant, readability may be important for exploitation purposes when a portion of the graph is to be graphically displayed for humans—one of the main strengths of graphs. For the reader's reference, Table 1 provides a comparison of the different terms used to denote the same concepts in the DB, KR and LPG communities. In the following we will use them interchangeably, depending on the needs and context.

Table 1. Alignment between DB, Ontology and LPG terminology.

DBs	KR	LPGs
Entity	Class	Node (label)
Relationship	Object Property	Arc (type)
Attribute	Data Property	Property
Value	Datatype	Value
Instance	Individual	Node/Arc

The relevance of the graph-based approach to DB technology nowadays is witnessed by many big players in the industry developing their own solutions: just consider Google's 'Knowledge Graph', Facebook's 'Social Graph' and Twitter's 'Interest Graph'. All these solutions are proprietary and specifically intended for use in the products of such companies. As a more general-purpose solution we may mention Microsoft Research's 'Graph Engine' (previously known as 'Trinity') [17], a project started in 2010 and released as open source in 2017; however, no recent news is available for it, nor any particular success has been reported for it. In the following we will refer to Neo4j [18], the most popular graph DB according to DB-Engines, a platform that ranks DBMSs according to their popularity [19]. It is currently ranked #17, gaining 4 places in the past year [20]. It has been adopted by many big companies and governmental organizations for several different and relevant use cases, including Recommendation, Biology, Artificial Intelligence and Data Analytics, Social Networks, Data Science and Knowledge Graphs [21].

In Neo4j labels usually represent classes, nodes represent class instances, types represent relationships and arcs represent relationship instances. Each node may be associated with many labels, while each arc may have at most one type. Neo4j comes with a powerful query language (Cypher) and extensive libraries for advanced data manipulation (APOC). However, Neo4j (as most graph DBs) is schema-free: the user may apply any label/type or property to each single node or arc. Only simple 'constraints' may be defined to bias the DB content; while ensuring great flexibility, this causes the lack of a clear semantics for the graph contents. This motivated this work, aimed at proposing a schema formalism for graph DBs. In particular we believe the schema must be in the form of an ontology, so as to enable high-level reasoning on the available knowledge and still benefit from the advantages provided by graph DBs and LPGs. Specifically, we may leverage the advantages of DBMSs (scalability, storage optimization, efficient handling, mining and browsing of the data, etc.) and LPGs (flexibility, expressive power) for handling individuals, and exploit the high-level functionalities of ontological reasoners (allowing formal reasoning on, and consistency or correctness checks of, the data) on the ontological part.

On the methodological side, a few theoretical works analyze the possibilities of cooperation between ontologies and graph DBs, e.g., ref. [22] recognizes the need, but

limited adoption, of logic-based KR for the development of KGs and summarizes some attempts to tackle this issue. Ref. [23] uses Neo4j to show how ontological schemas can be applied to Multilayer graphs (graphs whose labeled edges belong to a number of predetermined classes) and their algebraic counterpart, ontological tensors, also elaborating on complexity.

Other approaches are more practical, aimed at mapping ontologies or KGs to graph DBs. Ref. [24] stores the Freebase KG in Neo4j. As opposed to our proposal, it is not interested in developing ontologies as schemas for the graph DB; actually, it focuses on simple 'querying', not on 'reasoning', and the power or the proposed queries is incomparable to what can be obtained using automated reasoning techniques from AI. Most other works specifically focus on the mapping between OWL and LPGs. G2GML [25] maps OWL (RDF graphs) to PGs to overcome the limitations of SPARQL in implementing traversal or analytics algorithms. It proposed an exchangeable serialization format to support different graph DBMSs and their interoperability, but redefined the PG model. OWL2LPG [26] maps OWL 2 ontologies to an LPG representation, and vice versa, identifying specific kinds of queries that in Neo4j should be both easily expressible and more performant than in WebProtégé 4.0. Since the queries concern the ontology axioms and their revisions, it translates the ontology, not the data. In our approach the ontology stays apart from the DB, where only the data are stored and queried. SciGraph [27] aims at representing OWL ontologies and data as Neo4j graphs. It is strictly 'OWL-centric' and implementation dependant: it reads only formats available to the OWLAPI [28]—an API for OWL which is fully compliant with the official OWL specifications by W3C—and ignores the rest. It is clearly stated that creating ontologies based on the graph and supporting reasoning are not goals of this work. Therefore, it is exactly opposite to our work. VirtualFlyBrain [29] aims at translating only "a well defined subset" of OWL 2 EL ontologies into Neo4j and back in such a way that entailments and annotations (not the syntactic structure) are preserved after the round-trip. Differences from other mappings, such as SciGraph, are quite technical, e.g., having to do with the treatment of blank nodes or with the use of 'safe labels' for typing relations (a safe label is basically the URI with all non-alphanumeric characters being replaced by underscores). The authors point out some 'idiosyncrasies' of the approach, again very technical. Like us they only support datatypes that are supported by both Neo4j and OWL. As opposed to us, they label individuals with their most direct class, while we label them with their top-level class. All these approaches adopted a perspective biased towards ontologies and on their mapping on the graph DB. Since LPGs are more structured than RDF graphs, this direction seems quite obvious, at least syntactically. Since we believe that the DB technology is more mature and widely exploited than the ontology one, we take the opposite perspective and aim at preserving the DB structure and organization, superimposing the ontology on it only so far as it can be easily done.

OWLStar [30] exports Neo4j to OWL but specifying ontological semantics (e.g., OWL-DL interpretations), to be converted to OWL, in edge properties, so the driving perspective is again OWL-centric. It uses RDF* (and its query language SPARQL* that extends SPARQL), in an attempt to bring PGs into RDF by adding syntax to attach properties to edges. Ref. [31] proposes a formal mapping between LPGs and RDF* that can be leveraged to keep the data in the DB and render them in RDF*. However, RDF* is an extension of RDF and thus not compliant with standard reasoners, which prevents immediate reuse of the many reasoners available in the literature for performing ontological reasoning that involves instances. To overcome this limitation we developed a mapping of LPGs onto standard RDF. This required reconciling the differences between the two models and notably the inability of RDF to express datatype properties on relationships.

Some discussions and practical proposals can be found in the Neo4j community blog. The mainstream approach [32] proposes solutions for interoperability of Neo4j data and automated reasoning on them. The former is obtained by exporting Neo4j instances to RDF, e.g., upon request of an ontological reasoner. One way to do this is exporting Neo4j data in JSON using Cypher and the APOC libraries [33] and then further translating the result

into other ontological formats (e.g., using libraries such as [34]). The latter is obtained by importing an RDF ontology into Neo4j, e.g., using the tool provided by the 'official' Neo4j library [35]. The RDF triples specifying the ontology are just transposed into nodes and arcs in the graph, so that the graph DB includes the schema, almost like schemas are stored in relational DBs as tables within the DB itself. On this representation, some (simple) kinds of ontological reasoning (e.g., navigation of the subclass hierachy) are translated into DB queries using Cypher. This solution has several drawbacks. First, the graph would include two disjoint parts, the ontology and the data, to be handled in totally different ways albeit coexisting in the same graph (in relational DBs they would be stored in different schemas, while in graph DBs there is a single overall graph). Second, no formal discussion is provided about what kinds of reasoning can be mapped onto graph DB queries. We expect them to be quite limited if compared to the power of state-of-the-art ontological reasoners. Furthermore, implementing these reasoning facilities is still in charge of the applications accessing the DB. Finally, it does not prevent data that are not compliant with the intended ontology to be inserted into the DB.

Instead, we propose an API, to be exploited by all applications accessing the DB, that wraps the DB and enforces compliance of the data with the intended schemas in both building and consulting the DB. In our vision KB designers must provide pre-specified data schemas, expressed in the form of ontologies for LPGs, that this API will interpret and use to drive all subsequent accesses to the DB. By referring to a schema, the applications will commit to be compliant with it, as in traditional databases. Just like in Triplestores and RDF* this will ensure a tight integration between the data and the schema. As opposed to Triplestores, RDF* and most of the cited works, where the ontology is ingested in the graph, the data/instances (stored in the graph DB) are kept apart from the schema/ontology (specified in a file external to the DB, using an ontological representation format). As discussed in Section 4, we leverage this separation between the data repository and the data schema to obtain the additional opportunity of applying different (but compatible) schemas to the same DB. Indeed, each schema may represent a different, partial view on the same data, allowing to limit or expand the possible interactions depending on specific needs and adding flexibility to our solution. Again, this is not even thinkable in Triplestores.

Proposing an ontological format brings the need for tools to comfortably build, browse and edit the ontologies expressed in this format. Several tools have been proposed, in the literature and practice, for the current standard ontology representations (notably OWL). Each pursues different objectives as regards the construction, editing, annotation and merging of ontologies [36]. Protégé [37,38], based on the OWLAPI, is the most popular and mature. Different versions, extensions and plugins for Protégé have been proposed (e.g., [39,40]), including an online version. Since sometimes they are not completely compatible with the original tool, we will take the OWLAPI as the standard reference in the rest of this paper. Since the ontological format for LPGs we propose in this paper has different features than those available for the RDF graph model, we also developed a corresponding tool for ontology definition and handling. In particular, it allows the ontology designer to specify attributes also for relationships and to specify labels for nodes and types for arcs, which is not allowed by extant ontological standards and tools. Therefore, our starting point was the need to define a schema for the graph DB, and the tool was developed so as to allow the users to comfortably define a schema to be used for building the KB. Then, in order to enable OWL reasoning capabilities, the translation in standard ontology format was a consequential objective. The various approaches proposed in the literature to assess the quality of tools for the construction of ontologies [41] can provide useful hints for improving and extending our tool with advanced features.

3. GraphBRAIN Graph Database Scheme Format

The *GraphBRAIN Schema* (GBS) format we propose to define graph DB schemas consists of an XML file whose tags allow us to exploit the representational features provided

for by the LPG model (we developed a DTD for automated syntax checking of GBS files). In the following, when specifying the GBS file structure, we will adopt the usual notation of square brackets $[\dots]$ to denote optional elements, curly brackets $\{\dots\}$ to denote repeated elements and pipes in parentheses $(\dots|\dots)$ to denote choices. Furthermore, we write XML tag names in boldface, XML tag attribute names in italics and entity or relationship names in smallcaps. Text in plain typeface reports comments useful to understand the various elements and their behavior.

The main structure of the XML with the tags and their nesting is reported in Table 2, where the universal entity ENTITY and the universal relationship RELATIONSHIP, acting resp. as the roots of the entity and relationship hierarchies, are implicitly assumed (remember that in ontological terminology entities correspond to classes and relationships correspond to object properties). Therefore, entities and relationships are to be specified only starting from the first level of specialization, which we will call *top-level*. Since each node (resp., arc) in the graph must be associated with one top-level entity (resp., relationship), the top-level entities (resp., relationships) are to be considered as disjoint. They may be the roots of specialization hierarchies of sub-entities (resp., sub-relationships). The set of direct specializations of a (sub-)entity or (sub-)relationship are in turn disjoint and are not to be intended as a partition: instances that do not fit any of the specializations of a parent (sub-)entity or (sub-)relationship may be directly associated with the parent. Therefore, also the root and intermediate levels of each hierarchy admit instances in the knowledge base. This design choice prevents multiple inheritance (associating an instance to many classes belonging to different branches in the hierarchy). We partially recover this at the level of instances: when two instances of different (sub-)entities represent the same object, we link them using an ALIASOF relationship. The single reference object represented by all these instances takes the union of their attributes.

Table 2. Main structure of GBS files.

domain // tag enclosing the overall ontology
[**imports**]
entities // tag enclosing the classes
{**entity**} // see **(*)**
relationships // tag enclosing the relationships
{**relationship**} // see **(*)**

Entities and relationships are specified using the structure shown in Table 3. **Reference** is used only in relationships to specify their possible domain-range pairs, **taxonomy** is optional (used only if the entity or relationship has sub-entities or sub-relationships) and allows us to conveniently represent the specialization-type assertions; all other object properties are to be specified in the **relationships** section. **Attributes** is mandatory for entities (an entity instance must be described by some attribute) and optional for relationships (a relationship may carry information in its very linking two instances). **Specialization** is a recursive tag, allowing to define hierarchies of sub-entities or sub-relationships. In addition to its own attributes each specialization inherits all the attributes of the (sub-)entities (resp., (sub-)relationships) on the hierarchy path from its specific **specialization** section up to the corresponding top-level entity (resp., relationship).

Table 3. Structure for describing entity and relationship hierarchies in GBS files.

(*) (**entity** | **relationship** | **specialization**) tag
[**references**]
{**reference**}
[**taxonomy**]
{**specialization**} // see **(*)** (recursive)
[**attributes**] specifying the data properties
{**attribute**}

Some tags have XML attributes that specify the details of the item they represent in the schema:

- **domain** tag:
  name the unique identifier for the domain being described
  author the author of the schema
  version the version of the schema
- **entity** tag:
  name the unique identifier for the entity
- **relationship** tag:
  name the unique identifier for the relationship
  inverse the unique identifier for the inverse relationship of *name*
- **reference** tag:
  subject the identifier of the entity that is the domain of the (sub-)relationship
  object the identifier of the entity that is the range of the (sub-)relationship
- **specialization** tag:
  name the unique identifier for the specialization (sub-entity or sub-relationship)
  [***inverse***] the unique identifier for the inverse sub-relationship of *name* (not used for sub-entities)
- **attribute** tag:
  name an identifier for the attribute
  mandatory = (**true** | **false**)
  whether the attribute must take a value in each instance
  distinguishing = (**true** | **false**)
  whether the attribute may concur in distinguish instances having the same values for mandatory attributes
  display = (**true** | **false**)
  whether the attribute represents interesting additional information with respect to mandatory and distinguishing attributes, to be possibly displayed
  datatype = (**integer** | **real** | **boolean** | **string** | **text** | **select** | **tree** | **date** | **entity**)
  [***length***] the maximum allowed number of characters (used only when datatype = string)
  [***target***] an entity name (used only when datatype = entity)

Therefore, the union of mandatory and distinguishing attributes of an entity or relationship can be used to specify a key for uniquely identifying its instances. The union of mandatory, distinguishing and display attributes of an entity or relationship can be used to build and display a summary reporting the most relevant information about the instances.

Regarding datatypes, attributes of type *integer*, *real*, *boolean*, *string*, *text* take an atomic value of the corresponding type, where *text* is intended for free text of any length, differently

from *string* which has a limited maximum length that can be specified in the 'length' attribute. Attributes of type *date* take values in one of the following forms:

- Year;
- Year/month;
- Year/month/day.

where year is any integer, month $\in \{01,\ldots,12\}$ and day $\in \{01,\ldots,31\}$. Attributes of type *select* denote a choice in an enumeration of values, described using the substructure reported in Table 4; attributes of type *tree* denote a choice in a tree of values, described using the recursive substructure shown in Table 5. Attributes of type *entity* denote 1:1 relationships between an instance of the current entity and an instance of another entity (specified in the 'target' attribute of the tag), e.g., the birthplace of an entity Person would be modeled as an attribute of type *entity* with target='Place':

```
<entity name="Person">
   <attributes>
      <attribute name="birthplace" datatype="entity" target="Place"/>
   </attributes>
</entity>
```

Table 4. Structure for describing enumerative attribute values in GBS files.

attribute ... datatype="select" tag
values
{**value**}

Table 5. Structure for describing enumerative attribute values in GBS files.

(**) (**attribute ... datatype="tree"** | **values**) tag
values
{**value**} // see **(**)** (recursive)

As a conventional notation we propose identifiers made up of uppercase letters, lowercase letters or decimal digits only. They should start with an uppercase letter for entity names and enumeration or tree values, or with a lowercase letter for domain, relationship and attribute names. Multi-word names are built by juxtaposing their constituent words, using an uppercase letter for the first letter of each word (except for the first one, as prescribed above). When writing documentation, a relationship 'rel' between an entity 'Subj' and an entity 'Obj' can be represented using the dot notation

Subj.rel.Obj

which is not ambiguous since dots are not allowed in our entity and relationship names.

Tables 6 and 7 show a fragment of a GBS file concerning the domain of computing. We see entity 'Component', representing an electronic component and including a taxonomy of sub-classes, some of which have specific attributes of various type, e.g., sub-class 'Memory' has attributes 'capacity' and 'speed' in addition to those inherited by 'Component' ('name', 'description', 'originalPrice' and 'announcementDate'). In the relationships section we see that relationship 'wasIn' may be established between a 'Component' and an 'Event' (to signify that the component was on show at the event), or between a 'Person' and a 'Place' (meaning that the person was in that place), etc.

Table 6. Sample fragment of ontology in GBS format (part 1).

```
<!--  <!DOCTYPE domain SYSTEM "graphbrain.dtd">  -->
<domain name="retrocomputing" author="stefano" version="1">
   <entities>
      <entity name="Component">
         <attributes>
            <attribute name="name" mandatory="true" datatype="string"/>
            <attribute name="description" mandatory="false" datatype="text"/>
            <attribute name="originalPrice" mandatory="false" datatype="real"/>
            <attribute name="announcementDate" mandatory="false" datatype="date"/>
         </attributes>
         <taxonomy>
            <specialization name="Chip">
               <taxonomy>
                  <specialization name="Logic">
                     <taxonomy>
                        <specialization name="FlipFlop">
                           <attributes>
                              <attribute name="type"
                                    mandatory="false" datatype="select">
                                 <values>
                                    <value name="D"/>
                                    <value name="FK"/>
                                    <value name="JK"/>
                                    <value name="T"/>
                                 </values>
                              </attribute>
                           </attributes>
                        </specialization>
                        <specialization name="Memory">
                           <attributes>
                              <attribute name="capacity"
                                    mandatory="false" datatype="string"/>
                              <attribute name="speed"
                                    mandatory="false" datatype="string"/>
                           </attributes>
                           <taxonomy>
                              <specialization name="EPROM"/>
                              <specialization name="PROM"/>
                              <specialization name="RAM"/>
                              <specialization name="ROM">
                                 <attributes>
                                    <attribute name="content"
                                          mandatory="false" datatype="string"/>
                                 </attributes>
                              </specialization>
                           </taxonomy>
                        </specialization>
                     </taxonomy>
                  </specialization>
                  <specialization name="MicroProcessor">
                     <attributes>
                        <attribute name="speed" mandatory="false" datatype="string"/>
                        <attribute name="bits" mandatory="false" datatype="integer"/>
                     </attributes>
                  </specialization>
                  <specialization name="PLA"/>
                  <specialization name="RRIOT"/>
               </taxonomy>
            </specialization>
            [...]
         </taxonomy>
      </entity>
      [...]
   </entities>
```

Table 7. Sample fragment of ontology in GBS format (part 2).

```
   <relationships>
      <relationship name="wasIn" inverse="hosted">
         <references>
            <reference subject="Company" object="Event"/>
            <reference subject="Company" object="Place"/>
            <reference subject="Component" object="Event"/>
            <reference subject="Event" object="Place"/>
            <reference subject="Person" object="Company"/>
            <reference subject="Person" object="Event"/>
            <reference subject="Person" object="Place"/>
            [...]
         </references>
         <attributes>
            <attribute name="reason" mandatory="false" datatype="string"/>
            <attribute name="position" mandatory="false" datatype="string"/>
         </attributes>
      </relationship>
      [...]
   </relationships>
</domain>
```

Each GBS schema is intended to describe one domain. However, sometimes wider domains involve ontological elements that are already described in more 'basic' schemas (e.g., the schemas for Cultural Heritage, Food and Transportations might be exploited in the ontology aimed at supporting a touristic application) and it might be useful to reuse such schemas, both for standardization of the definitions and for building on existing knowledge. Actually, the combination of many schemas is more powerful a representation than the simple juxtaposition of their elements. Indeed, their shared entities act as bridges that allow, through the relationships available in those domains, to connect proprietary entities of each domain that would not otherwise have a chance to be related with each other. In the GBS framework, classes and relationships in different ontologies are considered the same (and thus are shared) if they have the same name. They may have, however, different attributes, reflecting the different perspectives associated with the different domains. If an attribute is present in different domains it must have the same type in all of them. Moreover, additional cross-schema relationships (and entities) may be defined in the overall ontology, building on the existing ones. GBS schemas support such opportunity by providing for an optional section in which existing schemas can be imported. The structure of this section (delimited by tag **imports** and placed at the beginning of the schema, before the entities and relationships) is as shown in Table 8. The tag attributes are:

- **import** tag:
  schema: the name of a schema to be imported
- **delete** tag:
  elementtype = (entity | relationship)
  elementname: the name of the element to be deleted

Table 8. Structure for describing imported schemas in GBS files.

imports tag
  {**import**}
  [{**delete**}]

Schemas are imported in the same order as specified by the sequence of **import** tags. Definitions of top-level elements (entities or relationships) in an imported schema having the same name as elements defined in previous imported schemas override the previous definitions. Finally, elements defined in the **entities** or **relationships** sections of the importing schema override elements with the same name in all imported schemas. Since it may happen that some elements of the imported schemas are not needed in the current domain, **delete** tags allow to remove them from the overall ontology.

In addition to the API for GBS-based handling of Neo4j, we developed tools for GBS schema/ontology editing and for data management. They were implemented as Web Applications based on the Java Server Faces technology and the PrimeFaces library. JavaScript was used for handling interactive browsing of the graph. A connection to Prolog allows it to carry out rule-based reasoning on selected portions of the data. Obviously Neo4j was used to store the knowledge graph, while Postgres was used to store user and usage data (roles, access rights, change log, etc.). A demo of the tools can be found at http://193.204.187.73:8088/GraphBRAIN/ in the form of a general-purpose system for the collaborative development, management and (personalized) fruition of a KB, in the same spirit as Freebase [42]. After logging into the system, the user may choose a domain and all subsequent interaction is driven by the corresponding GBS schema. Screenshots of the current online prototypes are shown in Figures 1–3.

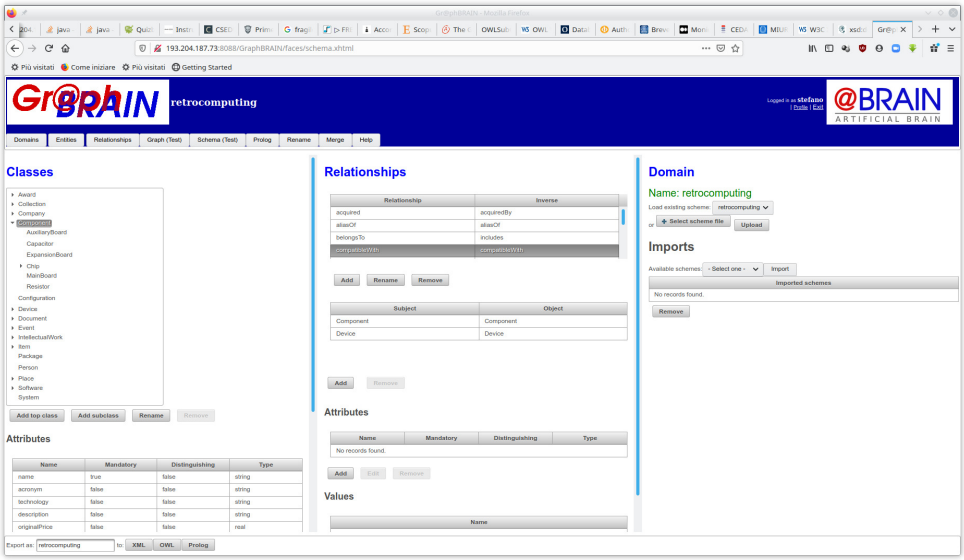
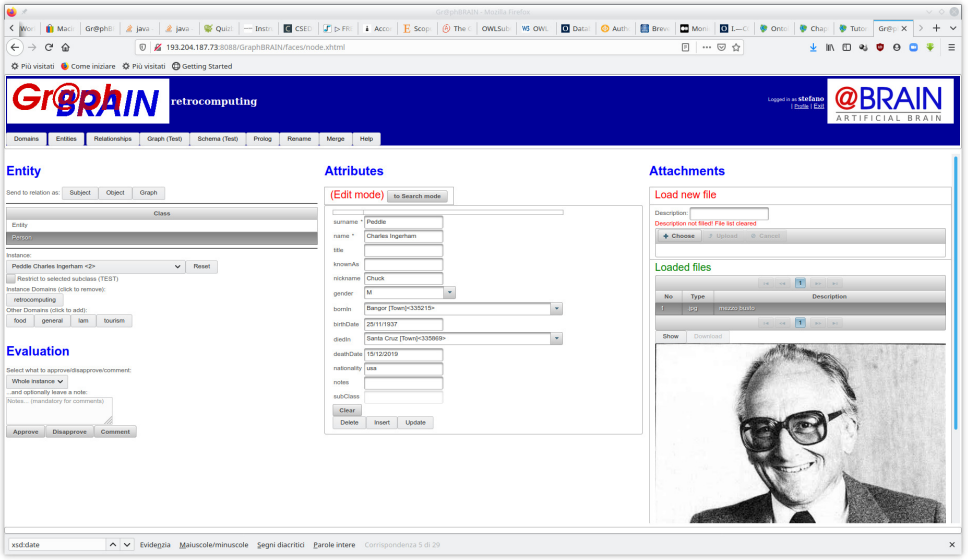
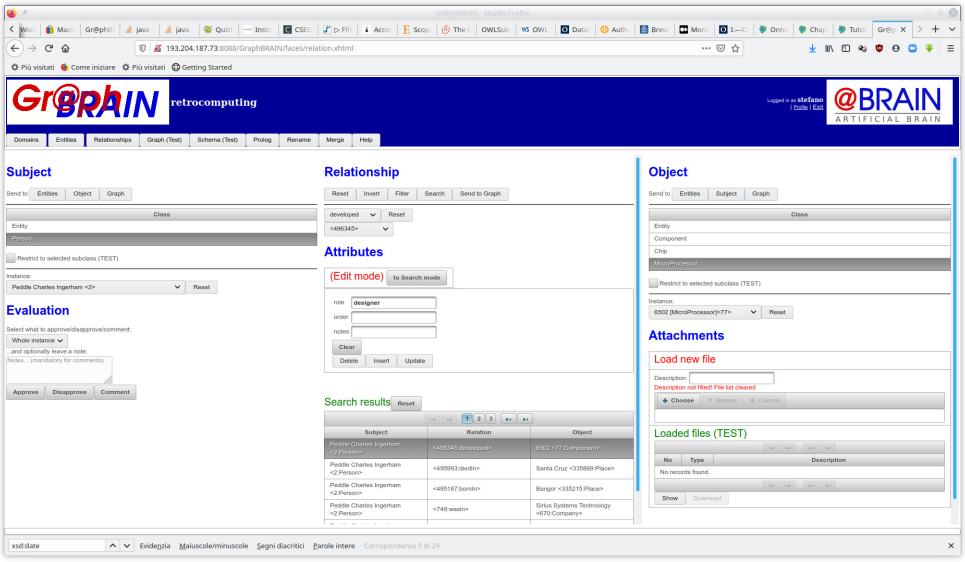


Figure 1. Online editor for GBS schemas/ontologies.

Figure 1 shows the interface for building, editing and browsing GBS schemas/ontologies. In the left-hand-side section the entity hierarchy, with entity attributes and attribute types and values, can be handled. In the center section the same can be done for relationships, also including inverse relationships and references. On the right-hand-side section imports can be handled and existing schemas can be loaded. On the bottom several save and export buttons are available. Figure 2 shows the interactive interface to feed and consult information in the knowledge base by direct interaction. It consists of two form-based tabs, one for entities (Figure 2a) and one for relationships (Figure 2b), allowing the user to insert, update, remove or query instances. The forms are automatically generated by the system from the GBS specification of a schema and interact with the graph DB using our API to enforce consistency with the selected schema. Let us first describe the entity tab. In the left-hand-side section (sub-)entities and corresponding instances can be selected. In the center section a form with the attributes of the selected (sub-)entity is shown, possibly filled with the values from the selected instance. Regarding the relationships tab, the center section allows to choose a relationship, for which subject and object (sub-)entities and corresponding instances can be selected in the left- and right-hand-side sections, respectively. When a triple (subject, relationship, object) is selected, the center section also shows a form with the attributes of the selected (sub-)relationship. If subject and object instances are also selected, a drop-down menu allows selecting a specific relationship instance, in which case the attribute form is filled with the corresponding values. More functions are available (e.g., handling of attachments to the selected instances, or search and collaborative evaluation facilities) but their description is beyond the scope of this paper.



(**a**)



(**b**)

Figure 2. Online interfaces for managing and consulting GBS knowledge bases: (**a**) entities, (**b**) relationships.

Figure 3 shows the tab in which users can display and manually browse the graph. Since the whole KB would be too large to be readable, only a portion thereof is shown in this tab. The portion is dynamically generated so as to focus on the portion of graph of interest to the user based on their profile, optionally starting from selected nodes specified by him. In the figure, the graph was generated for user 'stefano' starting from nodes representing Chuck Peddle (a pioneer in microprocessor design) and the 6502 (one of the earliest and most successful microprocessors on the market), identified by a thicker node border. Different colors of nodes denote different classes (e.g., light blue for Person, yellow for Component, etc.). At a glance, it is possible to see clusters of nodes that represent possibly relevant aggregates of information to be investigated or explored. Note that the nodes and arcs in this view may belong to different schemas, not only to the schema selected for the form-based interaction. Therefore, here the user may discover connections that are beyond the starting domain. The user may pan and zoom on the graph, drag nodes, dynamically follow links, read attributes of nodes and/or arcs, further expand the graph around nodes of interest and run analytics and mining algorithms from menus

on the right-hand-side and contextual menus that appear by clicking on the graph. The information on a node or arc in this view is the complete set of properties for that node or arc, gathered from all domains in which it is involved.

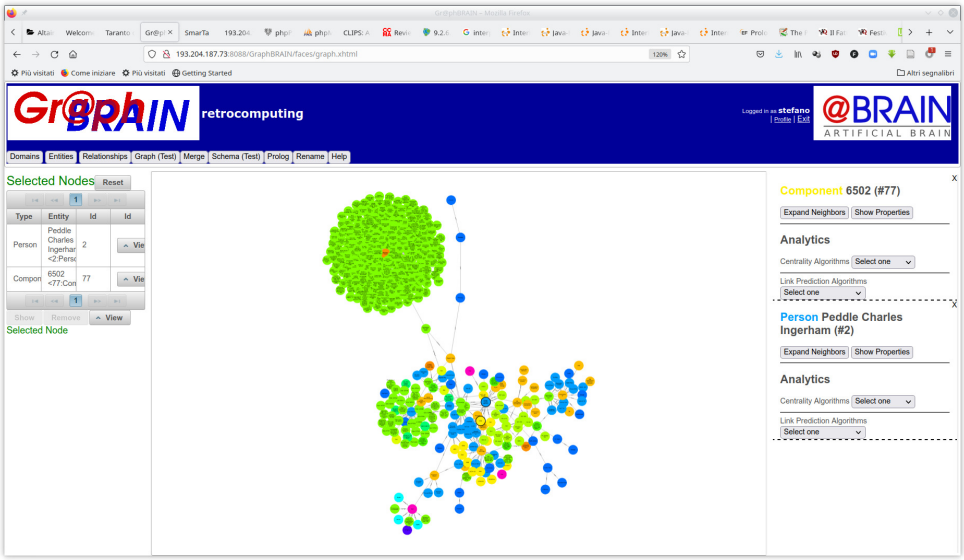


Figure 3. Online interface for browsing GBS knowledge bases.

4. Mapping onto DB and Ontology

Since graph DBs are naturally suited to express knowledge graphs, i.e., knowledge bases underlying given ontologies, a fundamental requirement of our approach is that our schemas can be mapped onto both the DB and to an OWL representation which can then be processed by a reasoner. In this section, we report in detail how these two mappings work in practice.

4.1. Use as a Graph DB Schema

As said, part of the main motivation for defining GBS schemas is to endow LPG-based graph DBs with a schema that ensures a clear semantics to the information pieces they contain and provides directions for their management and interpretation. According to this perspective the DB users will be required to work according to pre-specified data schemas expressed in the form of ontologies. Operationally, the DB will be wrapped into a layer, e.g., in the form of an API (see the previous section), that takes as input a GBS schema specifying the desired domain ontology and controls all interactions, allowing the external applications to manipulate and consult only information items that are compliant with the ontology.

In our approach we also provide an additional opportunity. Specifically, we allow a single graph DB to underlie several domains (schemas), provided that their elements (entities and relationships) are compatible. By *compatible* we mean that for elements having the same name in the different schemas, attributes having the same name must have the same datatype, too. The other attributes, or non-shared elements, can be freely defined. Therefore, using any of such schemas on the DB would provide a partial view of its contents, perhaps representing a different perspective or aimed at limiting access to the DB contents for some users or applications.

Let us now show how the GBS elements are implemented using LPG features. For easy reference, Table 9 summarizes the mapping.

Table 9. Correspondence between GBS elements and LPG features.

GBS Element	LPG Feature
entity instance	node
relationship instance	arc
entity name	label
relationship name	type
domain name	label
entity attribute	node property
relationship attribute	arc property

4.1.1. Entities and Relationships

Leveraging the possibility of using many labels for nodes, each node is labeled with the top-level entity it belongs to and with all the domains for which it is relevant (e.g., 'Herbert Simon' would be labeled with 'Person' for the entity and with 'economy' and 'computing' for the domains). When the same DB underlies several domains, this allows to select only the instances actually involved in a domain of interest. On the other hand, since each arc may take at most one type, we use it for specifying the relationship it expresses. The domains for which a relationship instance is relevant may be inferred from the domain labels of the nodes it connects by considering all the domain labels that are present in both its subject and its object.

4.1.2. Attributes

Concerning attributes, we propose to reserve an attribute name ('*specialization*') to store which is the specific sub-entity (resp., sub-relationship) the entity (resp., relation) instance belongs to. Given the top class (resp., relationship) specified in the labels (resp., types) and the specific sub-entity specified in the 'specialization' property, the path of specializations between these two may be easily recovered bottom-up starting from the latter and climbing the specialization hierarchy in the ontology up to the former (since nodes admit many labels, one might specify all the sub-entities in such a specialization path as labels; for the sake of uniformity with arcs, where this is not possible, we propose the above solution). We also propose to implicitly assume another reserved attribute '*notes*' for both nodes and arcs, that allows to add information not considered by the other, domain-specific attributes.

4.1.3. Attribute Types and Values

Attribute values of types *integer*, *real*, *boolean*, *string* and *text* are stored as literal values for the corresponding DB types, e.g., Neo4j provides the following types matching GBS types: Integer and Float (both subtypes of an abstract type Number), Boolean, and String.

For types *select* and *tree* the string corresponding to the selected value in the list or tree is stored.

An attribute of type *entity* actually corresponds to a relationship between the current instance and an instance of the target entity and thus it is stored in the DB as an arc, connecting the nodes corresponding to these two instances and having the attribute name as type. Note that in our proposed naming policy attribute names start with a lowercase letter, just like relationship names.

Finally, albeit Neo4j provides for temporal types, including 'Date', following [18] we propose to model attributes of type *date* as relationships, as well. We assume the ontology implicitly defines four entities, as shown in Table 10:

DAY representing a specific day of a specific year, with integer attributes *day*, *month*, *year*;

MONTH representing a specific month of a specific year, with integer attributes *month*, *year*;

YEAR representing a year, with a single integer attribute *year*.

TIMELINE representing the overall timeline.

This allows to specify dates at different granularity, differently from the Date type available in Neo4j. Neo4j provides functions for Date truncation to Month or Year, but such truncations actually correspond to the first day of the month or year and thus there is no way to distinguish whether a date like 2020/01/01 actually refers to the specific day or is a truncation for the month (2020/01) or year (2020). A single TIMELINE node is automatically added to the DB. DAY, MONTH or YEAR nodes are automatically added to the DB for each year/month/day, year/month or year value, resp., in date attributes of instances. The DB will also automatically link, using arcs of type BELONGSTO, each DAY node with the corresponding MONTH node, each MONTH node with the corresponding YEAR node and finally all YEAR nodes with the TIMELINE node. This will allow collecting all instances referring to the same date at different levels of granularity. Furthermore, arcs of type FOLLOWS may be added and maintained between adjacent days, months or years in the DB. This will allow to easily extract from the DB time intervals and associated information.

Table 10. Implicit entities and relationships for time handling.

```
<entities>
   <entity name="Timeline"/>
   <entity name="Year">
      <attributes>
         <attribute name="year" mandatory=""true" datatype="integer"/>
      </attributes>
   </entity>
   <entity name="Month">
         <attribute name="month" mandatory=""true" datatype="integer"/>
         <attribute name="year" mandatory=""true" datatype="integer"/>
   </entity>
   <entity name="Day">
         <attribute name="day" mandatory=""true" datatype="integer"/>
         <attribute name="month" mandatory=""true" datatype="integer"/>
         <attribute name="year" mandatory=""true" datatype="integer"/>
   </entity>
</entities>
<relationships>
   <relationship name="belongsTo" inverse="includes">
      <references>
         <reference subject="Day" object="Month"/>
         <reference subject="Month" object="Year"/>
         <reference subject="Year" object="Timeline"/>
      </references>
   </relationship>
   <relationship name="follows" inverse="precedes">
      <references>
         <reference subject="Day" object="Day"/>
         <reference subject="Month" object="Month"/>
         <reference subject="Year" object="Year"/>
      </references>
   </relationship>
</relationships>
```

4.2. Mapping to OWL Format

The other part of our motivation for this work was using the ontology level not only as a DB schema, but also to carry out formal reasoning and consistency or correctness checks on the individuals. As noted in Section 2, a widespread standard for representing ontologies is OWL, based on a different model than LPGs, on which GraphBRAIN ontologies are based. While of course new reasoners may be purposely developed for GBS ontologies, it would be desirable to translate GBS ontologies into OWL, so as to allow immediate reuse of the many existing tools for OWL ontologies. This section provides a strategy for this

translation, aimed at overcoming and reconciling the differences in concepts, perspectives and expressive power between the two ontological models. For compliance with existing tools and reasoners, our implementation of GraphBRAIN adopted the same OWL-API as Protégé for its ontology export functionality, so that the generated ontologies are fully compliant with the standard and may be edited using Protégé. So, in the following, we will use the OWL-RDF syntax accepted by Protégé.

When serializing GBS ontologies to OWL format we propose to use prefix **gbs** in the namespaces, so that they can be easily recognized.

Note that here we just provide the translation for the basic GBS format, expressing the DB schema. Additional tags/features can be added to this basic format to express information intended for use by the ontological level (e.g., transitivity of relationships, etc.), but this is a wide path of investigation and will be developed in future work.

As a reference for the subsequent discussion, we provide in Figures 4–6 some screenshots of a sample GBS ontology (concerning the domain of 'computing') exported in OWL using our API and opened with Protégé.

4.2.1. Entities

Entities in GBSs correspond to Classes in OWL. Each (sub-)entity is declared in OWL using the **owl:Class** statement. Specializations are associated with their immediate superclass using the **rdfs:subClassOf** statement. The implicit universal entity ENTITY, generalizing all (sub-)entities defined in the schema, corresponds to the 'Thing' class in OWL. Since classes are to be considered as disjoint (see Section 3), the axioms for classes in the top level and the specializations of each (sub-)class also include (many) **owl:disjointWith** statements to all of their sibling (sub-)classes, e.g., the following fragment of taxonomy for entity DOCUMENT:

```
<entity name="Document">
   <taxonomy>
      <value name="Printable">
         <taxonomy>
            <value name="Book"/>
            <value name="Letter"/>
         </taxonomy>
      </value>
   </taxonomy>
</entity>
```

translates into the following OWL fragment:

```
<owl:Class rdf:about="http://owl.api.ontology#Document">
   <owl:disjointWith rdf:resource="http://owl.api.ontology#Component"/>
   <owl:disjointWith rdf:resource="http://owl.api.ontology#Device"/>
   <owl:disjointWith rdf:resource="http://owl.api.ontology#Person"/>
   <owl:disjointWith rdf:resource="http://owl.api.ontology#Place"/>
</owl:Class>

<owl:Class rdf:about="http://owl.api.ontology#Printable">
   <rdfs:subClassOf rdf:resource="http://owl.api.ontology#Document"/>
</owl:Class>

<owl:Class rdf:about="http://owl.api.ontology#Book">
   <rdfs:subClassOf rdf:resource="http://owl.api.ontology#Printable"/>
   <owl:disjointWith rdf:resource="http://owl.api.ontology#Letter"/>
</owl:Class>
```

```
<owl:Class rdf:about="http://owl.api.ontology#Letter">
   <rdfs:subClassOf rdf:resource="http://owl.api.ontology#Printable"/>
   <owl:disjointWith rdf:resource="http://owl.api.ontology#Book"/>
</owl:Class>
```

In the OWL translation, each entity instance is associated with the sub-class specified by its 'specialization' attribute of the top-level class specified in its labels.

In Figure 4, in the left-hand-side area of the window we see the class hierarchy, in which class 'Computer' (a sub-class of 'Device') has been selected and corresponding details are shown in the right-hand-side area. We may notice that Computer has in turn several sub-classes.

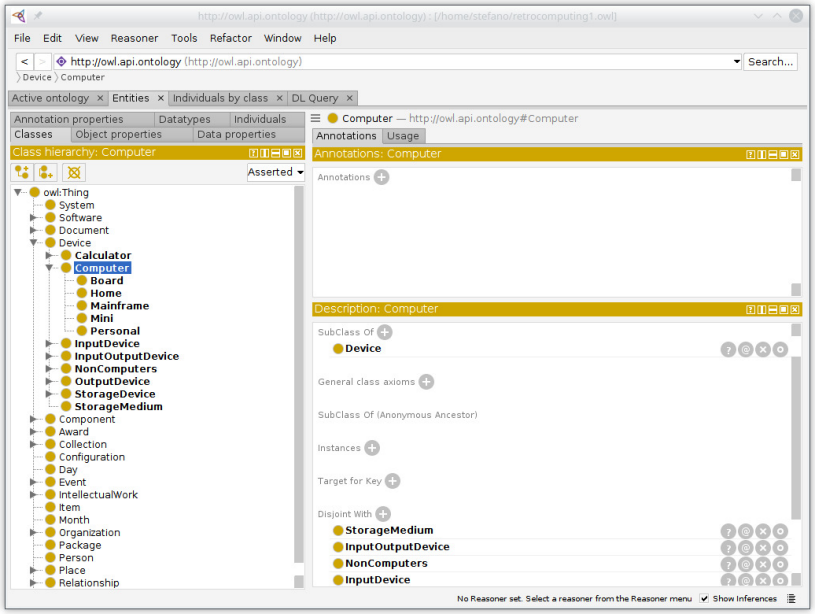


Figure 4. OWL translation of a sample GBS ontology loaded in Protègè: classes.

4.2.2. Relationships

Relationships in GBSs correspond to Object Properties in OWL. Each (sub-)relationship is declared in OWL using the **owl:ObjectProperty** construct. Specializations are associated with their immediate super-relationship using the **rdfs:subPropertyOf** construct. The implicit universal relationship RELATIONSHIP, generalizing all (sub-)relationships defined in the schema, corresponds to the 'topObjectProperty' object property in OWL. Subject and Object entities acting as references of a relationship in GBSs correspond to Domain and Range of the Object Property in OWL, expressed by constructs **rdfs:domain** and **rdfs:range**, respectively. The name for the inverse of a relationship in GBS is translated into OWL using the **owl:inverseOf** construct.

GBSs may use the same relationship name applied to possibly many Subject–Object pairs as references. This cannot be expressed directly in OWL. Adding all the Subject (resp., Object) entities as domain (resp., range) classes to the corresponding OWL object property would be interpreted in OWL as the intersection of the Subject (resp., Object) classes as the domain (resp., range) of the OWL object property.

When the subject (resp., object) of all references in a relationship is the same, the logical disjunction (OR) operator of the classes in the object (resp., subject) would solve the problem, e.g., the following relationship:

```
<relationship name="produced" inverse="producedBy">
   <references>
      <reference subject="Company" object="Device"/>
      <reference subject="Company" object="Software"/>
   </references>
</relationship>
```

meaning that companies may produce devices or software (but a specific company might produce both, or either, or none of them), might be represented as a single object property

Company.produced.(Device OR Software)

and the following relationship:

```
<relationship name="belongsTo" inverse="includes">
<references>
      <reference subject="Device" object="Collection"/>
      <reference subject="Document" object="Collection"/>
   </references>
</relationship>
```

meaning that devices or documents may belong to collections, might be represented as a single object property

(Device OR Document).belongsTo.Collection

However, in general, when the subjects and objects both involve many classes, adding the logical disjunction (OR) of the Subject entities as the domain and of the Object entities as the range would be a wrong translation, because it would not prevent OWL from accepting instances from incompatible Subject–Object pairs, e.g., if relationship WASIN can be applied to reference pairs COMPANY-EVENT and PERSON-PLACE:

```
<relationship name="wasIn" inverse="hosted">
   <references>
      <reference subject="Company" object="Event"/>
      <reference subject="Person" object="Place"/>
   </references>
</relationship>
```

using '(Company OR Person)' as the domain and '(Event OR Place)' as the range:

(Company OR Person).wasIn.(Event OR Place)

would admit relating an instance of Company to an instance of Place, which was not intended by the GBS ontology. We reconcile this by introducing in OWL one object property for each GBS relationship, using the same name and the disjunction (OR) of the Subject entities as the domain and the disjunction (OR) of the Object entities as the range. Then, for each Subject–Object reference pair for a relationship 'rel' in GBS, in OWL we define a new relationship 'rel_Subject_Object' with domain Subject and range Object, as a subObjectProperty (OWL feature **rdfs:subPropertyOf**) of 'rel' (not ambiguous since underscores are not allowed in GBS entity and relationship names).

The OWL translation of the previous example would be:

```
<owl:ObjectProperty rdf:about="http://owl.api.ontology#hosted"/>

<owl:ObjectProperty rdf:about="http://owl.api.ontology#wasIn">
   <owl:inverseOf rdf:resource="http://owl.api.ontology#hosted"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:about="http://owl.api.ontology#wasIn_Company_Event">
   <rdfs:subPropertyOf rdf:resource="http://owl.api.ontology#wasIn"/>
   <rdfs:domain rdf:resource="http://owl.api.ontology#Company"/>
   <rdfs:range rdf:resource="http://owl.api.ontology#Event"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://owl.api.ontology#wasIn_Person_Place">
   <rdfs:subPropertyOf rdf:resource="http://owl.api.ontology#wasIn"/>
   <rdfs:domain rdf:resource="http://owl.api.ontology#Person"/>
   <rdfs:range rdf:resource="http://owl.api.ontology#Place"/>
</owl:ObjectProperty>
```

In principle, we should add some constraint telling OWL that 'rel' is an 'abstract' relationship, i.e., it does not admit direct instances (any instances must belong to a subObjectProperty of 'rel'), but unfortunately this cannot be expressed in OWL [43]. However, since the OWL functionality will be applied only to the instances in the DB, which are controlled by the GBS ontology, in practice this constraint will be implicitly enforced for explicit instances. Only the reasoning might identify individuals belonging to 'rel'. Another option would be defining only the subObjectProperties, but semantically we would miss the information that they express the same concept declined for different references and operationally we would miss the opportunity of defining in 'rel' a core set of properties that apply to all of its sub-relationships. On the other hand, defining attributes (Datatype Properties) on Object Properties is forbidden by OWL and must be handled appropriately in the translation, as we will see in the next sections.

When the *name* of a relationship and its *inverse* in GBS are the same, instead of adding the inverse object property, the object property is labeled as symmetric, using the **owl:SymmetricProperty** construct, e.g., ALIASOF:

```
<relationship name="aliasOf" inverse="aliasOf">
   <references>
      <reference subject="Company" object="Company"/>
      <reference subject="Person" object="Person"/>
   </references>
</relationship>
```

is translated as:

```
<owl:ObjectProperty rdf:about="http://owl.api.ontology#aliasOf">
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#SymmetricProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://owl.api.ontology#aliasOf_Company_Company">
   <rdfs:subPropertyOf rdf:resource="http://owl.api.ontology#aliasOf"/>
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#SymmetricProperty"/>
   <rdfs:domain rdf:resource="http://owl.api.ontology#Company"/>
   <rdfs:range rdf:resource="http://owl.api.ontology#Company"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://owl.api.ontology#aliasOf_Person_Person">
   <rdfs:subPropertyOf rdf:resource="http://owl.api.ontology#aliasOf"/>
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#SymmetricProperty"/>
   <rdfs:domain rdf:resource="http://owl.api.ontology#Person"/>
   <rdfs:range rdf:resource="http://owl.api.ontology#Person"/>
</owl:ObjectProperty>
```

In Figure 5, the left-hand-side area reports the hierarchy of object properties corresponding to GBS relationships, all depending from the universal class 'topObjectProperty'. Object properties 'aliasOf' and 'belongsTo' have been expanded, showing the subproperties generated by the corresponding references. 'belongsTo_Award_Collection' is selected, whose details are reported on the right-hand-side area. Specifically, we see that its domain is class 'Award' and its range is class 'Collection' and that it is a subPropertyOf class 'belongsTo'.

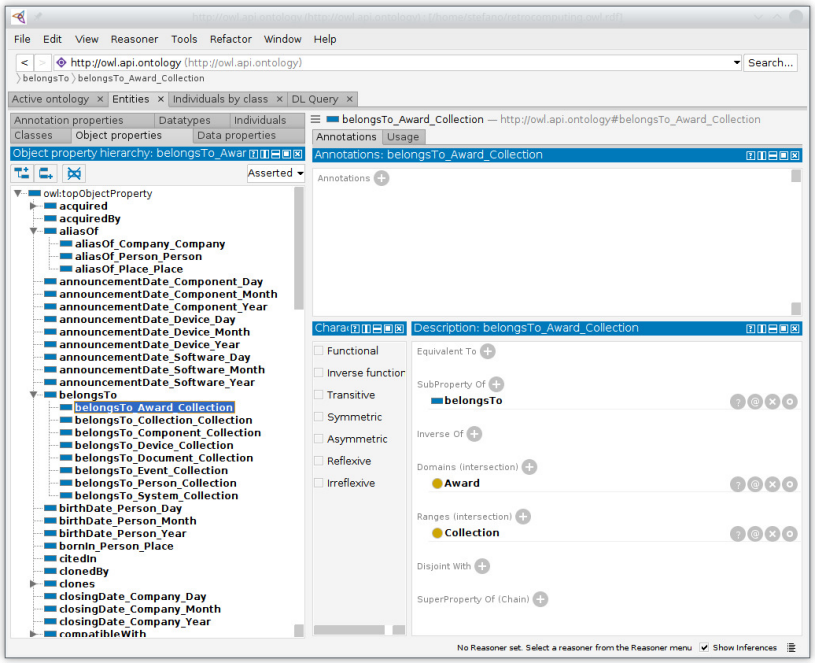


Figure 5. OWL translation of a sample GBS ontology loaded in Protègè: object properties.

4.2.3. Data Types

Attributes of data types *integer*, *real*, *boolean*, *string* and *text* are translated into OWL using the corresponding datatypes **xsd:integer**, **xsd:decimal**, **xsd:boolean**, **xsd:string** (for both string and text). Note that OWL provides several versions of some datatypes.

For types *select* and *tree*, we define in OWL an Enumerated datatype specifying the values in the list or tree. We do not need to store the tree structure, so we can flatten the tree values into a list, because (a) in GBS the tree is just a conceptual aid to the users, in order to build the interfaces to the DB; and (b) there are no duplicate values in the tree, e.g., the values for attribute 'gender' of entity 'Person' in this GBS fragement:

```
<entity name="Person">
   <attributes>
      <attribute datatype="select" mandatory="false" name="gender">
         <values>
            <value name="M"/>
            <value name="F"/>
         </values>
      </attribute>
</entity>
```

would be specifies as the range of the datatype property 'gender_Person' made up of the list of string values {M, F}:

```
<owl:DatatypeProperty rdf:ID="gender_Person">
  <rdfs:range>
    <owl:DataRange>
      <owl:oneOf>
        <rdf:List>
           <rdf:first rdf:datatype="&xsd;integer">M</rdf:first>
           <rdf:rest>
              <rdf:List>
                 <rdf:first rdf:datatype="&xsd;integer">F</rdf:first>
                 <rdf:rest rdf:resource="&rdf;nil" />
              </rdf:List>
           </rdf:rest>
        </rdf:List>
      </owl:oneOf>
    </owl:DataRange>
  </rdfs:range>
</owl:DatatypeProperty>
```

Attributes of type *entity* actually correspond to a relationship between the current instance and an instance of the *target* entity and thus they have as values the individuals of the corresponding *target* class.

Finally, OWL provides several datatypes for expressing the GBS *date* type (e.g., **xsd:date**). While for some purposes they may be enough for representing and handling this type, having an ontological description of time may allow more powerful reasoning. Recently, a specific OWL ontology of temporal concepts, OWL-Time [44], has been proposed for describing and handling temporal properties. This might be another solution, in the same spirit as our proposal but more complex and powerful. We reproduce the strategy discussed in Section 3. This option involves adding to the OWL ontology classes 'Day', 'Month', 'Year' and 'Timeline', and object properties 'belongsTo_Day_Month', 'belongsTo_Month_Year' and 'belongsTo_Year_Timeline', as specializations of a general 'belongsTo' relationship, to suitably connect these classes.

4.2.4. Entity Attributes

As usual in databases, attributes in different entities might have the same name but different meaning. Since in OWL each name must identify one element, we disambiguate by merging the attribute name with the entity it belongs to. Therefore, attribute 'attr' of entity 'Ent' will be stored as 'attr_Ent' in the OWL version of the ontology (not ambiguous since underscores are not allowed in entity names).

Attributes of data types *integer*, *real*, *boolean*, *string* and *text* are translated into OWL as datatype properties having the attribute class as the domain and the corresponding primitive OWL datatype as the range (as specified in the previous section).

As shown in the previous section, attributes of types *select* and *tree* are translated into a datatype property having the attribute class as the domain and an Enumerated Type as the range.

In Figure 6, on the left-hand-side, the data properties are shown, all depending from the 'topDataProperty' root. Some correspond to entity attributes. 'buttons_Mouse' is selected, showing its domain class ('Mouse') and the associated datatype ('integer').

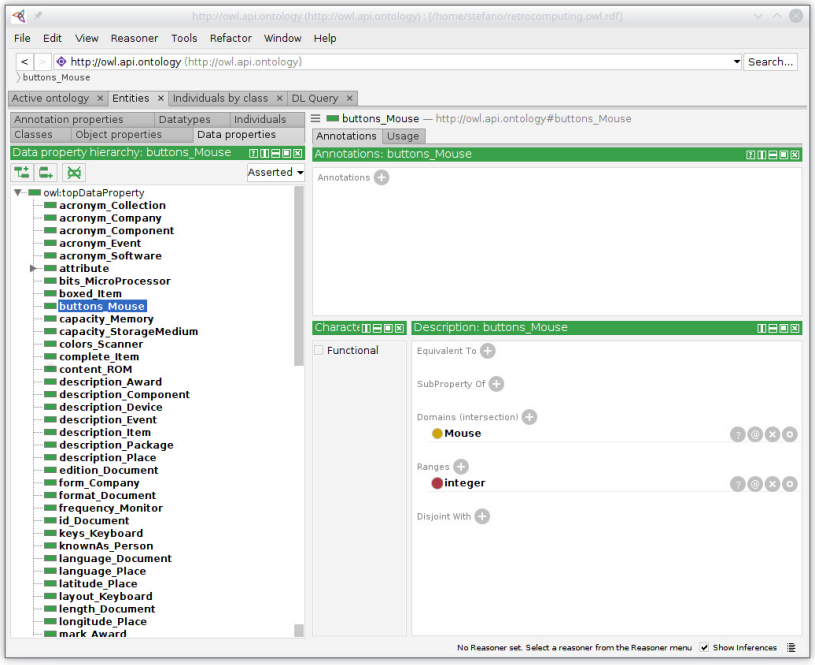


Figure 6. OWL translation of a sample GBS ontology loaded in Protègè: data properties.

Attributes of type *entity*, actually corresponding to a relationship between the instances of the attribute entity and those of the target entity, are translated as object properties having the attribute class as domain and the target class as range. This is compliant with our proposed naming policy, since attribute names start with a lowercase letter just like object property names. Specifically, since the target class individual associated with each domain class instance is unique, we also set this object property in OWL as functional (**owl:FunctionalProperty**).

Finally, according to the what reported in the previous section, attributes of type *date* can be modeled as datatype properties or as object properties.

In Figure 5, some object properties correspond to entity attributes of type 'entity' or 'date', e.g., 'announcementDate_Component_Day' represents the object property expressing the 'announcementDate' attribute (of type 'Date') of entity 'Component' (which is the domain of this object property), linking it to entity 'Day' (acting as the range of this object property).

4.2.5. Relationship Attributes

As previously noted, OWL does not allow expressing attributes (datatype properties) on relationships (object properties). In the ontological practice this is solved by a process of *reification*, by which the object property becomes a class, to which the attributes can be associated, and considering it as the subject of two object properties, linking it respectively to its domain and range. We adopt the same strategy in our translation. After turning the relationship into a class, its attributes are handled as reported in the previous section, e.g., considering again relationship WASIN:

```
<relationship name="wasIn" inverse="hosted">
   <attributes>
      <attribute datatype="string" mandatory="false" name="reason"/>
      <attribute datatype="date" mandatory="false" name="startDate"/>
   </attributes>
</relationship>
```

the OWL classes, datatype properties (for attribute 'reason' and object properties (for attribute 'startDate') generated after reification would be:

```
<owl:Class rdf:about="http://owl.api.ontology#wasIn">
   <rdfs:subClassOf rdf:resource="http://owl.api.ontology#Relationship"/>
</owl:Class>

<owl:DatatypeProperty rdf:about="http://owl.api.ontology#reason_wasIn">
   <rdfs:domain rdf:resource="http://owl.api.ontology#wasIn"/>
   <rdfs:range rdf:resource="http://owl.api.ontology#string"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:about="http://owl.api.ontology#startDate_wasIn_Day">
   <rdfs:subPropertyOf rdf:resource="http://owl.api.ontology#RelationshipProperty"/>
   <rdfs:domain rdf:resource="http://owl.api.ontology#wasIn"/>
   <rdfs:range rdf:resource="http://owl.api.ontology#Day"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://owl.api.ontology#startDate_wasIn_Month">
   <rdfs:subPropertyOf rdf:resource="http://owl.api.ontology#RelationshipProperty"/>
   <rdfs:domain rdf:resource="http://owl.api.ontology#wasIn"/>
   <rdfs:range rdf:resource="http://owl.api.ontology#Month"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://owl.api.ontology#startDate_wasIn_Year">
   <rdfs:subPropertyOf rdf:resource="http://owl.api.ontology#RelationshipProperty"/>
   <rdfs:domain rdf:resource="http://owl.api.ontology#wasIn"/>
   <rdfs:range rdf:resource="http://owl.api.ontology#Year"/>
</owl:ObjectProperty>
```

While this transformation is required only for relationships having attributes, it may not be appropriate to have some relationships translated as object properties (those with no attributes) and others translated as classes. Therefore, we translate all relationships both as classes (possibly with attributes), reproducing their hierarchy under the RELATIONSHIP top-level class, and as object properties.

4.3. Logical Architecture and Workflow

Figure 7 provides a high-level graphical description of the involved components and the flow of information in GraphBRAIN. The GraphBRAIN system is shown as a grey box, including the graph DB that stores the data, the GBS schemas and the API. Shapes denote kinds of information: the schemas (empty shapes) define the allowed information patterns and information (filled shapes) is stored in the DB based on these patterns (the shape of the information blocks is the same as that of the schema they refer to). Some information may belong to different schemas (shown as overlapping shapes in the DB). Note that the schemas are kept apart from the data, that several schemas may be used on the same DB and that the API is independent of the schemas (the same API may be used on all DBs, since the schema to be used are provided as an input during the operations).

All interactions between external entities and the system pass through the API. Applications (e.g., the Web Application described in Section 3) may ask the API to provide information about the patterns in one of the available schemas and use them to inform their data handling requests. When they request to store (insert/update) or retrieve (read) information based on a schema, the API checks that their structure is consistent with the patterns defined in the specified schemas, in which case the request is fulfilled. Requests for information patterns not defined in the scheme (the triangle in the figure) are blocked. Given an existing KG, its ontological part can be imported in a schema; if required, also its instances can be imported into the DB based on the imported schema. Conversely, a schema can be exported to an ontology for a KG and possibly the corresponding data in the DB can be exported as instances to the KG, as well.

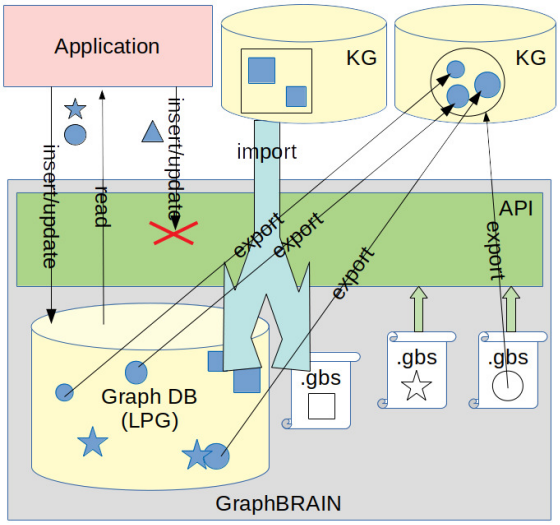


Figure 7. Interplay among components and roles.

5. Conclusions

Formal ontologies, described as RDF graphs, have traditionally been investigated as a means to formalize an application domain so as to carry out automated reasoning on it. The union of the terminological and assertional parts of an ontology is known as a Knowledge Graph. On the other hand, database technology has ever since focused on the optimal organization of data so as to boost efficiency in their storage, management and retrieval. Graph databases, based on the Labeled Property Graphs (LPG) model, are a recent technology specifically focusing on element-driven data browsing rather than on batch processing. Furthermore, graph databases are typically schema-less, preventing uniform interpretation of the data by, and interoperability of, the applications. In spite of the patent and intuitive complementarity and connections between these technologies, the underlying graph models are partially incompatible and little exists to bring them to full integration and cooperation.

Whilst most efforts in the literature are OWL-centric and aimed at mapping RDF ontologies to LPGs, we place more emphasis on the database, so as to benefit from efficient data handling, and aim at enriching it with reasoning capabilites that exploit as much as possible the flexibility of the LPG model. To the best of our knowledge this is a completely novel perspective in the literature.

For this purpose, we proposed to express database schemas in the form of ontologies, so as to clearly describe the database content and to allow users to carry out complex reasoning on it, beyond the queries allowed by the database query language. Specifically, we defined an intermediate format (GBS) that can be easily mapped onto formal ontology standards on one hand and onto the graph database structure on the other. A peculiarity of our approach is that many schemas/ontologies can be applied to the same graph to express different domains or perspectives on its content. These ontologies may share classes and relationships, allowing cross-fertilization of the knowledge from the corresponding domains. The use of ontologies enables multistrategy formal, automated reasoning on the data, that goes much beyond what simple queries can do.

In this paper, for the first time, we provided the full specification for GBS and discussed how its components can be mapped on a most famous graph DB (Neo4j) and on a standard formal ontology (OWL). Operationally, this framework is supported by an API that is meant to act as a wrapper for the DB, ensuring that its content is compliant with a GBS schema, and that can connect the instances in the DB with an ontological reasoner using the same schema as an ontology. Based on this API many different applications may exploit this powerful combinations of databases and ontologies in their functions. Among these applications we developed a tool to build, browse and edit GBS schemas, and a tool to

add, edit and consult the DB content according to a pre-specified schema. Such a tool is described in this paper, as well.

The API and tools are continuously under development to be extended and refined, and research is ongoing to further improve the mapping between the GBS and OWL formalisms, so as to fully exploit their respective advantages in both the instance (database) and the schema (ontology) part of the knowledge graph. In particular, we are working at the extension of the schema format with additional tags/features to express information that may improve the effectiveness of reasoning at the ontological level.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The author would like to thank Domenico Redavid for the useful discussions on the methodology, and Davide Di Pierro for their contribution in the implementation of the schema management section. Grateful thanks go to Artificial Brain S.r.l. for implementing most of the Web Application.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Ehrlinger, L.; Wolfram, W. Towards a definition of knowledge graphs. In Proceedings of the SEMANTICS 2016: Posters and Demos Track, CEUR Workshop Proceedings, Leipzig, Germany, 12–15 September 2016; CEUR-WS.org: Aachen, Germany, 2016; Volume 1695.
2. Schrader, B. *What Is the Difference between an Ontology and a Knowledge Graph? (White Paper)*; Technical Report; Enterprise Knowledge: Arlington, VA, USA, 2021.
3. Available online: https://www.ontotext.com/knowledgehub/fundamentals/what-is-a-knowledge-graph/ (accessed on 8 September 2021).
4. Hogan, A.; Blomqvist, E.; Cochez, M.; d'Amato, C.; Melo, G.D; Gutierrez, C.; Kirrane, S.; Gayo, J.E.L; Navigli, R.; Neumaier, S.; et al. Knowledge Graphs. *ACM Comput. Surv.* **2021**, *54*, 1–37. [CrossRef]
5. Noy, N.; Gao, Y.; Jain, A.; Narayanan, A.; Patterson, A.; Taylor, J. Industry-Scale Knowledge Graphs: Lessons and Challenges. *Commun. ACM* **2019**, *62*, 36–43. [CrossRef]
6. Ferilli, S.; Redavid, D. The GraphBRAIN System for Knowledge Graph Management and Advanced Fruition. In *Foundations of Intelligent Systems*; Springer: Berlin/Heidelberg, Germanny, 2020; Volume 12117, *LNAI*, pp. 308–317.
7. Ferilli, S.; De Carolis, B.; Buono, P.; Di Mauro, N.; Angelastro, S.; Redavid, D. Una piattaforma intelligente per la gestione integrata del settore turistico. In Proceedings of the Primo Convegno Nazionale CINI sull'Intelligenza Artificiale—Workshop on AI for Cultural Heritage, Rome, Italy, 18–19 March 2019; CINI: Rome, Italy; p. 2. (In Italian)
8. Ferilli, S.; Redavid, D. An Ontology and a Collaborative Knowledge Base for History of Computing. In Proceedings of the 1st International Workshop on Open Data and Ontologies for Cultural Heritage (ODOCH-2019), at the 31st International Conference on Advanced Information Systems Engineering (CAiSE 2016), Central Europe (CEUR) Workshop Proceedings, Rome, Italy, 3 June 2019; CEUR-WS.org: Aachen, Germany, 2019; Volume 2375, pp. 49–60.
9. Ferilli, S.; Redavid, D. An Ontology and Knowledge Graph Infrastructure for Digital Library Knowledge Representation. In *Digital Libraries: The Era of Big Data and Data Science*; Communications in Computer and Information Science; Springer: Berlin/Heidelberg, Germany, 2020; Volume 1177, pp. 47–61.
10. Studer, R.; Benjamins, R.; Fensel, D. Knowledge engineering: Principles and methods. *Data Knowl. Eng.* **1998**, *25*, 161–198. [CrossRef]
11. Rudolph, S. Foundations of Description Logics. In *Reasoning Web. Semantic Technologies for the Web of Data: 7th International Summer School 2011, Galway, Ireland, 23–27 August 2011, Tutorial Lectures*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 76–136.
12. Available online: https://www.w3.org/OWL/ (accessed on 23 October 2021).
13. Available online: http://owl.cs.manchester.ac.uk/tools/list-of-reasoners/ (accessed on 23 October 2021).
14. Available online: https://www.w3.org/RDF/ (accessed on 23 October 2021).
15. Rodriguez, M.; Neubauer, P. Constructions from dots and lines. *Bull. Am. Soc. Inf. Sci. Technol.* **2010**, *36*, 35–41. [CrossRef]
16. Available online: https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/ (accessed on 8 September 2021).
17. Shao, B.; Wang, H.; Li, Y. Trinity: A distributed graph engine on a memory cloud. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13), New York, NY, USA, 22–27 June 2013; ACM: New York, NY, USA, 2013; pp. 505–516.
18. Robinson, I.; Webber, J.; Eifrem, E. *Graph Databases*, 2nd ed; O'Reilly Media: Sebastopol, CA, USA, 2015.

19. Available online: https://db-engines.com/en/ranking (accessed on 23 October 2021).
20. Available online: https://db-engines.com/en/system/GraphDB%3BNeo4j (accessed on 8 September 2021).
21. Available online: https://neo4j.com/use-cases/ (accessed on 23 October 2021).
22. Krötzsch, M. Ontologies for Knowledge Graphs? In Proceedings of the 30th International Workshop on Description Logics, Montpellier, France, 18–21 July 2017; CEUR Workshop Proceedings; CEUR-WS.org: Aachen, Germany, 2017; Volume 1879.
23. Drakopoulos, G.; Kanavos, A.; Mylonas, P.; Sioutas, S.; Tsolis, D. Towards a framework for tensor ontologies over Neo4j: Representations and operations. In Proceedings of the 8th International Conference on Information, Intelligence, Systems & Applications, IISA 2017, Larnaca, Cyprus, 27–30 August 2017; pp. 1–6.
24. Elbattah, M.; Roushdy, M.; Aref, M.; Salem, A.B.M. Large-scale ontology storage and query using graph database-oriented approach: The case of Freebase. In Proceedings of the 2015 IEEE Seventh International Conference on Intelligent Computing and Information Systems (ICICIS), Cairo, Egypt, 12–14 December 2015; pp. 39–43.
25. Chiba, H.; Yamanaka, R.; Matsumoto, S. G2GML: Graph to Graph Mapping Language for Bridging RDF and Property Graphs. In *The Semantic Web—ISWC 2020*; Springer: Cham, Switzerland, 2020; pp. 160–175.
26. Available online: https://protegeproject.github.io/owl2lpg (accessed on 8 September 2021).
27. Available online: https://github.com/SciGraph/SciGraph/wiki/Neo4jMapping (accessed on 8 September 2021).
28. Available online: http://owlcs.github.io/owlapi (accessed on 23 October 2021)
29. Available online: https://github.com/VirtualFlyBrain/neo4j2owl (accessed on 8 September 2021).
30. Available online: https://github.com/cmungall/owlstar (accessed on 8 September 2021).
31. Hartig, O. Foundations to Query Labeled Property Graphs using SPARQL. In *Proceedings of the CEUR Workshop Proceedings Joint Proceedings of the 1st International Workshop on Semantics for Transport and the 1st International Workshop on Approaches for Making Data Interoperable Co-Located with 15th Semantics Conference (SEMANTiCS 2019)*; CEUR-WS.org: Aachen, Germany, 2019; Volume 2447.
32. Available online: https://neo4j.com/blog/ontologies-in-neo4j-semantics-and-knowledge-graphs/ (accessed on 8 September 2021).
33. Available online: https://neo4j.com/labs/apoc/4.1/export/json/ (accessed on 23 October 2021).
34. Available online: https://www.w3.org/2016/01/json2rdf.html (accessed on 23 October 2021).
35. Available online: https://neo4j.com/docs/labs/nsmntx/current/importing-ontologies/ (accessed on 23 October 2021).
36. Abburu, S.; Babu, G.S. Survey on Ontology Construction Tools. *Int. J. Sci. Eng. Res.* **2013**, *4*, 1748–1752.
37. Knublauch, H. An AI Tool for the Real World: Knowledge Modeling with Protégé. *JavaWorld*, 20 June 2003. Available online: https://www.infoworld.com/article/2073547/an-ai-tool-for-the-real-world.html?page=2 accessed on 23 October 2021).
38. Available online: https://protege.stanford.edu (accessed on 23 October 2021).
39. Rubin, D.; Knublauch, H.; Fergerson, R.; Dameron, O.; Musen, M. Protégé-OWL: Creating Ontology-Driven Reasoning Applications with the Web Ontology Language. In *AMIA Annual Symposium Proceedings*; American Medical Informatics Association: Rockville, MD, USA, 2005; Volume 2005.
40. Knublauch, H.; Fergerson, R.; Noy, N.; Musen, M. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In *International Semantic Web Conference*; Springer: Berlin/Heidelberg, Germany, 2004; Volume 3298, pp. 229–243.
41. Gherasim, T.; Harzallah, M.; Berio, G.; Kuntz, P. Methods and Tools for Automatic Construction of Ontologies from Textual Resources: A Framework for Comparison and Its Application. In *Advances in Knowledge Discovery and Management—Volume 3 [Best of EGC 2011, Brest, France]*; Studies in Computational Intelligence; Guillet, F., Pinaud, B., Venturini, G., Zighed, D.A., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; Volume 471, pp. 177–201.
42. Bollacker, K.; Evans, C.; Paritosh, P.; Sturge, T.; Taylor, J. Freebase: A collaboratively created graph database for structuring human knowledge. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, 10–12 June 2008; pp. 1247–1250.
43. Available online: https://mailman.stanford.edu/pipermail/protege-owl/2007-September/003823.html (accessed on 23 October 2021)
44. Available online: https://www.w3.org/TR/owl-time/ (accessed on 23 October 2021).