Model-Independent Design of Knowledge Graphs — Lessons Learnt From Complex Financial Graphs[*]

Luigi Bellomarini
Banca d'Italia

Andrea Gentili
Banca d'Italia

Eleonora Laurenza
Banca d'Italia

Emanuel Sallinger
TU Wien and University of Oxford

ABSTRACT

We propose a model-independent design framework for Knowledge Graphs (KGs), capitalizing on our experience in KGs and model management for the roll out of a very large and complex financial KG for the Central Bank of Italy.

KGs have recently garnered increasing attention from industry and are currently exploited in a variety of applications. Many of the common notions of KG share the presence of an extensional component, typically implemented as a graph database storing the enterprise data, and an intensional component, to derive new implicit knowledge in the form of new nodes and new edges.

Our framework, KGModel, is based on a meta-level approach, where the data engineer designs the extensional and the intensional components of the KG—the graph schema and the reasoning rules, respectively—at meta-level. Then, in a model-driven fashion, such high-level specification is translated into schema definitions and reasoning rules that can be deployed into the target database systems and state-of-the-art reasoners. Our framework offers a model-independent visual modeling language, a logic-based language for the intensional component, and a set of new complementary software tools for the translation of meta-level specifications for the target systems. We present the details of KGModel, illustrate the software tools we implemented and show the suitability of the framework for real-world scenarios.

1 INTRODUCTION

While many different notions of Knowledge Graphs (KGs) are offered in the literature [36], the presence of an *extensional component*—often modeled as a property graph [3]—and an *intensional component*, materialized or inferred through a *reasoning process*, clearly emerges, as captured by recent definitions [12]. Intuitively, the extensional component of a KG can be thought of as a graph-based representation of enterprise data. The intensional component, on the other hand, is the specification, for example in the form of reasoning rules, of derived knowledge (e.g., new nodes, edges, attributes thereof), originating from the application of potentially complex domain knowledge to the extensional component, in the so-called *reasoning process* [12].

In Economic and Financial applications, Knowledge Graphs (KGs) are rapidly gaining importance as reference tools to enable knowledge- and data-driven innovation, commonly shared under many names including *FinTech*, *RegTech*, *SupTech*, and *InsurTech* [6, 11, 14, 15, 22, 23, 26, 40, 47, 48, 51]. Underlying all of these areas, there is the need for a detailed representation of complex domains of interest, often characterized by the presence of many interconnected entities, as well as dynamic inference capabilities, to generate derived knowledge and insights that are not immediately available from the enterprise data.

This paper originates from our industrial experience in the design and roll out of Knowledge Graphs for the Central Bank of Italy, specifically dedicated to economic and financial applications. Ours is not an isolated context: thousands of medium and large companies are currently investing in building their enterprise or application-specific KGs and exploiting them to make better business decisions, thanks to their reasoning capabilities.

Challenges. In the construction of such systems, fundamental, recurring questions are always met: how do I understand, design, and communicate how a complex domain (such as the financial one) looks like? How do I eventually structure the KG so that it mirrors the enterprise view of the domain? How can I be sure that the conceptual specification corresponds to the one of the real KG to be deployed? How do I cope with different target systems? How do I express complex business behaviour at high level and cope with heterogeneity of the systems?

We rationalize these questions as the growing need for a *design methodology* for Knowledge Graphs. Such a methodology should guide the data engineer from a conceptual, high-level, and possibly visual representation of the reality to a production-ready KGs. Following the undisputed success of the Entity-Relationship model in the relational world, to be broadly and successfully adopted, a KG design methodology should be easy to grasp, so to help domain understanding and facilitate communication among the stakeholders, and, at the same time, should provide conceptual models that are inherently mappable into the underlying logic data models, to be deployed into the target systems in production environments.

Specifically, we individuate the following distinguishing ***desiderata* of a KG design methodology**.

- *Conceptual ergonomics.* The methodology should provide conceptual data models, enabling a simple, non-technical, high-level, and possibly visual representation of the domain.
- *Model independence.* The methodology should be applicable regardless of the specific technical implementation, that is, it should be possible to deploy the extensional component into any Graph Database Management System, relational, triple-store system, etc., and it should be possible to express the intensional components in a form that is independent of the query languages of the target systems.
- *Model awareness.* The possibility to adopt a more or less compelling graph schema is essential to capture real-world constraints, sustain the development of simple client applications, and ease data manageability [21]. The extensional component should conform to a graph schema and the language for the intensional component should refer to the schema constructs.
- *Expressive, efficient, and ergonomic intensional components.* Intensional components should be first-class citizens in KG design and should be specified with a reasoning language that is at the same time *graph ergonomic* and expressive enough to handle

[*]The views and opinions expressed in this paper are those of the authors and do not necessarily reflect the official policy or position of the Bank of Italy.

KGs. Along the lines of popular regular path query languages such as XPath, XQuery, SPARQL, Neo4J Cypher, etc. reflecting the UC2RPQs literature stream [49], the use of navigational expressions in the language should be seamless and intuitively supported by the syntax.

A good yardstick for the language expressive power is Datalog [24], which captures PTIME with a mild form of negation [27]. The language should be then at least expressive as recursive Datalog, possibly with a form of negation. Derived components should be the result of ontological reasoning processes. Reasoning to the extent of tractable description logic should be feasible (e.g., *DL-Lite*$_R$). In other terms, the language should be expressive enough to cover any SPARQL query over RDF datasets, under the entailment regime of OWL 2 QL [33].

- *Model drivenness*. From a high-level conceptual representation of the domain given by the data engineer, the graph schema for the target system and the executable specification of intensional components should derive as a direct consequence, along the lines of model-driven engineering [30].

Related Work. To the best of our knowledge, in the literature there is no comprehensive methodology for KG design and none of the existing approaches satisfies the illustrated desiderata.

The most comprehensive proposals for *property graphs schema languages* [21] do offer visual modeling metaphors, but are specifically thought for graphs—therefore are model dependent—while for KGs a more general support for the storage of the extensional component is required. Moreover, they do not foresee any intensional component.

Model independence and to some extent model drivenness have been successfully achieved in the *model management community* [18], with proposals for model-independent schema and data translation frameworks such as ModelGen and MIDST [5, 7] by Atzeni et al. where, along the lines of the original observation by Hull and King [37] on the existence of a common ground for data model semantics, a generic *super-model* is proposed, which contains constructs to capture any other data model; translations between models are then expressed via Datalog rules, operating within the super-model itself in a model-independent fashion. Yet, the intensional component is out of ModelGen and MIDST's scope, and these frameworks do not specifically focus on graph-based models but see the relational model as a primary representation, nor do they incorporate any design methodologies. However, as we shall see, we extend and adapt some ideas from MIDST for our purposes. Later adaptations of Atzeni's approach that aim at model-driven and system-independent graph database design [50] are also not applicable for KGs, since they only support graph-based models, do not consider the intensional component and offer a visual metaphor based on a performance-oriented translation of the Entity-Relationship model, which we consider overly low level for our non-technical users.

In the *semantic web community*, the study of ontology engineering was pioneered almost a decade ago [19, 35]. This research line has flourished, with the proposal of pattern-based design techniques (e.g., [2, 20]), as reflected by the rise of dedicated venues and discussion forums [38]. Visualization methodologies (e.g., [25]) have also been developed, in parallel. Our contribution is orthogonal to this line of research, in the sense that the large body of patterns that have been developed for ontology design is of high practical utility for KG design and we do encourage designers to get familiar with and adopt them. However, out of the large body of literature in this area, we could not individuate any methodology able to satisfy all the desiderata, as none of the approaches is either at conceptual level or RDF/OWL-independent, nor does it offer support for a large set of target systems.

Contribution. Our main contributions are:

- We propose KGModel, a model-independent framework for Knowledge Graphs design, comprising a *methodology* and a set of *support tools*.
- KGModel adopts a new KG-oriented version of the MIDST super-model, composed of super-constructs to represent the schemas of the extensional component.
- KGModel is augmented with reasoning programs expressed in MetaLog, a new variant of the Vadalog language for KGs [16], that we propose for the intensional component.
- Our super-model is intimately connected to a visual design methodology, in such a way that the data engineer can design the schemas at a conceptual level.
- Our tools then allow schemas and MetaLog programs to be enforced and executed, respectively, within the target systems, thanks to translation mechanisms.
- We show KGModel in action by sharing our experience in designing financial KGs.

In the next section we present more details of our industrial setting, a general description of KGModel with an overview of the contributions, and a description of the organization of the subsequent sections.

2 OVERVIEW

2.1 The Industrial Setting

The company KG of the Central Bank of Italy is enabling multiple AI-enhanced applications in the economics and supervision realms. The imminent soar of different KGs designed and used by several organizational units of the Bank, as well as more and more complex extensional and intensional components mirroring sophisticated regulatory frameworks, solicit the introduction of a design framework for data engineers, adhering to the illustrated desiderata and, more generally, sustaining a uniform structuring of the enterprise KGs. Let us now introduce the extensional and the intensional component of the KG at hand.

Extensional Component. The source for the enterprise data of the KG is provided by the Italian Chambers of Commerce and contains several features for each company such as legal name, address, incorporation date, legal form, shareholders and so on. Shareholders are either individuals or legal persons, which are typically companies. For persons, we find all the associated register data such as first name, surname, date and place of birth, sex, and so on. For companies, the source data also provide an extensive set of register features. Detailed shareholding structure is also available. For each shareholder, the KG holds the share amount, the kind of legal right associated to it (ownership, bare ownership and so on). All our entities and their associations are time dependent.

If we see the extensional component as a simple shareholding graph—where nodes are shareholders and edges denote owned shares—such graph consists of 11.97M nodes and 14.18M edges. There are 11.96M Strongly Connected Components (SCC), composed on average of one node, and more than 1.3M Weakly Connected Components (WCC), composed on average of 9 nodes. The largest SCC has 1.9k nodes, while the largest WCC has more than six million nodes. The *average in-degree* of each node is ≈ 3.12, the *out-degree* ≈ 1.78 and the *average clustering coefficient* is ≈ 0.0086, the *maximum in-degree* of a node is more than 16.9k

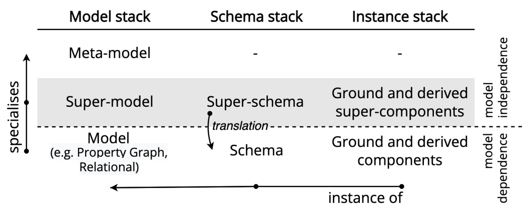


Figure 1: The KGModel modeling stack.

and the *maximum out-degree* is more than 5.1k nodes. As far as the topology is concerned, the graph exhibits a scale-free network structure, as common in the financial domain [10, 41, 45]: the degree distribution follows a power-law, with several nodes in the network acting as hubs.

Intensional Component. Many relationships are implicit and derive from the application of articulated regulatory frameworks to enterprise data. In the Central Bank of Italy KG, an interesting case is the *control* link [32] between companies, which represents the possibility of one company to exert decision power on others and depends on the financial network structure in a complex way; another one is *integrated ownership* [43], which measures the total shares owned by a shareholder, directly and indirectly throughout the whole graph; finally *close links* [42], where the European Central Bank specifies peculiar forms of financial conflict of interest between graph entities involved in the issuance and use as collateral of asset-backed securities. Intensional components are also used to capture relevant phenomena for analysis purposes, such as *company groups*, virtual concepts denoting a center of interest, shared among many firms, or *partnerships* between shareholders sharing the assets of some firm.

2.2 The KGModel Framework

Along the lines of MIDST and as commonly done in popular meta-modeling standards like MOF [46], KGModel adopts a layered approach to data representation, summarized in Figure 1.

The meta-level approach. KGModel is organized into three stacks of representations: *model*, *schema*, and *instance*, where each level contains a set of *constructs* that specialize (or are specialized by) the constructs of the level above (below). The instance stack instantiates the schema stack, which instantiates the model.

In the *model stack*, we adopt the idea of a *super-model* grouping the *super-constructs* that can be used to define different *Knowledge Graph* models, that are all specializations of the super-model. Possible models used to represent KGs are the different versions of *property graphs* [3] (PG) such as the models of Neo4J PG, Amazon Neptune, OrientDB or even non-graph-like models that are frequently used to serialize graphs, such as the relational data model, plain CSV files, and so on. Let us discuss our model stack.

- At the highest level we have a ***meta-model***, which contains the foundational *meta-constructs*, namely, MM_Entity (an abstract entity of the domain), MM_Link (a connection between entities), as well as their properties.
- The ***super-model*** contains *super-constructs* that specialize those of the meta-model and subsumes, i.e., generalizes, any possible KG model. Examples of super-constructs are SM_Node, SM_Edge, representing the general notions of node and edge, respectively, SM_Attribute for their attributes, and so on.
- The various ***models*** comprise *constructs* that specialize the super-constructs for a specific use. Example of PG constructs are Node, Relationship, and Label, instantiating SM_Node, SM_Relationship, and SM_Type, respectively.

In the *schema stack*, *schemas* capture the type of specific nodes, edges and properties of a given domain of interest, in the same sense that a relational database schema is an instance of the relational model. For example, a graph schema is the one of the companies, with its business entities such as people, shares, locations, and so on. As the super-model generalizes every model, a schema can be expressed in either a model-dependent way, as an instance of a model, or in a model-independent way, as an instance of the super-model, in which case we call it *super-schema*. A super-schema S_1 can be cast into a schema S_2 of a model M by a specific set of translation rules, namely, *mappings*, which apply the needed simplifications, when the case eliminating constructs of the super-model that are not supported by the specific target model, and finally instantiate the super-constructs into M constructs, accordingly. KGModel stores super-schemas and schemas into *graph dictionaries*, associated to the super-model and to each of the models, respectively. We define the mappings as sets of MetaLog rules. MetaLog is our new variant of the Vadalog language [16] for graphs. Vadalog extends Datalog with existential quantification and other useful features, while introducing mild syntactic restrictions to guarantee decidability and tractability of the reasoning task. Vadalog reasoning programs can be processed by the Vadalog System, a state-of-the-art reasoner, able to read ground data from heterogeneous data sources. MetaLog inherits the Vadalog (and thus Datalog) semantics and expressive power, enriching its syntax with the possibility to use pattern-matching graph exploration primitives. It is model independent, as it operates at meta-level, expressive and efficient enough to support ontological reasoning, model aware and ergonomic, since incorporates syntactic elements to closely exploit schema information.

The *instance stack* concretely represents the **extensional component** of the KG, i.e., both the ground data and those derived by materializing the intensional component (Section 6). The extensional component can be represented as either instance of the super-schema (*super-components*) or of the schema (*components*).

The design methodology. With KGModel, we offer the data engineer a model-driven design approach to KGs. The data engineer is provided with a **conceptual visual modeling language**, named *Graph Schema Language* (GSL) to design a graph schema as a super-schema. A GSL diagram defines an instance of the super-model, with *visual graphemes* denoting instances of the super-constructs. She also specifies the intensional component (possibly comprising further schema constraints) in MetaLog, with reasoning programs acting on the super-model constructs.

In order to deploy the designed schemas into the target systems, KGModel translates the super-schemas provided by the engineer into instances of the target models by applying the translation mappings. Schemas then contain all the information needed to be deployed and enforced, with different methods, depending on the target systems: for relational systems, for instance, they can be rendered as DDL statements, which include the respective constraints such as keys, foreign keys, domain constraints, and so on; for RDF stores, schemas can be rendered as RDF-S (RDF Schema) documents, to be validated by dedicated tools; for schema-less systems, like graph databases, schemas can be enforced with ad-hoc methodologies [21].

The data engineer also specifies the **intensional component** of the KG in MetaLog at super-model level. It is then translated into Vadalog and applied in a system-independent fashion.

The software modules. To support the explained methodology, our framework incorporates the following tools:

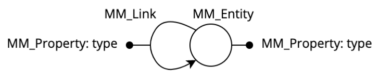


Figure 2: The Meta-model.

- **Graph Dictionaries**. A set of graph databases to store the instances of the super-model and of the models.
- **(KGSE) Knowledge Graph Schema Environment**. A tool to graphically design GSL schemas and store them into the super-model dictionary.
- **(MTV) MetaLog to Vadalog Translator**. A compiler to generate Vadalog programs from MetaLog code.
- **(SSST) Super-Schema to Schema Translator**. A module that takes as input a super-schema S, a super-model-level intensional component Σ expressed as a set of MetaLog rules, a MetaLog mapping $\mathcal{M}(M)$ for the translation of a super-schema into a schema of the target model M, and generates: (i) the instance S' of M, that is, the desired target schema; (ii) a new version of the intensional component that can be applied to S' instances. SSST uses MTV to compile and run MetaLog.

2.3 Organization of the Paper

In Section 3 we describe the meta-model, the super-model, and our visual design methodology. In Section 4 we introduce the MetaLog language. Section 5 shows how different models for KGs can be represented in our framework. A focus on the intensional component is introduced in Section 6. Our industrial case is pursued throughout all the sections, showing the conceptual design of the KG of the Bank of Italy, the applied MetaLog code, the generation of the model-level representations, important portions of the intensional component, respectively. Section 7 concludes the paper.

3 THE *META-LEVEL* APPROACH

As we have pointed out in the overview, the starting point of our approach is a model-generic representation of schemas, which we call super-schemas. To do that, we define models as sets of constructs, each specializing a super-construct of a general super-model. The super-constructs, in turn, specialize a small set of foundational meta-constructs. KGModel then features a visual modeling language to define super-schemas.

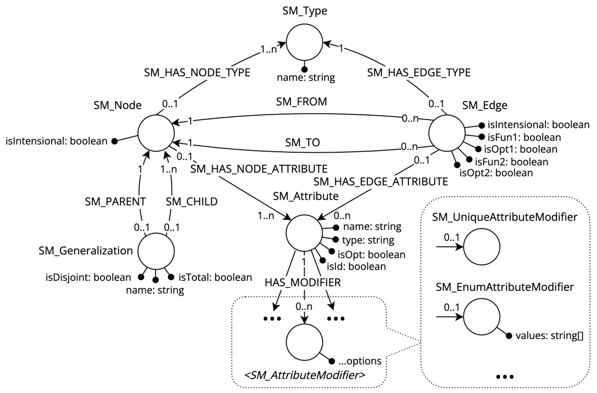
3.1 A Look Form Above: the *Meta-model*

At the highest level of our model representation stack, we find the meta-model, comprising the basic building blocks of any semantic data model: entities, links between them, and their properties. Figure 2 visualizes it as a PG: nodes are denoted by labeled circles, edges by labeled arrows with cardinality indication (like in UML), and node and edge attributes with the lollipop notation.[1]

Each meta-construct is identified by a unique internal *Object Identifier* (OID). MM_Entities are abstract named domain objects. MM_Properties have a name and a type. MM_Links express relationships $A \longrightarrow B$ between entities.

The meta-model lays the foundations for the visual representation of super-schemas. We do that by introducing an *instance rendering* function Γ_M, a bijection that specifies how to visualize the instances of a model M. For the meta-model, the rendering function $\Gamma_{\text{MM}} : C_{\text{MM}} \longrightarrow \mathcal{V}$ maps an instance $c \in C_{\text{MM}}$ of a meta-construct (C_{MM} is the set of all possible meta-construct instances) into a *grapheme* (an elementary graphic item) $g \in \mathcal{V}$, where $\mathcal{V}$, is the alphabet of graphemes. For example, entity instances are represented as name-labeled nodes, properties with the named lollipop notation; links are represented as name-labeled edges.

[1]We will use the PascalCase convention for entity names, UPPER_CASE for links and camelCase for propeties in all the stack of KGModel.



Super-construct	Attributes	Grapheme
SM_Node	isIntensional = false, name from SM_Type	○ name
SM_Node	isIntensional = true, name from SM_Type	◌ name
SM_Edge	isIntensional = false, name from SM_Type, $c1_{min}$, $c1max$, $c2_{min}$, $c2max$ from isOpt and isFun	name ─── $c2_{min}..c2_{max}$ $c1_{min}..c1_{max}$
SM_Edge	isIntensional = true, name from SM_Type, $c1_{min}$, $c1max$, $c2_{min}$, $c2max$ from isOpt and isFun	name - - - $c2_{min}..c2_{max}$ $c1_{min}..c1_{max}$
SM_Type	name	name
SM_HAS_NODE_PROPERTY	▨	▨
SM_HAS_EDGE_PROPERTY	▨	▨
SM_FROM	▨	▨
SM_TO	▨	➤
SM_Attribute	isOpt = false, isId = false	─●
SM_Attribute	isOpt = true, isId = false	─○
SM_Attribute	isOpt = false, isId = true	●
SM_HAS_NODE_PROPERTY	isIntensional = false	───
SM_HAS_EDGE_PROPERTY	isIntensional = true	- - - - -
SM_Generalization	isTotal = true, isDisjoint = true	➡
SM_Generalization	isTotal = false, isDisjoint = true	➡
SM_Generalization	isTotal = true, isDisjoint = false	⇨
SM_Generalization	isTotal = false, isDisjoint = false	⇨
SM_PARENT	▨	▨
SM_CHILD	▨	▨

Figure 3: The super-model dictionary and the tabular representation of the rendering function Γ_{SM} for its instances.

3.2 The Designer Level: the *Super-model*

The super-model provides the data engineer with a collection of model-independent conceptual elements: the super-constructs. Then, along the lines of model-driven software design [28], we encourage the data engineer to craft the extensional component of the KG by performing a *domain-based decomposition* of the reality. She should single out the core concepts of the domain at hand. For each concept, then she should individuate the most "semantically adequate" super-construct and instantiate it to capture the domain object. In so doing, she assembles instances of super-constructs, building a super-schema.

The semantic adequacy refers to the semantic category a real-world object belongs to. When designing a relational database with the Entity-Relationship model, for instance, the data engineer wonders whether a concept is an entity or an attribute based on its independent dignity and existence in the domain; she tries to distinguish entities from relationships, based on their occurrence contexts, and so on. Similar considerations are applied in every conceptual model to the extent that the design techniques develop the engineer's sensitivity to seizing the meaning of the conceptual model constructs and mapping real-world concepts into them. In this sense, a design methodology is tightly coupled with the semantics of the constructs. In KGModel, we take this consideration to a higher level and suggest that the data engineer work with super-model constructs and design super-schemas. We shall see in Section 5, how the super-schemas are finally cast into schemas, once a target model is selected.

As our framework targets graph-based applications, we propose a set of super-constructs that see PGs as first-class citizens. However, they are at the same time sufficiently general to capture many other target data models, as proven by the straightforward correspondences to other relational-based super-models [8].

Figure 3 renders our super-model dictionary, whose constructs are instances of the meta-model, by applying Γ_{MM}. To allow the data engineer to visually create instances of the super-model in design diagrams, we introduce the rendering function $\Gamma_{\text{SM}} : \mathcal{C}_{\text{SM}} \rightarrow \mathcal{V}$, that is detailed in tabular form in the Figure. The Graph Schema Language is the visual language for KG design diagrams originating from the application of Γ_{SM} and it includes specific notation to use the super-constructs; for example, we mark a partial disjoint generalization with a single-headed thick solid black arrow. While most of the super-construct have an explicit notation, there are some others which do not have it, like SM_HAS_CHILD; these super-constructs are marked with a gray background in the table.

Designing with the Super-constructs. We provide the core design guidelines by describing the KGModel super-constructs.

- SM_Node is the general notion of entity. It should be used to represent any relevant domain object that is characterized by its own identity, SM_Type, and set of distinguishing properties. An SM_Node always has one single identifier, composed of a set of identifying attributes.
- SM_Edge represents a binary aggregation of two SM_Nodes. It should be used to capture the existence of a relationship between domain concepts. Cardinalities are encoded as follows: isFun1 (resp. isFun2) is true if right (left) maximum cardinality is 1, it is N, otherwise; isOpt1 (resp. isOpt2) is true if right (left) minimum cardinality is 0, it is 1, otherwise. Note that, unlike SM_Nodes, as SM_Edges have one single SM_Type, super-schemas are *simple graphs* by construction.
- SM_Attribute models an attribute of a node or edge. It should be used for any relevant domain object that does not have its own identity, but is part of a more general concept. It can be optional (isOpt) or mandatory, identifying (isId) or not.
- SM_AttributeModifier, marked in italic in the figure, is a proxy for attribute modifiers that are generally used to enrich an attribute with additional information, such as formatting or domain constraints; each modifier has a corresponding *super-construct* which holds specific information. For example, the SM_UniqueAttributeModifier prescribes that an attribute has a unique value among nodes with the same SM_Type, or the SM_EnumAttributeModifier lists all the values an attribute may have. As a whole, KGModel defines many more modifiers, which help the data engineer to explicitly model business constraints. We will not focus on this super-construct in the rest of the paper, for space reasons.
- SM_Generalization should be used to capture the usual notion of specialization-abstraction relationship existing between entities, SM_Nodes in this case. It can be further characterized as *total* if every instance of the parent is also an instance of a child (non-total otherwise), *disjoint* if the instances of the parent are instances of a single child (non-disjoint otherwise).

Other super-constructs are the ones specializing MM_Link: SM_HAS_NODE_TYPE, SM_HAS_EDGE_TYPE, SM_PARENT,

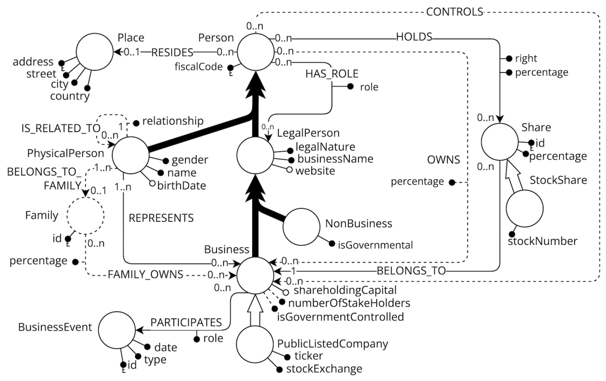


Figure 4: A portion of the Bank of Italy KG designed with KGModel methodology.

SM_CHILD, SM_FROM, SM_TO, SM_HAS_PROPERTY, and SM-_HAS_MODIFIER. Their use is straightforward, as they connect the respective SM_Node specializations.

3.3 A View on the Central Bank of Italy KG

We used KGModel for the conceptual design of our KG, culminating in the construction of a GSL diagram. A simplified but representative portion of such diagram, which we generally refer to as the "Company KG", is shown in Figure 4.

Throughout the conceptual design work, we could appreciate how the existence of a technology-independent super-model dictionary guides the designer through her modeling activity, offering a toolkit of lenses to capture real-world objects, understand their characteristics and relationships, and communicate the design choices with stakeholders. Let us try to narratively simulate a fragment of such a modeling journey.

The domain of the Company KG revolves around the notions of physical persons, i.e., individuals, or legal persons, which are entities capable of performing some actions like owning properties. Even if these two entities share some features like the way in which they interact with the world (e.g., they can buy or sell stocks of a company), they are different since they have different attributes (e.g., an individual has a gender, while a legal person has a legal nature, and so on).

«I will capture the structure by introducing distinct SM_Nodes for persons, i.e., PhysicalPerson and LegalPerson, characterized by a distinct set of SM_Attributes. As for the attributes, PhysicalPersons are identified by a unique fiscalCode and have a gender and a name; the birthDate is not always present in data, and as such it is optional; LegalPersons are as well identified by their fiscalCode, have a businessName and a legalNature and may have a website.»

Physical persons and legal persons can have their place for residence. They can hold stakes in a company shareholding capital. Moreover, both have a fiscal code, which uniquely identifies them in the national system. The fact that physical and legal persons share common traits and nevertheless have their specificities, suggests that they are two specializations of a generic actor.

«I will introduce a SM_Generalization, where a Person generalizes and collects the common features of PhysicalPerson and LegalPerson. As every person can be in exactly one of those two categories, the generalization will be disjoint and total.»

Let us focus on the residence. When available, the address is typically complex and composed of multiple parts (e.g. street,

street number, city, postal code). In the future, it can be enriched with further elements, such as the GPS coordinates.
💡*«Though tempting, modeling the address as an optional Person attribute is not a valid choice, which may lead to increased complexity: the address is an autonomous business entity, it has many details, is likely to change in the future and be enriched. I will introduce a Place SM_Node, modeling the address as an identifier and storing each part of it as an SM_Attribute.»*

The ownership structure of shareholding capital is available only for some specific legal persons, i.e., businesses. Moreover, some of these businesses are public companies whose capital is listed in stock exchanges. Other entities have different forms of legal nature for which shareholding capital does not make sense; at the same time, the analysts are interested in legal persons that are expression of the public sector in the economic system, such as territorial entities.
💡*«I will introduce a further SM_Generalization by specializing the LegalPerson into a Business SM_Node, gathering shareholding capital features, and a NonBusiness SM_Node, with specific isGovernmental SM_Attribute. As a legal person can be in exactly one of those two categories, the generalization will be disjoint and total. I will add one more specialization of Business by creating a child PublicListedCompany SM_Node hosting specific information about the stock exchange features. As a business can be publicly listed or not, the generalization will not be total.»*

Our data show that a person can withhold stakes in a company capital, and there are cases in which multiple distinct persons may have different rights upon the same portion of company capital (e.g., multiple owners of a single share or complex property structures like usufruct, in which bare owner and usufructuary have different rights on the property). Nevertheless, data analysis constantly requires insight about standard ownership.
💡*« While property would be elegantly described by a simple OWNS SM_Edge connecting the owning Person to the owned Business, this would not allow for multiple Persons holding a single stake in the company. Hence, I will introduce a Share SM_Node (which represents a portion of the business capital) and the HOLDS-BELONGS_TO SM_Edges decoupling owner-owned SM_Nodes so that multiple Persons can HOLD a Share each with a specific right and percentage. To ease the life of the analyst, I will introduce an intensional OWNS SM_Edge that compactly represents only property rights, hence connecting the same owner/owned SM_Nodes. I will introduce as well a numberOfStakeholders intensional property into Business SM_Node to make it available for analysis.»*

Our analysts are interested in understanding whether the ownership of a business can end up in holding, directly or indirectly, the majority of its stakes, thus controlling it.
💡*«I will introduce an intensional CONTROLS SM_Edge connecting the controlling Person to the controlled Business.»*

A share is characterized by a percentage indication of its proportion to capital total; when the capital is traded as stocks, the information of the number of stocks is available as well.
💡*«Having specific characteristics, I will model stock shares as a specialization of Share SM_Node, namely StockShare. As a share can be a stock share or not, the generalization will not be total.»*

While people and companies are clearly distinct notions in the domain, they are likely to participate in the same relationships, such as ownership or control ties which they can exert on other companies, or playing a role e.g., on the board of directors. Thus, there is a form of graph homogeneity, which we would like to pursue, where relevant entities (be they individuals or companies) are linked by shareholding relationships. At the same time only individuals belong to families and can represent businesses, and only businesses have a capital held by shareholders or participate in business events such as mergers.
💡*«I will leverage generalization and model relationships only between the topmost nodes in the generalization hierarchy which are involved in them. Hence, I will identify the following relationships: HOLDS, OWNS, CONTROLS, HAS_ROLE, RESIDES, that can be exerted by any Person; REPRESENTS that can be exerted only by PhysicalPerson nodes; PARTICIPATES, that can only be exerted by Businesses. As the participation in shareholding capital can only be exerted on entities having shareholdingCapital, the OWNS, CONTROLS, BELONGS_TO SM_Edges will be inbound to Business SM_Nodes; finally, a Person can have a role in NonBusinesses and Businesses, but not in PhysicalPersons, so the HAS_ROLE SM_Edge will be inbound to LegalPerson SM_Node.»*

Particularly relevant research objectives related to the use of our KG are based on the connections between shareholders, for instance family relationships.
💡*«Since these connections can be individuated by specific reasoning tasks, I will add an intensional IS_RELATED_TO SM_Edge connecting two individuals, i.e., connecting the PhysicalPerson SM_Node to itself in the super-schema. In turn, each PhysicalPerson has an intensional BELONGS_TO_FAMILY SM_Edge connecting it to a Family SM_Node. Since a family can hold a business, it has an intensional FAMILY_OWNS SM_Edge to Businesses.»*

Finally, a data source reports information about company events like merger & acquisitions or splits; these events happen on a date and may generally involve multiple businesses, e.g., the acquirer and the acquired.
💡*«As these events involve only companies, the topmost entity in the generalization to which to refer is the Business SM_Node. I will add a new BusinessEvent SM_Node, characterized by type and date SM_Attributes, in which each business can participate through a PARTICIPATES SM_Edge with a specific role.»*

4 THE METALOG LANGUAGE

We now introduce MetaLog, the language we propose and use in KGModel. To strike a balance between the ergonomicity of regular path queries and the expressive power of modern ontological reasoning formalisms, MetaLog combines *Warded Datalog*$^{\pm}$, a logic language at the core of Vadalog that proved to be of high industrial applicability, and graph pattern matching.

Example 4.1. The following MetaLog program expresses company control. This notion is an essential ingredient for the intensional component of the Bank of Italy KG: control edges pinpoint when a company can exert decision power on another one.
A business x controls a business y, if: (i) x directly owns more than 50% of y; or, (ii) x controls a set of companies that jointly (i.e., summing the share amounts), and possibly together with x, own more than 50% of y.

$$(x : \mathit{Business}) \rightarrow \exists c\ (x)[c : \mathit{CONTROLS}](x) \qquad (1)$$

$$\begin{aligned}(x : \mathit{Business})[: \mathit{CONTROLS}](z : \mathit{Business})\\ [: \mathit{OWNS}; \mathit{percentage} : w](y : \mathit{Business}),\\ v = \mathit{sum}(w, \langle z \rangle), v > 0.5 \rightarrow \exists c\ (x)[c : \mathit{CONTROLS}](y)\end{aligned} \qquad (2)$$

Every company x controls itself (1), and (2), whenever x controls a set of companies z such that the sum of their shares w over one single company y is more than 50%, then x controls y. ■

In terms of expressive power, MetaLog is the union of all programs in *Non-Recursive Warded Datalog*$^\pm$ extended with transitive closure of binary relations, and all the Vadalog programs. To introduce the language, we recall the foundations of logic reasoning with Vadalog [13], the standard notion of property graph [3], and finally present MetaLog syntax and semantics.

Relational Foundations and Vadalog. Let **C**, **N**, and **V** be disjoint countably infinite sets of *constants*, *(labeled) nulls* and (regular) *variables*, respectively. A *(relational) schema* **S** is a finite set of relation symbols (or predicates) with associated arity. A *term* is either a constant or variable. An *atom* over **S** is an expression of the form $R(\bar{v})$, where $R \in \mathbf{S}$ is of arity $n > 0$ and $\bar{v}$ is an n-tuple of terms. A *(database) instance* over **S** associates to each relation symbol in **S** a relation of the respective arity over the domain of constants and nulls. The members of relations are called *tuples*. By some abuse of notations, we sometimes use the terms tuple and fact interchangeably.

A rule is a first-order sentence of the form $\forall\bar{x}\forall\bar{y}(\boldsymbol{\varphi}(\bar{x},\bar{y}) \rightarrow \exists\bar{z}\,\boldsymbol{\psi}(\bar{x},\bar{z}))$, where $\boldsymbol{\varphi}$ (the *body*) and $\boldsymbol{\psi}$ (the *head*) are conjunctions of atoms with constants and variables. We typically omit universal quantifiers and replace $\wedge$ with comma for conjunctions of atoms. A program Σ is then defined as a set of rules.

The semantics of (a set of) existential rules can be intuitively explained as follows: for each fact $\varphi(\bar{t},\bar{t}')$ of an instance D, then there exists a tuple $\bar{t}''$ of constants and fresh labeled nulls such that the facts $\psi(\bar{t},\bar{t}'')$ are also in I. The semantics is more formally defined via *chase-based procedures* [1]. The chase alters D by adding new facts, possibly with fresh labeled nulls for existentially quantified variables, until $\Sigma(D)$ satisfies all the existential rules of Σ. It is well-known that in the presence of general recursion and existential quantification, the *reasoning task*, which intuitively amounts to answering queries over $\Sigma(D)$, is undecidable [34]. Wardedness poses syntactical restrictions on the interplay of existential quantification and recursion, so that the reasoning task remains decidable and PTIME in data complexity, i.e., when the program is fixed and the data are made to vary.

Example 4.2. In Vadalog, company control can be encoded with the following existential rules.

$$Company(x) \rightarrow CONTROLS(x,x) \qquad (1)$$

$$CONTROLS(x,z), Own(z,y,w), v = sum(w,\langle z\rangle), v > 0.5 \rightarrow CONTROLS(x,y) \qquad (2)$$

Property Graphs. A (regular) *Property Graph* (PG) is a tuple of the form $G = (N, E, \mu, \lambda, \sigma)$, where: N is a finite set of nodes; E (disjoint from N) is a finite set of edges; the *incidence function* $\mu : E \rightarrow N^n$ is a total function that associates each edge in E with an n-tuple of nodes from N (n=2, for our goals); the *labelling function* $\lambda : (N \cup E) \rightarrow \mathbf{L}$ is a partial function that associates nodes/edges with a label from a set **L**; the function $\sigma : (N \cup E) \times \mathbf{P} \rightarrow \mathbf{V}$ is a partial function that associates nodes/edges with properties from **P** to a value from **C** for each property.

MetaLog. Let **C**, **N**, and **V** be disjoint countably infinite sets of *constants*, *(labeled) nulls* and (regular) *variables*, respectively, and $G = (N, E, \mu, \lambda, \sigma)$ be a property graph. A MetaLog program is a set Σ of existential rules $\boldsymbol{\varphi}(\bar{x},\bar{y}) \rightarrow \exists\bar{z}\,\boldsymbol{\psi}(\bar{x},\bar{z})$, where $\bar{x}$, $\bar{y}$, and $\bar{z}$ are tuples of variables, $\boldsymbol{\varphi}$ is a conjunction of *PG node atoms*, *path patterns*, *conditions*, and *expressions* and $\boldsymbol{\psi}$ is a conjunction of *PG node atoms* and *path patterns*.

- A *PG atom* can have two forms: $(x : L;\ \bar{K})$ (*PG node atom*), or $[x : L;\ \bar{K}]$ (*PG edge atom*), where $x \in \mathbf{V}$ is the atom identifier, $L \in \mathbf{L}$ is a label, $\bar{K}$ is a tuple of named terms $A_i : x_i$, where $A_i \in names(\bar{K})$ are attribute names in **P** and $x_i \in \mathrm{values}(\bar{K})$ are variables of **V** or constants of **C**. The atom identifier and the named variables can be omitted if the anonymous binding is intended. An example of PG atom is $(x$: PhysicalPerson; name : n, gender : "male"), which selects all the PhysicalPerson labeled nodes x having male gender and binds their name to the variable n. An example of PG edge atom is $[o$: HOLDS; right : "ownership", percentage : $s]$, which selects all the ownership rights, binding the share amount to the variable s.
- An *expression* is an assignment of the form $z_i = f(\bar{x},\bar{y})$, where $z_i \in \bar{z}$, and f is a generic function, which may be *tuple-level* (e.g., an algebraic operation, a string operation, and so on) or *multi-tuple* (e.g., an aggregation), where the standard stratified semantics is assumed [39]. For instance $t = \mathrm{sum}(w,\langle y\rangle)$ aggregates w over y and returns t.
- A *condition* is a Boolean expression over a variable in $\bar{x}$ or $\bar{y}$, for example: *right="ownership" and percentage<0.3.*
- Let $\mathcal{A}$ be the alphabet of elements ρ corresponding to PG edge atoms. An expression $x\rho y$ denotes the existence of a binary relation between the nodes selected by the PG atoms x and y, and by the PG edge atom ρ. Let us introduce the *inverse* operator $^-$: ρ^- denotes the inverse binary relation, i.e., if ρ is p, then ρ^- is p^- and if ρ^- is p, then ρ is p^-. A *path pattern* is expressed by means of regular expressions R over the alphabet $\mathcal{A}$. A path pattern xRy individuates all the pairs of nodes $\langle x, y\rangle$ connected by a semi-path that conforms to the regular language $L(R)$ defined by R. A semi-path from x to y is a sequence of the form $(n_1, e_1, n_2, e_2, \ldots, e_q, n_{q+1})$, where $q \geq 0$, $n_1 = x$, $n_{q+1} = y$, and for each (n_i, e_i, n_{i+1}), we have that either $\mu(e_i) = \langle n_i, n_{i+1}\rangle$ or $\mu(e_i) = \langle n_{i+1}, n_i\rangle$. A semi-path defined by an expression R conforms to a path pattern if $e_1, \ldots, e_q \in L(R)$.

The semantics of MetaLog descends from the Vadalog one in the natural way. For each fact of $\varphi(\bar{t},\bar{t}')$ of G, that is, a conjunction of paths of G, there exists a tuple $\bar{t}''$ of constants from **C** and fresh labeled nulls from **N**, such that the paths $\psi(\bar{t},\bar{t}'')$ are also in G. Given a set Σ of MetaLog rules, the chase alters G by adding new paths, until $\Sigma(G)$ satisfies all of them.

To guarantee decidability and tractability of the reasoning task we require that: *the transitive closure of relations in path patterns, via the Kleene star operator, is allowed only if the program Σ is non-recursive, i.e., the dependency graph of rules is acyclic.* If Σ does not include the transitive closure, then it can be reduced into a warded program and therefore the reasoning task is decidable and PTIME; if Σ includes the transitive closure, as it is non-recursive, it can be reduced into a *Piecewise Linear Datalog*$^\pm$ [17], a subset of Warded Datalog$^\pm$.

MetaLog features syntactic elements to natively refer to nodes and edges. As in KGModel the super-model is stored in a graph dictionary, then MetaLog can be used to operate on it.

Example 4.3. In a super-schema, we link all the pairs of SM_Nodes that are in descendant-ancestor relationship, at any level.

$$(x : SM\_Node)([: SM\_CHILD]^- \cdot [: SM\_PARENT])*$$
$$(y : SM\_Node) \rightarrow \exists w\ (x)[w : DESCFROM](y)$$

A more complex regular expression is used instead of recursion. The dot $(\cdot)$ notation represents concatenation in regular expressions. ■

MetaLog and Vadalog. The MTV component of KGModel performs a MetaLog to Vadalog translation, enabling its execution. For each rule, the translation consists in replacing the PG atoms with relational atoms, as used in Vadalog. We have three phases: (1) the input PG instance G is translated into a database instance D; (2) the PG node atoms of the MetaLog rules are mapped into relational atoms; (3) the path patterns are resolved. The translation of conditions and expressions is straightforward.

(1) *PG-to-relational mapping. The input PG instance G is translated into a relational database instance D as follows.*
   - L-labeled nodes $n \in N$ (with $L_n \in \mathbf{L}$) are translated into facts $L(\hat{c}_x, c_f^1, c_f^2, \ldots, c_f^n)$ of predicate L, where for each property $f_i \in \mathbf{P}$, we have a constant term c_f^i of L of value $\sigma(n, f_i)$. Note that we assume every node has an internal OID x, whose value $\hat{c}_x = \sigma(n, x)$ identifies its facts.
   - L_e-labeled edges $e \in E$, are translated into facts L_e: $L_e(\hat{c}_x, \hat{c}_x^1, \ldots, \hat{c}_x^k, f_1, \ldots, f_m)$, where for each argument i of the function μ, there is a constant $\hat{c}_x^i$ of L_e with value $\sigma(n, x)$, where $n = \mu(e)[i]$ and x is the identifier of n, and for each feature $f_i \in \mathbf{P}$ of e there is a constant c_f^i of L_e with value $\sigma(e, f_i)$. We assume the presence of an internal OID x, whose value $\hat{c}_x = \sigma(e, x)$ identifies the edge facts.

(2) *PG node atoms to atoms translation.* PG node atoms of the MetaLog program $(x : L;\ \bar{K})$ are translated into $L(x, x_i, \ldots, x_n)$, where $x_i \in \text{values}(\bar{K})$ are constants of $\mathbf{C}$ or variables of $\mathbf{V}$.

(3) *Resolution of path patterns.* After step (2), graph patterns have the form $L(x, x_1, \ldots, x_n)\ R_{xy}\ M(y, y_1, \ldots, y_m)$, where L, M are predicate names mapping labels in $\mathbf{L}$ and R_{xy} is a regular expression over $\mathcal{A}$, that defines paths from x to y. The translation $\tau(R_{xy})$ of R_{xy} is inductively defined as follows:
   - $R_{xy} = [z : T\ \bar{K}] \Rightarrow \tau(R_{xy}) = T(z, x, y, values(\bar{K}))$
   - $R_{xy} = (S_{xy} | T_{xy}) \Rightarrow \tau(R_{xy}) = \alpha(x, y, \bar{z})$, where α is a new atom defined by the following new Vadalog rules: (i) $\tau(S_{xy}) \rightarrow \alpha(x, y, \bar{z})$; (ii) $\tau(T_{xy}) \rightarrow \alpha(x, y, \bar{z})$, where $\bar{z}$ is the tuple of body variables except x and y.
   - $R_{xy} = S_{xq} \cdot S_{qm} \ldots \cdot S_{nv} \cdot T_{vy} \Rightarrow \tau(R_{xy}) = \tau(S_{xq}), \ldots, \tau(S_{vy})$
   - $R_{xy} = (S_{xy})^{-} \Rightarrow \tau(R_{xy}) = \tau(S_{yx})$
   - $R_{xy} = (S_{hq})* \Rightarrow \tau(R_{xy}) = \beta(x, y, \bar{z})$, where β is a new atom defined by the following new Vadalog rules: (i) $\tau(S_{hq}) \rightarrow \beta(h, q, \bar{z})$; (ii) $\beta(v, h, \_), \tau(S_{hq}) \rightarrow \beta(v, q, \bar{z})$, where $\bar{z}$ is the tuple of variables of $vars(S_{hq}) \setminus \{h, q\}$.

In Vadalog, the atoms deriving from MetaLog PG node and edge atoms are populated from the input sources via automatically generated *annotations* of the form `@input(atom,query)`, where `atom` is the relational atom name and `query` is expressed in the target system language—so, e.g., SQL for relational systems, or Cypher in Neo4J— and implements the translation step (1).

Example 4.4. What follows is the Vadalog translation of the MetaLog program in Example 4.3.

$$SM\_Node(x, \ldots), \beta(x, y, \ldots), SM\_Node(y, \ldots) \rightarrow \exists w\ DESCFROM(w, x, y) \quad (1)$$

$$SM\_CHILD(\_, u, x, \ldots), SM\_PARENT(\_, u, y, \ldots) \rightarrow \beta(x, y, \ldots) \quad (2)$$

$$\beta(v, h, \ldots), SM\_CHILD(\_, u, h, \ldots), SM\_PARENT(\_, u, q, \ldots) \rightarrow \beta(v, q, \ldots) \quad (3)$$

```
@input(SM_Node,"(n:SM_Node) return n").
@input(SM_PARENT,"(n:SM_Node)-[p:SM_PARENT]->
(g:SM_Generalization) return (p,g,n)").
@input(SM_CHILD,"(n:SM_Node)<-[c:SM_CHILD]-
(g:SM_Generalization) return (c,g,n)").
```

The original rule contains inversion, Kleene star and concatenation operators. The annotations exemplify Neo4J extraction. ■

Linker Skolem Functors. Sometimes in KGModel we will need an OID generation/retrieval mechanism that is more controlled than standard null generation. For this purpose, let us introduce a new set of symbols $\mathbf{I}$, having empty intersection with $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{V}$. A MetaLog rule can be of the form $\boldsymbol{\varphi}(\bar{x}, \bar{y}) \rightarrow \exists \bar{z}\, \exists k_{sk(\bar{v})} \boldsymbol{\psi}(\bar{x}, \bar{z}, k)$, where one or more existentially quantified variables k are associated to a *linker Skolem functor sk*, with the following semantics: whenever the rule is applied, the chase generates for k a fresh value from $\mathbf{I}$. Such value is computed by applying the functor sk to a tuple of universally quantified variables $\bar{v} \subseteq \bar{x} \cup \bar{y}$. Skolem functions are assumed to be injective, deterministic and range disjoint, i.e., for every pair of functors $sk^A \neq sk^B$, their images do not overlap.

5 MODEL LEVEL

By crafting a GSL design diagram, the data engineer has specified the structure of the KG extensional component within a super-schema. Such object is still at conceptual and super-model level. To be operated and hence enforced in the target systems, first the KGSE serializes and stores the visual diagram into the super-model dictionary. Then, the SSST translates it into a schema of the target logical model (Section 5.1). The fully specified schema can be then enforced with well-known techniques depending on the specific model and technology. This is out of scope for this paper and we have already referred to some schema validation techniques in Section 2.

5.1 Super-schema to Schema Translation

A model is represented in KGModel by specializing and renaming a subset of the super-constructs. Figures 5 and 7, for instance, represent the PG model and the relational model, respectively, and highlight which super-constructs are used.

The translation of a super-schema S, instance of the super-model, into a schema S', instance of a target model M, is performed as described in Algorithm 1.

Algorithm 1 *SSST Schema Translation Algorithm.*
Input: super-schema S, target model M; *Output:* schema S' of M.

```
1: $\mathbf{M} \leftarrow$ select candidate mappings to $M$ from REPO
2: $\mathcal{M}(M) \leftarrow$ prompt for implementation strategy
3: $\mathcal{V}(M) \leftarrow$ MTV.$translateToVadalog(\mathcal{M}(M))$
4: $S^{-} \leftarrow Reason(S, \mathcal{M}(M)$.Eliminate$)$
5: $S' \leftarrow Reason(S^{-}, \mathcal{M}(M)$.Copy$)$
```

After individuating a set of candidate mappings for M from a rule *repository* (line 1), the system involves the data engineer (line 2) who refines the choice on the basis of the desired *implementation strategy* for the super-schema. Implementation strategies encode the different modeling choices for the translation into a given model M. They can be driven by performance considerations, as typical in the design of NoSQL databases [44] and logical design of relational databases [9], or reflect different tactics to implement conceptual notions of the super-model with the features offered by the target systems: for PGs, whether SM_Generalization should be implemented via child-parent edges or node tagging is an example of different tactics, highly depending on the target model and system which may allow multi-tagging or not. Observe that, however, the data engineer is not responsible for the design of the mappings, and only selects them from a pre-built *library of translations* in KGModel.

The mapping $\mathcal{M}(M)$ is a MetaLog program implementing the translation. Our mappings are based on the property that the super-model contains a superset of the constructs of all the

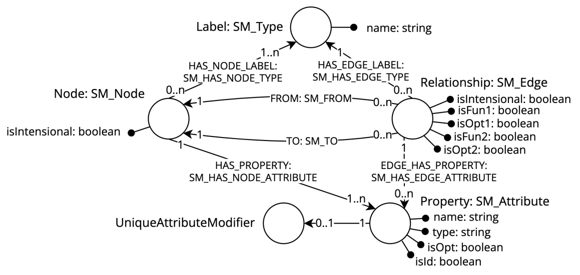


Figure 5: An essential PG model implemented using KG-Model *super-model.* Each construct name is suffixed (with the ":" separator) with the name of the super-construct it instantiates (e.g., Node: SM_Node).

models, up to renaming. $\mathcal{M}(M)$ is structured as two sets of MetaLog programs, *Eliminate* and *Copy*, capturing the following steps: (i) it translates S into the intermediate super-schema S^-, which is a clone of S where all the super-constructs that are not specialized by M are eliminated by encoding them with different super-constructs, supported by M; the others are just copied (line 4); (ii) S^- is downcast into S', by copying and renaming the super-constructs that are specialized in M (line 5). Steps (i) and (ii) are implemented by translating all the respective programs into Vadalog (Section 4), and orderly (i.e., first *Eliminate* and then *Copy*) performing the reasoning tasks in the Vadalog System.

Sections 5.2 and 5.3 now zoom into how a mapping from our super-model into the PG and relational model can be encoded.

5.2 Super-model to Property Graph Model

Many variants of the most common PG model exist, each with its own set of features and limitations. We focus here on the most adopted one, where the constructs are labeled nodes and edges. Nodes can be tagged with multiple labels, and a uniqueness constraint can be imposed on attributes. Plus, there is no support for generalizations. We structure the mapping $\mathcal{M}(M)$ to PGs into the following sets of MetaLog programs:[2]

Eliminate.CopyNodes. SM_Nodes of S are copied into new SM_Nodes of S^-.
Eliminate.CopyUniqueAttributeModifier. *Omitted.*
Eliminate.CopyEdges. SM_Edges of S are copied into new SM_Edges of S^-.
Eliminate.CopyAttributes. SM_Attributes of S are copied into new SM_Attributes of S^-, and linked to the respective new SM_Nodes and SM_Edges.
Eliminate.DeleteGeneralizations.

(1) For every SM_Node n in S involved in a SM_Generalization g, and for every SM_Node a that is an ancestor of n via g, the SM_Type of a is copied into a new SM_Type of S^- and linked to the new node of S^- corresponding to n.
(2) For every SM_Attribute a in S of an SM_Node n involved in a SM_Generalization g, and for every SM_Node c that is a child of n via g, then a is copied into a new SM_Attribute of S^- and linked to the new node of S^- corresponding to c.
(3) For every outgoing (resp. incoming) SM_Edge e from a SM_Node n involved in a SM_Generalization g to a SM_Node m in S, and for every SM_Node c that is a child of n via g, then e is copied into a new SM_Edge of S^- and linked to the new nodes of S^-, corresponding to n and m, respectively.
(4) For every outgoing (resp. incoming) SM_Edge e from a SM_Node n involved in a SM_Generalization g to a SM_Node m in S, and for every SM_Node c that is a child of n via g, let e' be the copy of e in S^-. The SM_Attributes of e are copied to S^- and linked to e'.

Copy.StoreNodes. The SM_Nodes of S^- are copied into new Nodes of S'.
Copy.StoreRelationships. The SM_Edges of S^- are copied into new Relationships of S'.
Copy.StoreProperties. The SM_Attributes of SM_Nodes and SM_Edges of S^- are copied into new Properties of S' and linked to the new Nodes and Relationships.
Copy.StoreUniquePropertyModifiers. All SM_UniqueAttributeModifiers of S^- are copied into new UniquePropertyModifiers of S' and linked to the new Properties.

We discuss the MetaLog code of some rules of the *Eliminate.DeleteGeneralizations* program.

Example 5.1. The following MetaLog rule implements Eliminate.DeleteGeneralizations(1).

$$\begin{aligned}(n : SM\_Node; schemaOID : s)\\ ([: SM\_CHILD]^- \cdot [: SM\_PARENT]) * (a : SM\_Node)\\ [r : SM\_HAS\_NODE\_TYPE](t : SM\_Type; name : w),\\ s = 123 \rightarrow \exists_{sk^S(s)} s^- \exists_{sk^H(n,r)} h \exists_{sk^T(t)} l \exists_{sk^N(n)} x\\ (x : SM\_Node; schemaOID : s^-)[h : SM\_HAS\_NODE\_TYPE]\\ (l : SM\_Type; name : w)\end{aligned}$$

SM_Generalizations are not copied into the target schema, but the SM_Nodes of S^- accumulate multiple types, inherited from their parent nodes, at any level. All the body PG node and edge atoms have the schemaOID attribute, to select the specific super-schema S (123, in this case) we are interested in. We will omit attributes for the sake of compactness. The target schema identifier s^- is obtained with a dedicated Skolem functor. ■

We omit the MetaLog code of *Eliminate.DeleteGeneralizations*(2), whose logic is similar to that of (1). Instead, we concentrate on edge inheritance, focusing on the generation of new SM_Edges, whilst the copy of SM_Attributes is omitted, for space reasons.

Example 5.2. The following MetaLog rule implements Eliminate.DeleteGeneralizations(3) for the inheritance of outgoing edges.

$$\begin{aligned}(c : SM\_Node; schemaOID : s)\\ ([: SM\_CHILD]^- \cdot [: SM\_PARENT])*\\ (n : SM\_Node)[r : SM\_FROM]^-(e : SM\_Edge)[t : SM\_TO]\\ (m : SM\_Node), s = 123 \rightarrow\\ \exists_{sk^E(e,c)} f \exists_{sk^N(c)} x \exists_{sk^N(m)} z \exists_{sk^{FR}(r,c)} u \exists_{sk^{TO}(t,c)} t\\ (x : SM\_Node)[u : SM\_FROM]^-(f : SM\_Edge)\\ [t : SM\_TO](z : SM\_Node)\end{aligned}$$

The rule builds a new SM_Edge connecting SM_Nodes x and z in S^-, which in S correspond to the child node c and to the target node m, respectively. ■

Figure 6 visualizes the portion of the scheme of the Bank of Italy KG, automatically obtained by translating the super-schema we have shown in Figure 4 into the PG model at hand.

[2]Some programs will be omitted here and in Section 5.3 for space reasons.

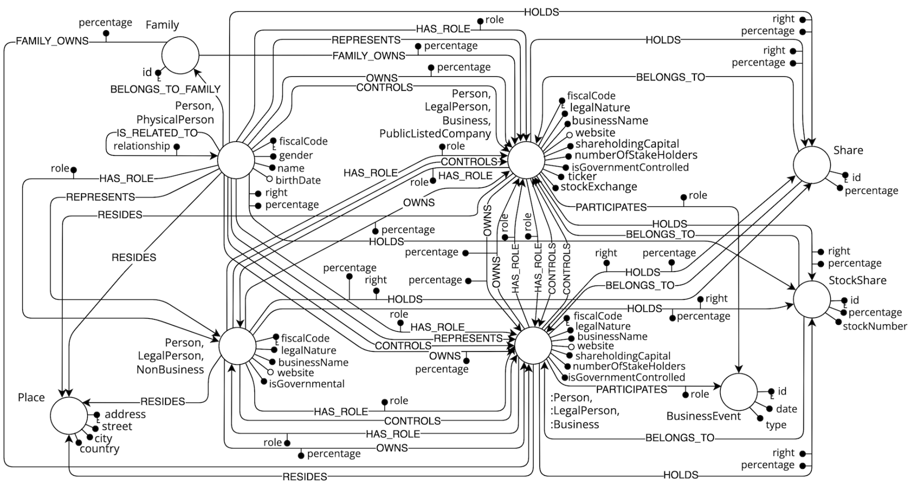


Figure 6: The example super-schema of Figure 4 translated to the PG model.

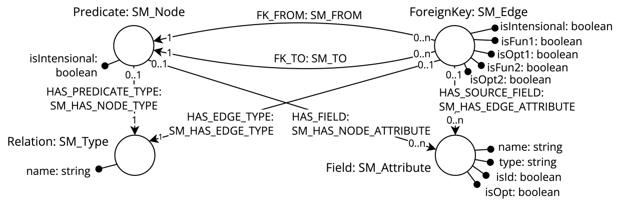


Figure 7: An essential relational model expressed using KGModel *super-model.*

5.3 Super-model to Relational Model

The relational model is a common choice for KG data. Let us see the translation of super-schemas into relational schemas.

Our super-model representation of the relational model is summarized in Figure 7. Relations specialize SM_Type. Each Relation is characterized by a set of Fields, that specialize SM_Attribute. A Predicate is a construct (SM_Node) that connects a Relation to its Fields. ForeignKeys (SM_Edges) constrain a set of Fields of the source relation (referred to via HAS_SOURCE_FIELDS) to take only values from the identifier of the target relation. While the relational model inherently supports many SM_AttributeModifiers, we omit them for space reasons, as their translation is mechanical and uninteresting.

We present part of the mapping $\mathcal{M}(M)$ to the relational model next. Intuitively, the elimination phase simplifies generalizations and many-to-many edges into one-to-many edges, which can be directly converted into relational foreign keys in the copy phase. For the representation of generalizations in the relational model many tactics exist, based on data volumes and access statistics [31]. In the mapping we will adopt the following strategy, omitting the details: we use a relation for each generalization member, connecting each child relation to the respective parent relation via foreign keys.

Eliminate.CopyNodes. SM_Nodes of S are copied into new SM_Nodes of S^-.

Eliminate.CopyTypes. SM_Types of S are copied into new SM_Types of S^-.

Eliminate.CopyNodeAttributes. SM_Attributes of SM_Nodes in S are copied into new SM_Attributes of S^-, and linked to the respective new SM_Nodes.

Eliminate.CopyOneToManyEdges. Let e be a SM_Edge from a SM_Node n to a SM_Node m in S with isFun1 = *false* and isFun2 = *true* (hence, one-to-many), and let n^- and m^- be the SM_Nodes in S^- corresponding to n and m.

(1) For every e, a new SM_Edge e^- is created and linked to the respective from/to SM_Nodes n^- and m^-.
(2) The SM_Attributes of e are copied to S^- and linked to m^-.

The many-to-one edges are eliminated symmetrically. The one-to-one edges can be handled with a similar approach.

Eliminate.DeleteManyToManyEdges. Let e be a SM_Edge from a SM_Node n to a SM_Node m in S with isFun1 = *false* and isFun2 = *false* (hence, many-to-many).

(1) For every e, a new SM_Node p^- of S^- is created; the SM_Type of e is copied into a new SM_Type of S^- and linked to p^-, and the SM_Attributes of e are copied to S^- and linked to p^-.
(2) Let p^- and m^- be the corresponding SM_Nodes of e and m in S^-, respectively. For every e, a new SM_Edge fk_m^- is created in S^- with fixed attributes isOpt1 = e.isOpt1, isFun1 = *false*, isOpt2 = *false* and isFun2 = *false*. Then, fk_m^- is linked to p^- creating a SM_FROM link in S^-, and to m^- creating a SM_TO link. Finally, SM_Attributes of m with isId = *true* are copied in S^- and linked to fk_m^-.
(3) Let p^- and n^- be the corresponding SM_Nodes of e and n in S^-, respectively. For every e, a new SM_Edge fk_n^- is created in S^- with fixed attributes isOpt1 = e.isOpt2, isFun1 = *false*, isOpt2 = *false* and isFun2 = *false*. Then, fk_n^- is linked to p^- creating a SM_FROM link in S^-, and to n^- creating a SM_TO link. Finally, SM_Attributes of n with isId = *true* are copied in S^- and linked to fk_n^-.

Eliminate.DeleteGeneralizations. *Omitted.*

Copy.StorePredicatesAndRelations. *Omitted.*

Copy.StoreNodeAttributes. *Omitted.*

Copy.StoreOneToManyEdges. Let e^- be a SM_Edge from a SM_Node n^- to a SM_Node m^- in S^-, and let n' and m' be the new Predicates of S' corresponding to SM_Nodes of S^-. Then the e^- are copied into new ForeignKeys fk' in S' and linked to the from/to nodes n' and m'; for every SM_Attribute a^- of n^- with isId = *true*, a^- is copied in S' and linked through HAS_SOURCE_FIELD to fk'.

The many-to-one edges are eliminated symmetrically. The one-to-one edges can be handled with a similar approach.

Figure 8 represents the reference KG super-schema illustrated in Figure 4, translated to the relational model.

6 INTENSIONAL COMPONENTS AND TRANSLATIONS

Control edges, which we have defined in Example 4.2, are a case of derived edges in our Company KG. We aim at striking a balance between two forces: design ergonomicity and model independence. To address the former, the data engineer should be able to write a set Σ of high-level MetaLog rules that specify the intensional component by combining and creating instances of constructs of the super-schema and so "speak" the business language. In fact, Figure 3 shows that dash lines are the adopted graphemes for such nodes and edges. For the latter, we need to consider that, at the ground level, the data are stored in a database instance D of a schema S (of model M), which has been generated from the super-schema via one mapping $\mathcal{M}(M)$, as we have seen in Section 5. Σ must be applicable to D, independently of its schema S and model M. Algorithm 2 is the technique we use in KGModel for the materialization of the intensional component.

Algorithm 2 *Intensional Component Materialization Algorithm. Input:* instance D of schema S of a model M, an intensional component Σ; *Output*: materializes the intensional component.

```
1: $\mathbf{M} \leftarrow$ select candidate mappings to $M$ from REPO
2: $\mathcal{M}(M) \leftarrow$ prompt for implementation strategy
3: $\mathcal{V}(M) \leftarrow$ MTV.translateToVadalog($\mathcal{M}(M)$.instance)
4: $I \leftarrow Reason(D, \mathcal{V}(M)^{-1})$    ▹ Import $D$ into the super-model
5: $V^I_\Sigma \leftarrow$ build high-level input views from super-constructs
   instances to construct instances used in $\Sigma$
6: $V^O_\Sigma \leftarrow$ build high-level output views from construct in-
   stances used in $\Sigma$ to super-constructs instances
7: $\mathcal{V}(\Sigma) \leftarrow$ MTV.translateToVadalog($\Sigma \cup V^I_\Sigma \cup V^O_\Sigma$)
8: $I' \leftarrow Reason(I, \mathcal{V}(\Sigma))$
9: $D \leftarrow Reason(I', \mathcal{V}(M))$    ▹ Materialize into $D$
```

In broad terms, when requested to materialize the intensional component of a KG, the SSST tool first loads the instance D into the super-components (see Figure 1), composed of special *instance-level constructs* of the super-model constructs, which are the "instance twins" of the super-constructs (lines 1-4). Then, it generates two sets of rules, V^I_Σ and V^O_Σ, that provide intensional definitions, in terms of the instance-level constructs, for the atom names used in Σ (lines 5-6). Finally, once Σ is applied on the views (lines 7-8), the obtained facts are materialized back into D (line 9) as derived components. The mapping $\mathcal{M}(M)$ from the super-model to M is a core element of the process and we have extensively covered it in Section 5. Here, each mapping is complemented by an instance-level mapping $\mathcal{M}(M).instance$, which symmetrically translates instances of super-schemas into instances of schemas. Let us start with the constructs.

Instance-level Constructs. We enrich the super-model dictionary to make it directly suitable to store instances of super-schemas. A portion of it is shown in Figure 9. The model is extended by introducing for each super-construct C an I_C *instance super-construct*, representing the respective instance counterpart. Each instance super-construct is connected to the respective super-construct by a SM_References edge. In general, instance super-constructs only have the implicit OID attributes and instanceOID to denote the specific instance they refer to, except for I_SM_Attribute, which holds a *value* attribute, to store the value of the instantiated SM_Attribute. With the same logic, the model dictionaries of KGModel are extended with instance constructs as well. For each supported model, KGModel is able to load the source instance D_S into the model instance constructs. The instance D we have used in Algorithm 2 denotes D_S once it has been loaded into the instance constructs.

Super-schema Instances to Schema Instances Translation. A super-schema instance is memorized in the super-model by instantiating the instance-level constructs. With this structure in place, mappings from super-schema instances to schema instances are straightforward extensions of those we have seen in Section 5.1, which translate super-schemas into schemas.

Example 6.1. For example, the I_SM_Attribute copy rules for the PG model is as follows.

$$\begin{aligned}(x : \mathit{I\_SM\_Attribute}; \mathit{instanceOID} : i, \mathit{value} : v)\\ [: \mathit{SM\_REFERENCES}](a : \mathit{SM\_Attribute}), i = 234\\ \rightarrow \exists_{sk^{\mathrm{P}}(x)} w, \exists_{sk^{\mathrm{R}}(x)} r, \exists_{sk^{\mathrm{PI}}(a)} y\\ (w : \mathit{I\_M\_Property}; \mathit{instanceOID} : i, \mathit{value} : v)\\ [r : \mathit{REFERENCES}](y : \mathit{M\_Property})\end{aligned}$$

All the I_SM_Attributes of instance 234 are copied into the corresponding Properties. Note that the identifiers for I_M_Property and M_Property are generated by using linker Skolem functors, to be able to refer to them in other copy rules, like the one for I_SM_HAS_NODE_ATTRIBUTE. ■

Instance Loading. Once the source instance D_S has been loaded into the instance constructs of the target model M, we load it into instance super-constructs: given a translation mapping from super-schema instances to schema instances $\mathcal{M}(M)$, we translate it into Vadalog (line 3) and compute its inverse $\mathcal{V}(M)^{-1}$, which reads the data into the super-model. It is well-known that schema mappings are not necessarily invertible, due to potential information loss [4]. Here we can leverage an interesting *quasi-invertibility concept* from the schema mapping literature [29]. It is not hard to see that information loss can take place only in the elimination phase of the translation. Conversely, the copy phase is invertible by construction. Thus, we simplify $\mathcal{V}(M)^{-1}$ into $(\mathcal{V}(M).\text{copy})^{-1}$. Given D, by applying such quasi-inverse mapping, we obtain a super-schema instance I (line 4), such that the application of $\mathcal{V}(M)$ to it returns D. In fact, the data engineer has possibly crafted the schema of D using super-constructs that are lost in the elimination and are not recovered by the inverse mapping. However, quasi-invertibility is enough in our process, as any potential information loss is never caused by the inversion.

Construction of the Views. Let us now consider the intensional component Σ defined by the rules in Example 4.1 and see how we support it with the input and output views. The goal is to allow the execution of Σ, written in terms of model constructs, on a super-schema instance I. The views are automatically generated by KGModel from a static analysis of Σ. For each body node (edge), an input-node (edge) view is created; for each head node (edge), an output-node (edge) view is created. For a given PG node atom of Σ, an input-view generates the corresponding facts by reading from I_SM_Node and aggregating all the related I_SM_Attributes. Likewise, for a given PG edge atom, an input-view generates the facts by reading from I_SM_Edge and aggregating all the related I_SM_Attributes. The output views perform the inverse transformation, de-normalizing higher-level atoms into the super-schema instance constructs. To make things more concrete, we show one input view for nodes.

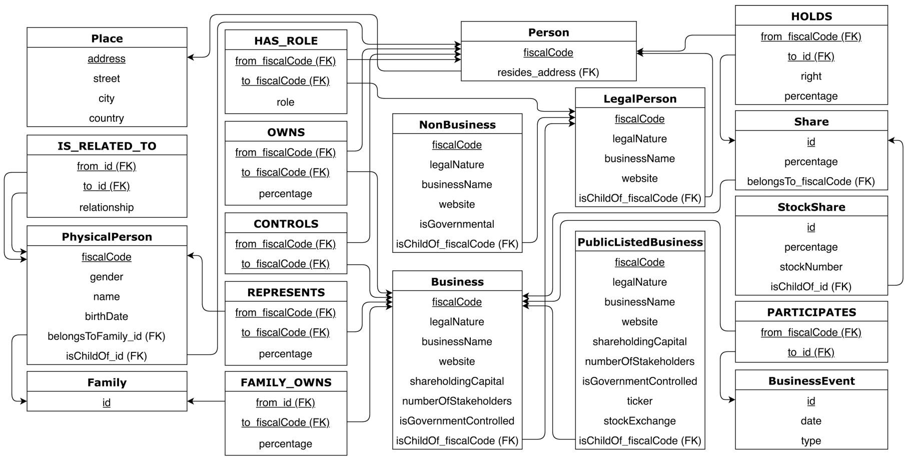


Figure 8: The example super-schema of Figure 4 translated to a relational schema.

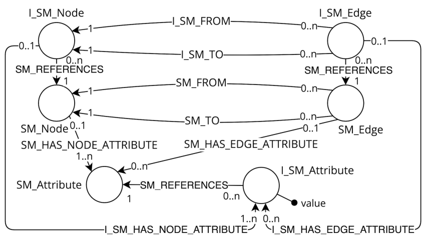


Figure 9: A portion of the super-model dictionary, extended with instance-level constructs.

Example 6.2. An input view of V_Σ^I creating facts for the Business PG node atom in the intensional component of Example 4.1.

$$\begin{aligned}(i : \mathit{I\_SM\_Node}; \mathit{instanceOID} : 123)\,[:\!\mathit{SM\_REFERENCES}]\\ (n : \mathit{SM\_Node})\,[: \mathit{SM\_HAS\_NODE\_TYPE}]\\ (: \mathit{SM\_Type}; \mathit{name} : \text{Business}),\\ (i)\,[:\!\mathit{I\_SM\_HAS\_NODE\_ATTR}]\,(ia : \mathit{I\_SM\_Attribute}; \mathit{value} : v),\\ (n)\,[: \mathit{SM\_HAS\_NODE\_ATTR}]\,(na\!: \mathit{SM\_Attribute}; \mathit{name} : n),\\ (ia)\,[: \mathit{SM\_REFERENCES}]\,(na),\\ \bar{p} = \mathit{pack}(n, v) \rightarrow \exists_{sk^{C}(i)} c\ (c : \mathit{Business}; *\bar{p})\end{aligned}$$

For a given instance (123), for every I_SM_Node where the type for the referenced SM_Node is Business, consider all the attributes whose names are in SM_Attribute and values in I_SM_Attribute, and aggregate all the $\langle n, v\rangle$ pairs with the multi-tuple expression using the operator pack; create a new Business c, unpacking all the pairs as terms of PG node atom, as denoted by the $*$ operator. ■

Performance Considerations and Optimizations. As far as complexity is concerned, the set of applied rules $\Sigma \cup V_\Sigma^I \cup V_\Sigma^O$ is specified in MetaLog, then translated into Vadalog, for which the reasoning task has been shown to be in polynomial time.

As for performance, we have seen that the atoms of V_Σ^I provide the input facts for Σ, and Σ those for V_Σ^O. We can then build the instance I' incrementally, in a stratified way, by first applying V_Σ^I, and materializing the temporary result as a database instance in a staging area; then, the standard reasoning process can take place; finally, I' is stored back into the target system.

There is a clear a trade-off between the needed storage space (for the instances in the super-model dictionary and the materialized versions of V_Σ^I and V_Σ^O) and the perceived elapsed time. In our experience, once V_Σ^I has been materialized, the system can process Σ without any overhead and accumulate changes to the target database. They can be eventually applied to the target database, in a batch fashion, by activating V_Σ^O to flush the instance constructs. For the KG of the Bank of Italy under consideration in this work, the overall control intensional component can be computed in a virtual machine with 16 cores, 128 GB RAM (Intel Xeon architecture) and HDD storage in ~160 minutes, whilst loading and flushing phases require ~15 minutes in total.

Following existing approaches about runtime schema translation [5], future optimized versions of our system could delegate part of the reasoning rules to the underlying database systems, when convenient. However, this improvement requires care, as intensional components typically involve the characterizing features of ontological reasoning, such as a complex interplay of recursion and existential quantification, which can be very laborious or even impossible to express in target languages.

7 CONCLUSION

In building their enterprise and application-specific Knowledge Graphs, companies need to understand, design, communicate, and deploy complex data-driven systems, where the ground data is enriched with a large amount of derived knowledge.

With this work, we aim at pushing the boundaries of the use of KGs in practice and contribute to the delivery of highly engineered systems able to support decision making at its best. Specifically, capitalizing on our experience with a large financial KG, this work provides the data engineer with a conceptual, model-independent, and model-driven framework. From a high-level conceptual specification of the domain to an actionable system, KGModel guides the data engineer through the design journey with a set of principled methodologies and tools, based on a meta-level representation of conceptual design diagrams (in the GSL formalism), and a declarative specifications of reasoning rules (in MetaLog), which are automatically translated into workable schemas and executable reasoning programs to be deployed into the target systems. As we are at the stage of delving into multiple novel KG projects in the Bank of Italy, we will be able to evaluate the advantages of KGModel on a wider set of use cases in terms of design, implementation and verification.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[2] Marjan Alirezaie, Karl Hammar, and Eva Blomqvist. 2018. SmartEnv as a network of ontology patterns. *Semantic Web* 9, 6 (2018), 903–918.

[3] Renzo Angles. 2018. The Property Graph Database Model. In *AMW (CEUR Workshop Proceedings)*, Vol. 2100. CEUR-WS.org.

[4] Marcelo Arenas, Jorge Pérez, Juan L. Reutter, and Cristian Riveros. 2009. Composition and Inversion of Schema Mappings. *CoRR* abs/0910.3372 (2009).

[5] Paolo Atzeni, Luigi Bellomarini, Francesca Bugiotti, Fabrizio Celli, and Giorgio Gianforme. 2012. A runtime approach to model-generic translation of schema and data. In *Inf. Syst.* 269–287.

[6] Paolo Atzeni, Luigi Bellomarini, Michela Iezzi, Emanuel Sallinger, and Adriano Vlad. 2020. Weaving Enterprise Knowledge Graphs: The Case of Company Ownership Graphs. In *EDBT*. OpenProceedings.org, 555–566.

[7] Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. 2005. ModelGen: Model Independent Schema Translation. In *ICDE*. IEEE Computer Society, 1111–1112.

[8] Paolo Atzeni, Paolo Cappellari, Riccardo Torlone, Philip A. Bernstein, and Giorgio Gianforme. 2008. Model-Independent Schema Translation. *VLDB Journal* 17 (2008), 1347–1370.

[9] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. 1999. *Database Systems - Concepts, Languages and Architectures.* McGraw-Hill.

[10] Albert-László Barabási. 2009. Scale-Free Networks: A Decade and Beyond. *Science* 325, 5939 (2009), 412–413.

[11] Luigi Bellomarini, Marco Benedetti, Stefano Ceri, Andrea Gentili, Rosario Laurendi, Davide Magnanimi, Markus Nissl, and Emanuel Sallinger. 2020. Reasoning on Company Takeovers during the COVID-19 Crisis with Knowledge Graphs. In *RuleML+RR*.

[12] Luigi Bellomarini, Daniele Fakhoury, Georg Gottlob, and Emanuel Sallinger. 2019. Knowledge Graphs and Enterprise AI: The Promise of an Enabling Technology. In *ICDE*. IEEE, 26–37.

[13] Luigi Bellomarini, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. 2017. Swift Logic for Big Data and Knowledge Graphs. In *IJCAI*.

[14] Luigi Bellomarini, Eleonora Laurenza, and Emanuel Sallinger. 2020. Rule-based Anti-Money Laundering in Financial Intelligence Units: Experience and Vision. In *RuleML+RR (Supplement) (CEUR Workshop Proceedings)*, Vol. 2644. CEUR-WS.org, 133–144.

[15] Luigi Bellomarini, Markus Nissl, and Emanuel Sallinger. 2021. Rule-based Blockchain Knowledge Graphs: Declarative AI for Solving Industrial Blockchain Challenges. In *RuleML+RR (To Appear) (CEUR Workshop Proceedings)*. CEUR-WS.org.

[16] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *PVLDB* 11, 9 (2018).

[17] Gerald Berger, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. 2019. The Space-Efficient Core of Vadalog. In *PODS*. ACM, 270–284.

[18] Philip A. Bernstein and Sergey Melnik. 2007. Model management 2.0: manipulating richer mappings. In *SIGMOD Conference*. ACM, 1–12.

[19] Eva Blomqvist, Karl Hammar, and Valentina Presutti. 2016. Engineering Ontologies with Patterns - The eXtreme Design Methodology. In *Ontology Engineering with Ontology Design Patterns*. Studies on the Semantic Web, Vol. 25. IOS Press, 23–50.

[20] Eva Blomqvist, Valentina Presutti, Enrico Daga, and Aldo Gangemi. 2010. Experimenting with eXtreme Design. In *EKAW (Lecture Notes in Computer Science)*, Vol. 6317. Springer, 120–134.

[21] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema Validation and Evolution for Graph Databases. In *ER (Lecture Notes in Computer Science)*, Vol. 11788. Springer, 448–456.

[22] Mercedes Campi, Marco Duenas, and Giorgio Fagiolo. 2019. *How do countries specialize in food production? A complex-network analysis of the global agricultural product space.* Technical Report. Lab. of Economics and Management (LEM), Sant'Anna School of Advanced Studies, Pisa, Italy. https://EconPapers.repec.org/RePEc:ssa:lemwps:2019/37

[23] Vasco M. Carvalho, Makoto Nirei, Yukiko U. Saito, and Alireza Tahbaz-Salehi. 2016. *Supply Chain Disruptions: Evidence from the Great East Japan Earthquake.* Discussion papers ron287. Policy Research Institute, Ministry of Finance Japan. https://ideas.repec.org/p/mof/wpaper/ron287.html

[24] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you always wanted to know about Datalog (and never dared to ask). *TKDE* 1, 1 (1989), 146–166.

[25] Anca Chis-Ratiu and Robert Andrei Buchmann. 2018. Design and Implementation of a Diagrammatic Tool for Creating RDF graphs. In *PrOse@PoEM (CEUR Workshop Proceedings)*, Vol. 2238. CEUR-WS.org, 37–48.

[26] Andreea Constantin, Tuomas A. Peltonen, and Peter Sarlin. 2018. Network linkages to predict bank distress. *Journal of Financial Stability* (2018). https://doi.org/10.1016/j.jfs.2016.10.011

[27] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.* 33, 3 (2001), 374–425.

[28] Eric Evans. 2004. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley.

[29] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. 2008. Quasi-inverses of schema mappings. *ACM Trans. Database Syst.* 33, 2 (2008), 11:1–11:52.

[30] Francesca Arcelli Fontana, Hugo Brunelière, Hausi A. Müller, and Claudia Raibulet. 2020. Guest editors' introduction to the special issue on Model Driven Engineering and Reverse Engineering: Research and Practice. *J. Syst. Softw.* 159 (2020).

[31] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional.

[32] J Franks and C Mayer. 1995. Trends in business organization: Do participation and cooperation increase competitiveness? *Ownership and control. In H. Siebertb (Ed), Tubingen: Mohr* (1995).

[33] Birte Glimm, Chimezie Ogbuji, S Hawke, I Herman, B Parsia, A Polleres, and A Seaborne. 2013. SPARQL 1.1 entailment regimes, 2013. W3C Recommendation 21 March 2013.

[34] Georg Gottlob and Andreas Pieris. 2015. Beyond SPARQL under OWL 2 QL Entailment Regime: Rules to the Rescue. In *IJCAI*. 2999–3007.

[35] Karl Hammar, Eva Blomqvist, David Carral, Marieke van Erp, Antske Fokkens, Aldo Gangemi, Willem Robert van Hage, Pascal Hitzler, Krzysztof Janowicz, Nazifa Karima, Adila Krisnadhi, Tom Narock, Roxane Segers, Monika Solanki, and Vojtech Svátek. 2016. Collected Research Questions Concerning Ontology Design Patterns. In *Ontology Engineering with Ontology Design Patterns*. Studies on the Semantic Web, Vol. 25. IOS Press, 189–198.

[36] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutiérrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. Knowledge Graphs. *ACM Comput. Surv.* 54, 4 (2021), 71:1–71:37.

[37] Richard Hull and Roger King. 1987. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Comput. Surv.* 19, 3 (1987), 201–260.

[38] Krzysztof Janowicz, Adila Alfa Krisnadhi, María Poveda-Villalón, Karl Hammar, and Cogan Shimizu (Eds.). 2019. *Proceedings of the 10th Workshop on Ontology Design and Patterns (WOP 2019) co-located with 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 27, 2019.* CEUR Workshop Proceedings, Vol. 2459. CEUR-WS.org.

[39] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. The magic of duplicates and aggregates. In *VLDB*. 264–277.

[40] C. Piccardi and L. Tajoli. 2018. Complexity, centralization, and fragility in economic networks. *PLoS ONE* 13 (2018).

[41] Cesar A. Hidalgo R. and Albert-László Barabási. 2008. Scale-free networks. *Scholarpedia* 3, 1 (2008), 1716.

[42] Register of Institutions and Affiliates Data [n.d.]. GUIDELINE (EU) 2018/876 OF THE ECB. https://cutt.ly/8jJQYys.

[43] Andrea Romei, Salvatore Ruggieri, and Franco Turini. 2015. The layered structure of company share networks. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 1–10.

[44] Pramod J. Sadalage and Martin Fowler. 2013. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence.* Addison-Wesley.

[45] Frank Schweitzer, Giorgio Fagiolo, Didier Sornette, Alessandro Vespignani Fernando Vega-Redondo, and Douglas R. White. 2009. Economic Networks: The New Challenges. *Science* 325, 1 (2009), 1716.

[46] Wei Tang. 2009. *Meta Object Facility.* Springer US, Boston, MA, 1722–1723. https://doi.org/10.1007/978-0-387-39940-9_914

[47] Asena Temizsoy, Giulia Iori, and Gabriel Montes-Rojas. 2017. Network centrality and funding rates in the e-MID interbank market. *JFS* (2017).

[48] Sonja Tilly and Giacomo Livan. 2021. Macroeconomic forecasting with statistically validated knowledge graphs. *CoRR* abs/2104.10457 (2021).

[49] Moshe Y. Vardi. 2016. A Theory of Regular Queries. In *PODS*. ACM, 1–9.

[50] Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. 2014. Model-Driven Design of Graph Databases. In *ER (Lecture Notes in Computer Science)*, Vol. 8824. Springer, 172–185.

[51] Yucheng Yang, Yue Pang, Guanhua Huang, and Weinan E. 2020. The Knowledge Graph for Macroeconomic Analysis with Alternative Big Data. *CoRR* abs/2010.05172 (2020).