Bridging Property Graphs and RDF for IoT Information Management

Abdullah Abbas[1] and Gilles Privat[2]

Orange Labs, Grenoble, France
[1] abdullah.abbas@orange.com
[2] gilles.privat@orange.com

Abstract. "Property graphs" have come to be widely used as the common meta-model of most graph databases. The anterior prevalence of the RDF meta-model, its rich suite of tools and its openness to the linked data cloud, make it essential to ensure the consistency and potential conversion between these two models. We propose to give property graphs a formal semantic grounding based on RDF/RDFS/OWL, with blank nodes reification, geared to JSON-LD serialization. On top of it, we also define a set of core cross-domain ontology classes for the IoT, based on this meta-model, intended to be used for IoT platforms alongside the proposed meta-model.

Keywords: IoT, Web of things, RDF, Property graphs, Meta-model, Ontology, Reification.

1 Introduction

Internet of Things (IoT) platforms, overly numerous as they are [1], have failed to converge on any semblance of a common information model. They mostly rely on low-level ad hoc data models, implicitly object-based and hierarchical, with weak unformalized semantics, falling short of requirements for inter-operation with other platforms and potential sharing of data. Beyond legacy solutions, an evolution that points in the right direction is towards the increasing adoption, though more so industry than academia, of the "property graph" meta-model, the common denominator of most graph databases that are already widely used in the industry. As an evolution of rigid relational database schemata, property graphs [2, 3] provide an obvious advantage in flexibility. Yet they are also highly relevant as an evolution of object-oriented information models that are implicitly based on an underlying arborescent data structure (directed rooted tree), whereas an unrestricted graph is vastly more versatile and expressive. Such a full-fledged graph-based model is the best candidate for a shared information model in the IoT and Cyber-Physical Systems. For this, it should capture not only the semantics of the individual descriptions of IoT and CPS components, but first and foremost, their *structure* and potentially their *behavior*, as we explain in the following sections.

Yet property graphs lack the universality, standardization and formal underpinnings of RDF and do not interoperate directly with linked data and other RDF datasets. Also they do not lend themselves to reasoning with RDF-based reasoning tools or querying with standard query languages such as SPARQL. To overcome these limitations, we propose a tentative formal grounding of a meta-model derived from "property graphs" on the basis of RDF/RDFS/OWL. On top of it, we also propose to define a set of core cross-domain ontology classes for the IoT, based on this meta-model: this cross-domain ontology is meant to ensure the articulation between the proposed property graph model and various domain-specific ontologies (such as may have been proposed by from specialty standards organizations in vertical business sectors such as smart cities, smart industry, smart buildings, etc.)

This work has been contributed to the "Context information Management" Industry Specification Group at ETSI [4], towards the specification of the NGSI-LD APIs [5] and data models.

2 Why Use a Graph-Based Model for the IoT?

Systems and environments about which data is stored and managed by IoT platforms span multiple scales and levels of subsystem-to-system nesting, with configurations that may be fixed or changeable, top-down-designed or "organically" grown in bottom-up fashion. Typical examples of such overarching systems would be smart homes, smart buildings, smart cities, and smart factories. These are typically "systems of systems" for which we need not only document the nature and descriptions of their individual components, as traditional semantic modeling addresses it: it is even more essential to capture the structure that describes the arrangement of subsystems into a system at each relevant level. The expressivity of a rich graph-based model such as property graphs allows us to describe such a complex structural arrangement, where an object-oriented hierarchical model would fall far short of the goal.

But graphs do also match other requirements of a suitable IoT information model:

- They can be visualized in many attractive ways to support human understanding.
- In a well-known benefit of graph databases, they support index-free retrieval according to relationships and paths.
- They can be exported and imported through REST APIs and match the requirements of "truly REST" (HATEOAS) APIs
- They match the overall model of the "linked data" cloud, which is, ultimately, also a graph, though with only one type or arcs corresponding to hyperlinks between resources.

2.1 Structural Representation of IoT/CPS Environments and Systems Through a Graph Model

The Internet of Things (IoT) does, in the extended acceptation [6] we proposed to take, reach beyond connected devices to encompass all kinds of physical entities, be

they composite physical systems, raw passive items or subsets of space. For a smart building, these entities would be walls, windows, rooms, pieces of furniture, floors, corridors, building appliances or entire technical subsystems of the building. For a city, they would be streets, lanes, parking spaces, pieces of street furniture, or buildings themselves, seen at this level as components of the city. These entities, or subsystems of an overarching system being captured, will be represented as the nodes of the overall IoT graph of the top-level system, identified and represented together with the attached devices that may provide them connectivity. The deep structure of Internet of Things (IoT) environments is, in this perspective, made up of all interwoven relationships of physical actuation, sensing, proximity, grouping and containment between these constituent subsystems. The information maintained by the IoT graph is, first and foremost, structural information about the relationships between these entities/subsystems) and devices, usually permanent or semi-static. They may correspond to the following:

- device used as primary sensor for an entity
- device used as secondary sensor for an entity
- device used as direct actuator for an entity
- device acting upon an entity as a side effect
- entity containing another, permanently as a subsystem, or temporarily
- entity adjacent to another

2.2 Link with Context Information Modeling

In the first wave of context-awareness research dating back to the 2000s, context used to be mostly, and implicitly, user-centric, typically capturing e.g. the activity or location of mobile users to adapt services offered to them. The view we presented of the IoT converges with a broader definition of context, where the very notion of context is de-centered and relative. The role of a context management platform does encompass and generalize that of an IoT platform, i.e. to mediate and consolidate data gathered from multiple heterogeneous data sources. For a context management platform, these sources are not only IoT networks or lower-level platforms; they may also be open data/linked data repositories, crowdsourced data, but also proprietary closed databases). In this perspective, the primary data of one application may be the context of another, and vice versa.

Viewing context as a graph is the most general and best-adapted possible data model for this kind of context information. An early and widely publicized definition of context, dating from the first generation of context research, was “any information that can be used to characterize the situation of an entity”. It may in fact remain valid in the broader context we outlined before if we represent entities as the nodes of a graph. Rather than through a vague notion of situation, we define context as the set of properties characterizing these nodes, together with the set of relationships that enmesh them together, and the properties of these relationships. Context is, thus decentered and broadly defined, the graph itself.

For a presentation of the use of graph-based model from the viewpoint of context information management platforms, refer to [7].

2.3 Graph Example Used Throughout the Paper

An example of a simple structural representation of a tiny piece of a city environment will be used as a lead example throughout this paper, presented in the Fig. 1. It describes a parking space, adjacent to two different streets. Information about the streets, and parking place, and the sensors that are monitoring them are explicit by the relationships (represented here as diamonds, just as in legacy "entity-relationship" diagrams) between entities (represented as rectangles), and the properties (represented by ovals) of entities that are shown in the figure. This is a concrete example that shows the full expressivity of a property graph as used to capture not only pure semantics as an RDF graph would, but also structural and behavioral (here real-time state) information.

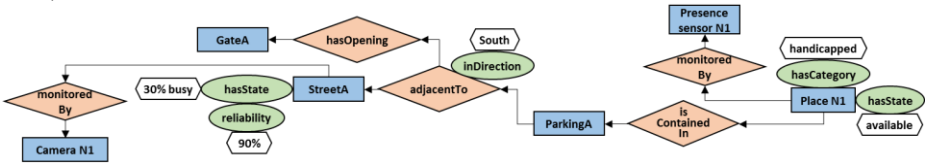

Fig. 1. Property graph example

3 Tentative Formal Definition of Property Graph Meta-Model Based on RDF/RDFS/OWL

At its core, as defined previously, the purpose of the meta-model we propose is to provide a means of conversion between property graph descriptions and the world of linked data which relies on the RDF framework. Defined using RDF/RDFS and OWL vocabulary, the information model provides this kind of compatibility allowing data-model-agnostic exports/imports into and from RDF graphs for an end-user. Thus at this level, this may mainly be summarized as an effort to extend the semantic expressivity of the RDF triples to make them compatible with property graph models (using reification which will be introduced in Sect. 3.2), and simultaneously providing an RDF based information model for data regardless of being internally stored in a property graph oriented database.

3.1 Grounding with Property Graphs and Graph Databases

Property graphs are the implicit semi-formal data models underlying many graph databases. They have gained widespread following as such, more in industry than in academia. They make it possible to attach properties (defined as key-value pairs) to relationships, which RDF does not directly support, but they lack the standardization and formal semantics of RDF and do not interoperate directly with linked data and

other RDF datasets. Also they do not lend themselves to reasoning with RDF-based reasoning tools or querying with standard query languages such as SPARQL.

Property graphs are a common denominator data model for most current Graph-Oriented Databases, such as Titan, Blazegraph, or OrientDB. They are defined as follows [8]:

- A property graph is made up of nodes (vertices), relationships, and properties.
- Nodes contain properties in the form of arbitrary key-value pairs. Keys are strings and values are arbitrary data types.
- A relationship (arc or directed edge of the graph proper) always has a direction, a label, a start node and an end node
- Like nodes, relationships can also have properties.

As can be seen, there are several key differences between property graphs (PG in the following) and the RDF meta-model

- RDF properties are expressed as regular triples, i.e. arcs of the graph, and their target can be either a literal , an IRI or a blank node, whereas the target of a PG property always corresponds to an RDF literal
- Graph links between PG vertices, usually called edges, are first-class citizens and have an internal data model similar to that of a vertex, corresponding to a document or object, with a set of properties defined by key-value pairs.
- They may be undirected, whereas RDF predicates (the arcs of the RDF graph) are always directed)
- Identifiers in a Property Graph are unique only within the scope of a given graph (typically as internal identifiers assigned by the Graph DBMS), not universally unique as URIs/IRIs are deemed to be
- Property graphs can be queried with specific query languages such as Cypher, that can find paths from a given start node, or start nodes identified with specific key/values.
- The edges themselves can be traversed by considering the associated key/value pairs.
- Key value properties cannot *directly* be attached to the edge (predicate) of an RDF triple. Reification is the way to achieve this which is essential to the conversion of property graphs to RDF triples.

3.2 RDF Reification

Superficially, RDF graphs could be seen as just an alternative way to the kind of IoT information we described in Sect. 2, cast as a graph, but they start from a very different premise, placing the emphasis on semantic rather than structural information, mixing individual-to-class (instance to category) with individual-to-individual semantics. RDF is a supremely parsimonious meta-model, defining only, from the theoretical grounding of first-order predicate logic, a basic <*subject* → *predicate* → *object*> triple as building block of a labeled graph, where a generic notion of property (instantiated in a predicate) does not distinguish relationships in the way property graphs do it (though

OWL does make a somewhat similar distinction between object properties and datatype properties). RDF and the associated ontology languages (RDFS/OWL) do natively support all kinds of semantics but they cannot directly express more complex constructs that go beyond the expressivity of first-order logic, such as properties of relationships and their derivatives, which a graph database natively supports.

Several solutions have been proposed transform property graphs into RDF graphs. Making statements about statements may be referred to as reification in general, and is useful, if not indispensable, for e.g. providing information about the provenance (lineage) of data in RDF graphs. It is also indispensable for transforming a property graph into an RDF. Reification in the RDF sense, i.e. morphing a statement into a resource, is the default way to support this transformation. Standard RDF reification [9] explicitly defines a new resource (with type *rdf:statement*) that encompasses a predicate jointly with its associated subject and object (i.e. a triple) This new statement resource is then linked back to the original subject, object and predicate of the statement through a trio of properties with types *rdf:subject*, *rdf:object* and *rdf:predicate*, respectively. A total of 4 additional statements (the infamous "reification quad") are thus required to fully define the reified statement as a resource, this only in order to make this resource the subject of other statement. This is a very cumbersome and heavyweight solution.

Reification by way of blank nodes is another way to achieve this, which is adopted in our meta-model.

Consider the following simple example in Fig. 2 (taken from example of Fig. 1):

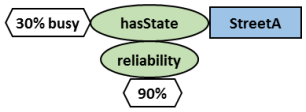

Fig. 2. Property graph example to be reified

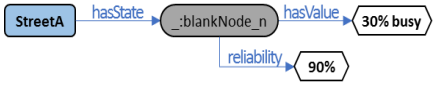

Fig. 3. RDF reified example

Assuming we want to reify this statement from this example:

```
[StreetA → hasState → 30% busy]
```

in order to make it the subject of another statement:

```
[[Statement_1] → reliability → 90%],
```

we just have to add a blank node to obtain an RDF-reified equivalent of our example property graph with three triples as follows (visualized in Fig. 3):

```
[StreetA → hasState → _:blankNode_n]
[_:blankNode_n → reliability → 90%]
[_:blankNode_n → hasValue → 30% busy]
```

This solution is especially convenient when the graph is serialized with JSON-LD (described further in Sect. 4) because blank nodes do not explicitly appear in the textual serialized description, and actually show up only when it is represented as a

graph. It is thus possible for a developer to generate the JSON-LD payload of an API in a form that is very similar to what he would have generated in plain JSON, or in the previous version of the NGSI data model and API.

This can be represented in JSON-LD, without explicitly showing the blank node, as follows:

```
{"@id": "StreetA",
"hasState":{"hasValue": "30% busy",
            "reliability": "90%"}
```

Otherwise, with other reification methods, users and developers are obliged to include supplementary terms and have to deal with complex redundant terms that may distract and confuse users. The simplicity of JSON-LD representation of property graphs reified with blank nodes is a key argument behind the choice of this solution.

Several reification methods are used in the literature, summarized and explained in e.g. [10].We choose two of the of the widely used reification methods to compare with our reification methods from two points of views: JSON-LD corresponding representation and SPARQL query complexity for extracting data.

The two other reification methods are presented in Fig. 4 (Standard RDF reification - with quads) and in Fig. 5 (Singleton property reification).

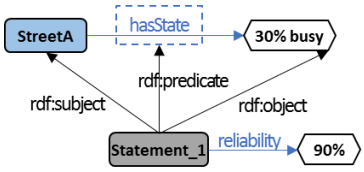

Fig. 4. Standard RDF reification

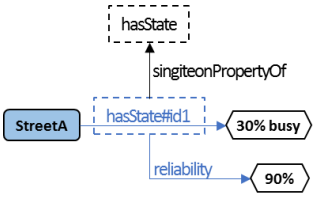

Fig. 5. Singleton property reification

JSON-LD Format:

```
[
{"@id": "StreetA",
"hasState":"30% busy"}

{"@id": "Statement_1",
"subject":
 {"@id": "StreetA"},
"predicate":
 {"@id": "hasState"},
"object":"30% busy",
"reliability": "90%"}
]
```

JSON-LD Format:

```
[
{"@id": "StreetA",
"hasState#id1":"30% busy"}

{"@id": "hasState#id1",
"singletonPropertyOf":
 {"@id": "hasState"},
"reliability": "90%"}
]
```

SPARQL Query:

```
SELECT ?R WHERE {
?st rdf:subject  :StreetA.
?st rdf:predicate
                 :hasState.
?st rdf:object "30% busy".
?st :reliablity ?R
}
```

SPARQL Query:

```
SELECT ?R WHERE {
:StreetA ?p "30% busy".
?p :singletonPropertyOf
                 :hasState.
?p :reliablity ?R
}
```

For reification with blank nodes, the SPARQL query is as follows:

```
SELECT ?R WHERE {
:StreetA :hasState ?bn.
?bn :hasValue "30% busy".
?bn :reliablity ?R.
}
```

Yet for asking a query about value of "hasState" instead of reliability, in our information model ontology we use "owl:propertyChainAxiom" as follows:

```
:hasState owl:propertyChainAxiom (:hasState :hasValue) .
```

This can be defined for all properties similar to "hasState" in the preceding statement. The SPARQL query for asking for the state becomes simple and equivalent to queries for data without reification.

```
SELECT ?V where {
:StreetA hasState ?V.}
```

3.3 Bridging Property Graphs and RDF Graphs

In order to bridge the property graph model to the RDF model, we define custom RDF/RDFS classes for each of the property graph core concepts (Entity, Relationship, Property, and Value). This meta-model is based on reification with blank nodes methods (described in Sect. 3.2), and uses specially defined RDF properties ("hasObject" and "hasValue") to associate blank nodes corresponding to Relationships and Properties with Entities and Values respectively.

Figure 6 depicts visual representation of the RDF/RDFS definitions that we adopt. We note that in Fig. 6 and in the rest of the document we use the term "Property" to refer to its meaning in the property graph sense, which is different from RDF Property. Thus the class "Property" in the lower level of Fig. 6 is an imitation to its counterpart in property graphs. Otherwise we only mean the RDF Property sense when the term "Property" is associated with the term "RDF" (i.e. not when standing alone).

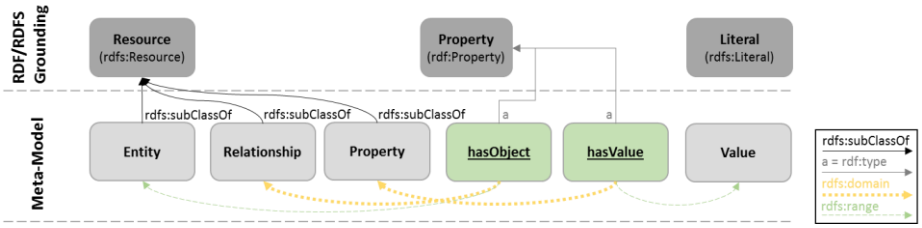

Fig. 6. Meta-model based on RDF/RDFS

Entity Types. Entity types are classes in OWL and shall be defined in a hierarchy as subclasses of the class "Entity".

This allows inheriting common characteristics from super classes, and gives a common vocabulary for the cross-domain ontology and for all the domain-specific ontologies that will ever be defined over the meta-model.

Properties and Relationships Types. Similar to Entity types, the types for Properties and Relationships are defined in a hierarchal manner as subclasses of the classes "Property" and "Relationship" respectively. They are used to categorize properties and relationships, and also to type blank nodes that follow their usages.

Entity, Property, and Relationship are direct subclasses of the *rdfs:Resourse* class.

```
:Entity rdf:type owl:Class ;
        rdfs:subClassOf rdfs:Resource .
```

Value in our meta-model are not limited the *rdfs:Literal*. The class Value should be extendible if needed to allow more value formats in domain specific extensions of our meta-model. Value may be an *rdfs:Literal*, but may also be some specific node type that shall neither be an Entity, a Property, nor a Relationship.

```
:Entity owl:disjointWith :Value .
:Property owl:disjointWith :Value .
:Relationship owl:disjointWith :Value .
```

We also define two primitive RDF properties "hasValue" and "hasObject". These will be used to assign Values to Properties, and Entities to Relationships respectively.

```
:hasValue rdf:type rdf:Property .
          rdfs:domain :Property ;
          rdfs:range :Value .
:hasObject rdf:type rdf:Property .
          rdfs:domain :Relationship ;
          rdfs:range :Entity .
```

"hasValue" and "hasObject" serves an important extension that allows each usage instance of Properties and Relationships to has its own characteristics in the given dataset, which provides a direct way of conversion of data from the property graph model to one of the RDF serializations - JSON-LD for example using RDF reification.

4 Serialization in JSON-LD

JSON-LD is a JSON-based format to serialize Linked Data standardized by W3C [11], and is considered itself one of the possible serializations for data represented in the RDF framework.

The property graph example given Fig. 1 can be represented in RDF, using our reification method with blank nodes as shown in Fig. 7.

Fig. 7. Property graph example transformed to RDF with reification

We note that it is an implementation choice whether all property and relationship instances must be reified, or only where reification is needed (i.e. in case of properties/relationships applied on other properties/relationships). In the RDF example of Fig. 7, for simplicity, we only reified instances where needed.

Our meta-model solution that is based on blank node reification is especially convenient when the graph is serialized with JSON-LD because blank nodes do not explicitly appear in the textual serialized description, and actually show up only when it is represented as a graph. It is thus possible for a developer to generate the JSON-LD payload of an API in a form that is very similar to what he would have generated in plain JSON.

The previous RDF example can be written in the JSON-LD format as follows:

```
[
{"@id": "ParkingA",
"adjacentTo":{"hasObject": {"@id":"StreetA"},
              "hasOpening": {"@id":"GateA"},
              "inDirection": "South"}},
{"@id": "StreetA",
"monitoredBy": {"@id":"CameraN1"},
"hasState":{"hasValue": "30% busy",
            "reliability": "90%"}}
{"@id": "PlaceN1",
"containsSpace": {"@id":"ParkingA"},
"monitoredBy": {"@id":"PresenceSensorN1"},
"hasCategory": "handicapped",
```

```
"hasState": "available"}
]
```

Such JSON-LD representation is much simpler than that for the same set of information if the standard RDF reification with quads was used, which is cumbersome, and shows nodes that are not desired or needed to be shown for developers and users.

5 Cross-Domain Ontology for IoT

The cross-domain ontology provides definition of property types, relationship types, and value types that are considered to be generic across different domains of the IoT. As shown in the diagram of Fig. 8, the cross-domain ontology bridges the property meta-model with domain-specific ontologies.

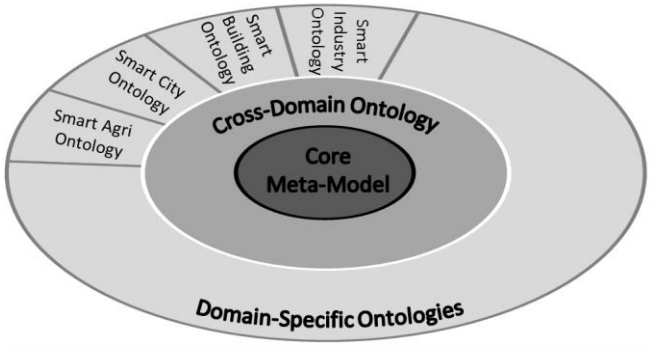

Fig. 8. The cross-domain ontology bridge

5.1 Rationale and Scope

The core meta-model defined before provides the basic conceptual constructs used to define our cross-domain ontology. While the former provides a mean for defining the preliminary structure of the data, the latter provides a mean to classify data instances, and constrain them in a way suitable for representing the informational data of real objects and applications.

The scope of such cross-domain ontology should be decided carefully to be useful for describing real and common situations or constraints, but not to cross the limits to domain-specific definitions that render its vocabulary too specific, and useless for other domain-specific use cases than the one for which it would be defined.

The cross-domain ontology is defined to allow different IoT domains, as pictured above, to share it as a common denominator model that should define a minimal set of common basic notions, avoiding to redefine them separately (and probably in a different way) in each domain-specific vocabulary or ontology. This promotes consistency when applications need to combine data from different domains in a common application framework, and thus enables access to information coming from many

different sources. In other words, all needed domain specific ontologies should be eligible for being defined on the top of our cross-domain ontology.

5.2 Defining Our Cross-Domain Ontology Based on Distinct Classes and Vocabulary

The cross-domain ontology provides vocabulary definitions and structural restrictions of entity types, property types, relationship types, and value types that are considered to be general yet common tools for any system that is using the information model.

Figure 9 shows an RDF/OWL based diagram representing the hierarchical relation between types defined in the cross-domain ontology.

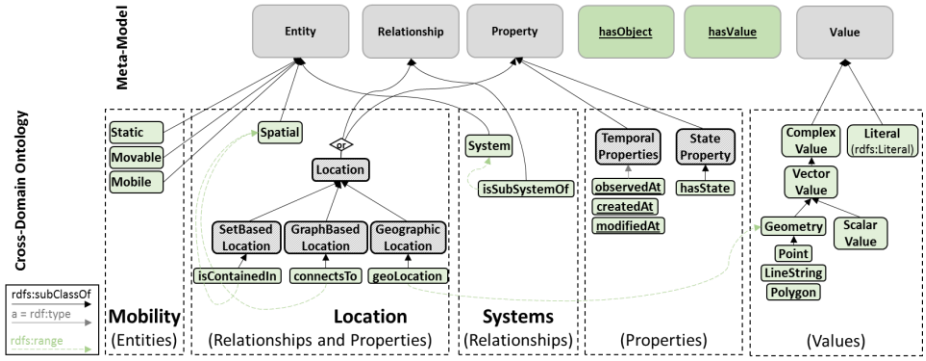

Fig. 9. The cross-domain ontology based on RDF/OWL

The class characteristics of the information model are general enough to be used in a common way throughout different domain specific ontologies. In addition, such distinction is itself useful for enriching the semantics of data.

In the following subsections, we describe different parts of the cross-domain metamodel separately.

Mobility (of Entities). Our information model allows distinguishing between entities based on mobility by typing. This is not to say that it is the only distinction between different entities, but it is a classification that we adopt in the cross-domain ontology. Any domain specific ontology may refer to them, and may define further classifications that are independent on mobility.

Choosing a mobility typing affects the constraints on this entity where these may be useful depending on the application.

1. Static: Location property/relationship associated with such an entity is static (should not normally change value).
2. Movable: Location/relationship property associated with such an entity is semi-static (may change value, but not frequently and not based on a sampling time. In other words, an occasional change may occur).

Location historic data may be available for such an entity.

3. Mobile: Location property/relationship associated with such an entity is instantaneous (changes value frequently or based on a sampling time).
   Location historic data may be available for such an entity.

Such distinction may be useful within some applications for defining error/prompt messages where convenient (e.g. trying to change a static property). It also allows implementations to control historic data storing behavior.

Location. It is suitable for IoT applications to differentiate between three different concepts of location:

1. Geographic: More generally, this can be considered as a coordinate system based location. There are many standards/ontologies that allow expressing geographic location data (e.g. OGC standards or geoJSON).
2. SetBasedLocation: In many cases it may be useful to describe location in terms of a set of entities [12], or in terms of a subset of another location (the term "contains" is used to assert this meaning).
   Example: BuildingA (contains)→ RoomA (contains)→ TableA
3. GraphBasedLocation: This notion of location may be used for indoor or outdoor-navigation, it defines potential routes from one entity (node) to another directly as a agraph
   Example: RoomA (connectsTo)→ RoomB

Systems. In the description of IoT environments, it is needed, as explained in the introduction, to distinguish between different levels of system integration that can be handled separately. We introduce a special vocabulary and typing for dealing with this kind of separate description of subsystems nested within a larger system that comprises them as building blocks. We define an entity type "System", and a relationship "isSubSystemOf" to allow the description of the composition of components to make up a larger system.

Figure 10 shows an example of system of systems. A building (1st system) includes two other systems: an apartment and a parking.

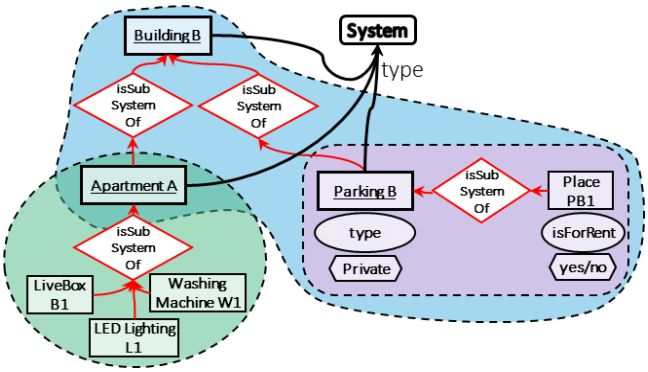

Fig. 10. System of systems example

Temporal Properties. "observedAt", "createdAt", and "modifiedAt" allows users to attach informational temporal tags for entities and other properties/relationships.

State Properties. A Property may be typed as a "StateProperty", which designates the instantaneous state of an entity in the precise sense of system theory, whereby the state summarizes the history of the behavior of a system, as its future states depend only on its current state and future inputs. The main purpose is to differentiate it from raw data and representation like sensors' readings or measurements.

Values. Values in our information model are more general than literals in RDFS. In addition to RDFS literal, they also include complex values. Vector values are an example of a complex value that we use in our information model.

Vector values are tuples (multiple components).

Example: For a geometric value we may need two components:

- type: [Point, LineString, Polygon, …]
- coordinates

Scalar value is a simple value, or in other words a vector value of single dimension.

5.3 Adapted Ontological Restrictions and Typing

In this section we explain how typing works with reified RDF representation, and how restrictions are applied to them in order to enable correct reasoning on the data representation.

A property in our model is an IRI which corresponds to an RDF property, and in the same time to an RDF class (which is a subclass of the class Property of the meta-model). The RDF class interpretation of the IRI will be used to type blank nodes that follow the property usage in the reified form.

The same holds for relationships. A single IRI corresponds to an RDF property and an RDF class (which is a subclass of the class Relationship of the meta-model).

Figure 11 shows an example of how typing holds on blank nodes following the reification of the relationship "isContainedIn".

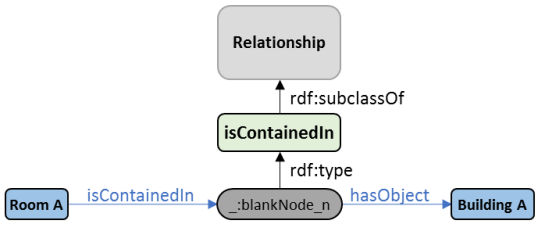

Fig. 11. Blank node typing example for reification

The example of Fig. 11 includes the information that "Room A" is spatially contained in "Building A". According to the restrictions (refer to Fig. 9 corresponding the cross-domain ontology), "Building A" should be an entity of type "Spatial". Yet this restriction cannot be directly asserted by defining "rdfs:range" of the property "isContainedIn" since blank nodes are actually the nodes follows the RDF property usage and not (directly) the spatial entity.

This is solved in two steps. The first step is to assert a property chain as follows:

```
:isContainedIn owl:propertyChainAxiom
                    (:isContainedIn :hasObject) .
```

In our example this means that we can infer that "Building A" is also directly connected to the "isContainedIn" RDF property, and not only to the "hasObject" property.

The second step is to define the RDF class that is a range for the "isContainedIn" RDF property. This should be a union of two RDF classes, the RDF class "isContainedIn" (to account of the blank node instances) and the RDF class "Spatial". This is defined in OWL as follows:

```
:isContainedIn rdfs:range
            [ rdf:type owl:Class ;
              owl:unionOf ( : isContainedIn :Spatial ) ] .
```

6 Conclusion

The use of graph-based representations for IoT information, as put forward in this paper, should make it clear that graphs are not used only to represent purely semantic information in the way RDF graphs do, even in these graphs can ultimately be converted to RDF graphs a universal graph meta-model, as proposed here! Graphs have indeed a long history of being used by both practitioners and theorists to represent both the structure and the temporal properties of complex systems, in a way that did

not account for any kind of semantics! We have only alluded here to the temporal aspect of these representations through the definition of state properties, but this is where the largest body of theory exists to represent the state transitions of systems as graphs in a multiplicity of different possible models (Moore/Mealy automata, Grafcet, Petri Nets, to name just a few …). "Semanticizing" these kinds of behavioral graphs in ways that take stock of this accumulated knowhow remains an ambitious challenge. As for the structural representations of systems and systems of systems, we have just proposed a set of core basic constructs to embed them in the property graph model. This work will be expanded to be ultimately proposed as an ETSI standard-track specification.

Acknowledgments. This work has been supported by the European Union's Horizon 2020 research and innovation programme within the FI-NEXT project under grant agreement No 732851-3. This work is also studied within "Thing'in" project for management of IoT data at Orange Labs in France.

References

1. "List Of 450 IoT Platform Companies," [Online]. Available: https://iot-analytics.com/product/list-of-450-iot-platform-companies/.
2. M. A. Rodriguez and P. Neubauer, "Constructions from Dots and Lines," *Bulletin of the American Society for Information Science and Technology*, 2010.
3. R. Angles, "A Comparison of Current Graph Database Models," in *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, 2012.
4. "'"Context information Management" Industry Specification Group at ETSI," [Online]. Available: https://docbox.etsi.org/ISG/CIM/Open.
5. "NGSI-LD Application Programming Interface (API)," 2018. [Online]. Available: http://www.etsi.org/deliver/etsi_gs/CIM/001_099/004/01.01.01_60/gs_CIM004v010101p.pdf.
6. G. Privat, "Extending the Internet of Things," *Communications & Strategies, Digiworld Economic Journal*, pp. 101-119, 2012.
7. W. Li, G. Privat, J. M. Cantera, M. Bauer and F. Le Gall, "Graph-based Semantic Evolution for Context Information Management Platforms," *GIoTS*, 2018.
8. D. D. Guinard and V. M. Trifa, Building the web of things, Shelter Island: Manning, 2016.
9. "RDF Schema 1.1," [Online]. Available: https://www.w3.org/TR/rdf-schema/.
10. J. Frey, K. Müller, S. Hellmann, E. Rahm and M.-E. Vidal, "Evaluation of Metadata Representations in RDF stores," *Semantic Web – Interoperability, Usability, Applicability an IOS Press Journal*, 2017.
11. "JSON-LD 1.0," [Online]. Available: https://www.w3.org/TR/json-ld/.
12. T. Flury, G. Privat and F. Ramparany, "OWL-Based location ontology for context-aware services," in *in Proceeding of the workshop on Artificial Intelligence in Mobile Systems. Bristol - UK*, 2004.