Beyond a House of Sticks: Formalizing Metadata Tags with Brick

Gabe Fierro
UC Berkeley
gtfierro@cs.berkeley.edu

Jason Koh
UC San Diego
jbkoh@cs.ucsd.edu

Yuvraj Agarwal
Carnegie Mellon University
yuvraj@cs.cmu.edu

Rajesh K. Gupta
UC San Diego
gupta@eng.ucsd.edu

David E. Culler
UC Berkeley
culler@cs.berkeley.edu

ABSTRACT

Current efforts establishing semantic metadata standards for the built environment span academia [3], industry [1] and standards bodies [2, 28]. For these standards to be effective, they must be clearly defined and easily extensible, encourage consistency in their usage, and integrate cleanly with existing industrial standards, such as BACnet. There is a natural tension between informal tag-based systems that rely upon idiom and convention for meaning, and formal ontologies amenable to automated tooling.

We present a qualitative analysis of Project Haystack [1], a popular tagging system for building metadata, and identify a family of inherent interpetability and consistency issues in the tagging model that stem from its lack of a formal definition. To address these issues, we present the design and implementation of the Brick+ ontology, a drop-in replacement for Brick [3] with clear formal semantics that enables the inference of a valid Brick model from an informal Haystack model, and demonstrate this inference across five Haystack models.

CCS CONCEPTS

• **Information systems → Ontologies**; *Data encoding and canonicalization*; *Information extraction.*

KEYWORDS

Smart Buildings, Building Management, Metadata, Ontologies, OWL, RDF, Brick, Haystack

ACM Reference Format:
Gabe Fierro, Jason Koh, Yuvraj Agarwal, Rajesh K. Gupta, and David E. Culler. 2019. Beyond a House of Sticks: Formalizing Metadata Tags with Brick. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Smart buildings have long been a target of efforts aiming to reduce energy consumption, improve occupant comfort, and increase operational efficiency. Although a substantial body of work advances the state-of-the-art — including automated control [8, 24, 31], modeling [25] and analysis [16, 30] — such approaches do not see widespread use due to the prohibitive cost of configuring implementations to each building. A major factor in this cost is due to lack of interoperability standards; without such standards, the rollout of energy efficiency measures involves customizing implementations to the one-off combinations of hardware and software configurations that are unique to each building. Limited deployment of energy efficiency applications constrains the ability to evaluate potential savings [20]. Recent studies by the US Department of Energy [14, 21] have established that a lack of interoperability standards for buildings reduces the cost-effectiveness and scalability of energy efficiency techniques and analyses.

Semantic metadata standards present a promising solution to enabling interoperability by offering uniform descriptions of building resources to application developers and building operators. Today, semantic metadata standardization efforts for buildings span academia [3], industry [1, 29] and standards bodies [2, 28]. As applications developed for the built environment become increasingly data-focused, recent metadata standard efforts have shifted from supporting the initial construction and commissioning phases of operation to enabling robust descriptions of the provenance and context of collected data.

1.1 Brick and Haystack Metadata Systems

Emerging data-oriented metadata standards differ in their support for *consistent* and *extensible* use. De-facto industrial metadata practices have embraced unstructured vendor- and building-specific idioms intended for human consumption rather than programmatic manipulation. Several standardization efforts have arisen to address the ad-hoc nature of building metadata. Of these, Brick [3] and Project Haystack [1] have seen adoption and investment from academic and industrial sources, and are involved in the ASHRAE 223P effort to standardize semantic tagging for building data [2].

Project Haystack is a commonly-used open building metadata standard that replaces unstructured labels with semi-structured sets of tags[1]. However, the informal and ad-hoc composition of these tags precludes consistent usage; this leaves interpretation of tags up to the tacit knowledge of domain experts.

Brick is a recently introduced metadata standard designed for completeness (describing all of the relevant concepts required for applications), expressiveness (capturing the explicit and implicit relationships required for applications) and usability (fulfilling the

[1]Referred to as "Haystack" in the rest of the paper

needs of domain experts and application developers). Although evaluations of Brick demonstrate its ability to robustly capture a wide variety of application requirements, the story of how Brick integrates with existing tooling and industrial practices, such as Haystack, has been less clear.

Put simply, Brick and Haystack serve different goals. Brick is designed for the complete and consistent modeling of concepts required for developing portable software that can be deployed at scale. Haystack is designed for building managers and engineers who need *familiar idioms* for developing and using software designed to function on a small number of buildings. However, these practices are not sufficient for the large-scale standardization of *consistent* semantic metadata necessary for the widespread deployment of energy efficiency applications. This requires a set of rules formalizing how metadata can be *defined*, *structured*, *composed* and *extended.*

In this paper, we present the design and implementation of *Brick+*, a drop-in replacement for the Brick ontology with clear formal semantics designed for the sensible composition of concepts required for portable building applications. The key design principle of Brick+ is the choice to *model concepts in terms of the formal composition of their properties.* This is more expressive than the original Brick class hierarchy which captures *specialization*, but not behavior. Brick+ enables the inference of properties beyond what can be captured by tag-based metadata schemes or the original Brick schema, including modeling the behavior of equipment and points and formalizing Haystack models. Ultimately, this enables the inference of a formal Brick+ model from an informal Haystack model.

1.2 Overview

§3 presents an analysis of the systemic interpretability and consistency issues endemic to the Haystack metadata system, and motivates the need for formal rules for composition. §4 presents the design of Brick+, a drop-in replacement for Brick with clear formal semantics. Brick+ defines a class lattice that structures the composition of concepts. This enables Brick+ to define inference from Haystack's informal tags to formal Brick classes. §5 presents the implementation of Brick+ using the OWL-DL ontology language, and defines the process by which a Brick model can be inferred from a set of tagged Haystack entities. §6 evaluates the Brick+ ontology and inference methodology by observing the accuracy of classifying entities from five Haystack models to Brick+, and examining the additional properties that can be inferred by Brick+ over 104 existing Brick models. §7 summarizes ongoing and future efforts to integrate the Brick and Haystack metadata standards and concludes.

2 BACKGROUND

We define a set of concepts for later use, provide an overview of the Brick and Haystack metadata models, and discuss how Brick+ fits into the existing body of literature.

2.1 Definitions

We refer to the following terms throughout the paper:

- A **tag** is an atomic fact or attribute; tags may or may not be associated with a value.
- A **tag set** is an unordered collection of tags associated with an entity.
- A **valid tag set** is a tag set with a clear, real-world definition.
- An **entity** is an abstraction of a physical, logical or virtual item.
- A **class** is a category of entities defined by a particular shared purpose and properties.

In Brick and Brick+, classes are organized by the subclass and superclass relationships between classes. This organizes classes naturally in terms of more specific or more general concepts. For example, the class of "sensors" is more general than the class of "temperature sensors" (sensors that measure the temperature property of some substance) and the class of "air sensors" (sensors that measure properties of air), which are both more general than the class of "air temperature sensors" (sensors that measure the temperature property of air). In Brick, `Air Temperature Sensor` is the class of all entities that measure the temperature of air.

2.2 Haystack

Haystack defines entities as a set of *value* tags (representing key-value pairs) and *marker* tags (singular annotations). Value tags define attributes of entities such as name, timezone, units and data type. `Ref` tags are a special kind of value tag that refer to other Haystack entities. Haystack provides a dictionary of defined tags on its website [1]. The set of marker tags for an entity constitute the "tag set" for that entity and construe the concept of which the entity is an example (its "type").

2.3 Brick

The Brick ontology has two components: an extensible *class hierarchy* representing the physical and logical entities in buildings, and a minimal set of *relationships* that capture the connections between entities. A Brick model of a building is a labeled, directed graph in which the nodes are entities and the edges are relationships. Brick is defined using the Resource Description Framework (RDF) data model [18], which represents graph-based knowledge as tuples of (`subject`, `predicate`, `object`) termed *triples.* A triple states that a subject entity has a relationship (predicate) to an object entity. Line 2 of Figure 3 is a triple for which the subject is `:sensor1`, the predicate (relationship) is `a`, and the object is `brick:Temperature_Sensor`.

Brick and Brick+ are both defined with the RDFS [13] and OWL [6] knowledge representation languages. These languages allow the expression of rules and constraints for authoring ontologies, which can be interpreted by a *semantic reasoner* such as HermiT [12] to materialize inferred triples from a set of input triples. Brick+'s use of a semantic reasoner is covered in §5.

2.4 Prior Metadata Construction Efforts

Several other works conduct inference and classification to extract structure from unstructured building metadata, including leveraging semi-supervisied learning approaches to learn parsing rules for unstructured labels [7, 17], classifying sensors by examining historical timeseries data [11, 15], or by combining timeseries analysis

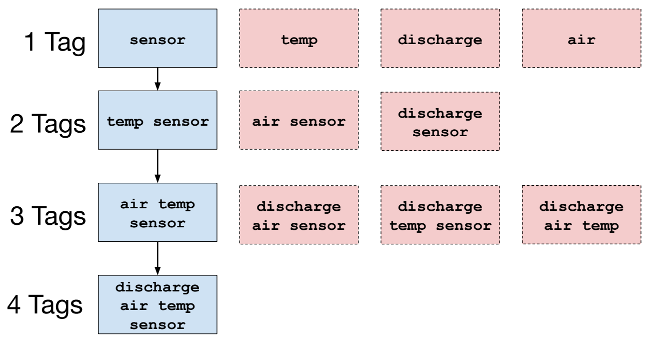

Figure 1: The set of valid (blue + solid outline) and invalid (red + dashed outline) tagsets for a set of four tags. The class hierarchy is established from top to bottom; subclass relationships are indicated by arrows.

with label clustering [5]. These efforts are largely complementary to Brick+. Brick+ defines formal methods for inference-based classification of tagged entities, but requires external support for extracting tagged entities from unstructured metadata.

Beyond the building domain, there is a family of work [22, 23] using ontologies to provide structure to tag-based folksonomies [19]. The approaches developed in these works, along with work on formal concept analysis [34] and concept lattices [33], form the theoretical basis for Brick+.

3 SYSTEMIC TAG ISSUES IN HAYSTACK

Although Haystack models have seen increasing adoption, the design of the Haystack data model has several intrinsic issues that limit its consistency and interpretability.

3.1 Lack of Formal Class Hierarchy

A well-formed class hierarchy organizes concepts by their specificity. This is essential for the creation of consistent metadata models because it facilitates automated discovery of classes by way of traversing the hierarchy for more general or more specific concepts. In the process of identifying an appropriate class for an entity, a user can browse the hierarchy from the most general classes (equipment, location, sensor, setpoint, substance) to the specific class whose definition best describes the entity.

A well-formed class hierarchy is extensible. Users can create new, more specific classes that subclass existing, but more general, superclasses. Even in the absence of a textual definition for this new class, the subclass relationship provides an immediate contextual scoping for how the class is meant to be used.

Haystack lacks an explicit class hierarchy, and the informal construction of Haystack complicates the automated generation of one. Recall that the type of each entity in a Haystack model is defined by a set of tags. We can formalize this as:

Definition 3.1. The set of tags for a class or entity x is given by $T(x)$. The definition of a class C is any entity that has the tags defined by $T(C)$. An entity e is a member of class C if $T(e) \supseteq T(C)$

This definition embeds the key assumption of tag-based metadata: subsets of tags convey more generic concepts. As an example, the pseudo-class identified by the Haystack tags `discharge air temp setpoint` is a subclass of the class identified by the tags `air temp setpoint`. However, the use of tag sets to specify the subclass relationship is insufficient for a well-formed class hierarchy because it is possible to construct two class definitions C_i and C_j such that $T(C_i) \supseteq T(C_j)$ but the definition of C_i is not semantically more specific than C_j.

Consider the counter example of two concepts: `Air Flow Setpoint` (the desired cubic feet per minute of air flow) and `Max Air Flow Setpoint` (the maximum allowed air flow setpoint). In Haystack, `Air Flow Setpoint` would be identified by the `air`, `flow` and `sp` tags, and `Max Air Flow Setpoint` would be identified by the `air`, `flow`, `sp` and `max` tags. Although suggested by the set-based relationship, `Air Flow Setpoint` is *not* a superclass of `Max Air Flow Setpoint`: the former is a setpoint, but the latter is actually a parameter governing the selection of setpoints and therefore belongs to a distinct subhierarchy.

As a result, the rules for defining valid tagsets for subclass relationships must be defined in terms of which tags can be added to a given tag set to produce a valid subclass relationship. Defining a concept requires knowing which tags *cannot* be added; without a clear set of rules for validation, a user may use the `max` and `min` tags to indicate the upper/lower bounds of deadband-based control, which is inconsistent with the intended usage of these tags.

3.2 Balancing Composability and Consistency

One of the primary benefits of an tag-based metadata scheme is composability. A dictionary of tags provides the vocabulary from which users can draw the terms they need to communicate some concept. Composing sets of tags together allows for communication of increasingly complex concepts, and adding new tags to the dictionary exponentially increases the number of describable concepts.

However, increased composability comes at the cost of lower *consistency*, a clear, unambiguous, one-to-one mapping between a set of tags and a concept. Without rules defining composability, consistent interpretation of a tag set is dependent upon idiom, convention and other "common knowledge" of the community using the tags. As a result, the set of tags used by one individual to describe an entity may have multiple meanings or no meaning at all to other individuals. In other words, the intended meaning of a tag becomes more ambiguous the more contexts in which it is used. For example, using the tags `heat`, `oil` and `equip` on an entity does not state if the equipment heats oil or if oil is what is being heated.

To mitigate this effect, Haystack defines "compound" tags. These are concatenations of existing tags into new atomic tags with specific semantics distinct from that of its constituents. For example, the `hotWaterHeat` compound tag is defined specifically as indicating that an air handler unit has heating capability using hot water. This trades composability — which tags can be used together — with consistency — an unambiguous definition for a set of tags. Haystack calls these "semantic conflicts":

> Another consideration is semantic conflicts. Many of the primary entity tags carry very specific semantics. For example the `site` tag by its presence means the data models a geographic site. So we cannot reuse the

site tag to mean something associated with a site; which is why use the camel case tag siteMeter to mean the main meter associated with a site.[26]

The most common types of semantic conflicts concern *process tags* and *substance tags*. We explore how ambiguities arise for these two family of tags; §4 demonstrates how Brick handles these issues.

Process Tags. For a metadata scheme to consistently describe a process, it must decompose a process into entities and capture how each entity relates to the process: does the entity monitor or control the process? Does it transport a substance or provide a means for two substances to interact?

It is difficult for a limited dictionary of tags to capture unambiguously the family of concepts involved. Consider the case of an air handling unit that heats air by passing it around a coil of hot water. With a limited dictionary of tags, most of the entities (equipment and points) involved will be tagged with hot, heat, air and/or water. However, a flat set of tags does not permit any differentiation between concepts that share the same tags: hot water heat to describe a device that produces hot water by heating it, or hot water heat to describe a device that uses hot water to heat something else (such as air).

Table 1 categorizes the intended usage of each Haystack tag containing the word "heat" (including "reheat") or "cool." Without this table or the Haystack documentation in hand, it is difficult to discern when to use a compound tag or several tags together: chilled+waterCooled, chilledWaterCool or chilled+water+cool? Furthermore, there is no formal, programmatically accessible form of the documentation that would allow this to be done in an automated fashion.

Substance Tags. What constitutes sufficient and consistent descriptions of substances (such as water and air) depends upon the breadth of intended use. Existing building metadata systems do not model substances directly; instead, they describe equipment and points in terms of what substances they manipulate, measure or utilize. Thus, an effective metadata scheme for substances must capture *at least* the nature of the relationship between substances, equipment and points.

Flat tag structures lack the expressive power to make these distinctions unambiguous. To reduce ambiguity, Haystack uses substance tags only on points and uses compound tags for equipment. For example hot water valve cmd and chilled water entering temp sensor use the water substance tag, but an air handler unit with a water-based chiller would use chilledWaterCool. This means that substance tags cannot be used to identify which points *and equipment* relate to a given substance. Furthermore, to perform such a query, a user would need to know the entire family of tags that relate to that substance. In the case of "water", this list is water, waterCooled, waterMeterLoad, chilledWaterCool, chilledWaterPlant, hotWaterHeat, hotWaterPlant and hotWaterReheat, not to mention any user-defined tags.

3.3 Lack of Composition Rules

Haystack sacrifices composability of tags for more consistent interpretability, such as through the use of compound tags. Without a set of rules for how tags *can* be composed, there is no programmatic or automated mechanism to enforce or inform consistent usage of the tag dictionary. Haystack contains a small set of explicit rules, but largely relies upon idiom and human interpretation for consistency.

	Tag	Desc. equip	Desc. point	Desc. mechanism	For AHU	For VAV	For Coil	For Valve	For Chiller	For Boiler
heating	heat	×	×				×	×		
	heating		×							
	hotWaterHeat	×		×	×					
	gasHeat	×		×	×					
	elecHeat	×		×	×					
	steamHeat	×		×	×					
	perimeterHeat	×				×				
reheating	reheat		×			×				
	reheating		×			×				
	hotWaterReheat	×		×		×				
	elecReheat	×		×		×				
cooling	cool	×			×	×	×			
	cooling		×							
	coolOnly	×			×					
	dxCool	×		×	×					
	chilledWaterCool	×		×	×					
	waterCooled	×							×	
	airCooled	×							×	

Table 1: An enumeration of the intended use and context of tags relating to heating and cooling, as given by the Haystack documentation. Note the differences in diction across compound tags, and how some compound tags could be assembled from more atomic tags. Some tags are used both for equipment and for points when equipment is modeled as a single point (such as VFDs, Fans, Coils)

Extending Tag Sets. In order to encourage consistent usage, metadata schemes need rules for generating new concepts and generalizing existing concepts. Rules for *generating* new concepts allow these concepts to be qualified by their relation to existing classes. Rules for *generalizing* existing concepts allow users (and programs) to reason about the behavior of a group of concepts. Formal mechanisms for generalization and specialization aid the discoverability, interpretability and extensibility of a metadata scheme. Unfamiliar concepts can be understood or referenced by their behavior or superclasses, and new concepts can be added seamlessly.

Concepts in Haystack can be extended through annotation with additional tags; e.g. temp sensor refines the concept of sensor. However, tags cannot be freely combined (Figure 1). One mechanism for defining valid tag sets parameterizes existing tag sets with a choice from a set of mutually exclusive tags. Haystack explicitly defines several of these. Two examples from many:

(1) The heating method for an AHU, which can given by one of the tags gasHeat, hotWaterHeat, steamHeat and gasHeat.
(2) The family of water meters recognized by Haystack can be differentiated by the tags domestic, chilled, condenser, hot, makeup, blowdown and condensate.

Haystack also has many *implicit* rules for defining valid extensions to tag sets. Application of these rules largely depends upon domain knowledge – for example an entity will likely not have two distinct substance tags such as air and water – as well as informal idioms conveyed through documentation. An example of the latter is the convention that points (sensors, setpoints and commands) will have a "what" tag (e.g. `air`), a "measurement" tag (e.g. `flow`) and a "where" tag (e.g. `discharge`). However, this is not a hard and fast rule, and many of the tag sets in Haystack's documentation break with this convention. Consequently, there is no clear notion of how concepts can be meaningfuly extended or generalized, which limits the extensibility of Haystack.

Modeling Choices. The lack of formal structures for constructing tag sets means that enforcement of consistency – choosing the same set of tags to represent the same concept – relies upon the conventions of industrial practice and the idioms of the Haystack community. As a result, there is substantial variation in how the same concept is modeled.

One prominent example in Haystack is the choice of whether to model pumps and fans as equipment or as points. Although pumps and fans are equipment, in many BMS they are represented by only a single point (usually the speed or power level). Haystack's documentation encourages simplifying the representation of such equipment under such circumstances:

> Pumps may optionally be defined as either an `equip` or a `point`. If the pump is a VFD then it is recommended to make it an `equip` level entity. However if the pump is modeled [in the BMS] a simple on/off point as a component within a large piece of equipment such as a boiler then it is modeled as just a `point`.[27]

Complex predicates such as these complicate the querying of a Haystack model. In particular, exploratory queries have to take the family of modeling choices into account: to list all of the pumps in a Haystack model, it is not sufficient to only look for entities with the `pump` and `equip` tag.

3.4 Discussion

These issues with tag-based metadata inhibit extensibility and consistency at scale. Most Haystack models are designed to be used by small teams familiar with the site or sites at hand, so it is enough for these models to be *self-consistent*. As long as there is agreement on how to tag a given concept, the informality of the model is not as detrimental; most tag sets in Haystack make intuitive sense to domain experts. However, the lack of formalization — specifically, a lack of a formal class hierarchy and rules for composability and extensibility — presents issues for adoption as an industrial standard and basis for automated analysis and reasoning.

In the next sections, we show that the tradeoff between composability and consistency is tied to the choice to use tags for annotation as well as definition. With an explicit and formal class hierarchy it is possible to design a system that exhibts the composability of simple tags, while retaining the consistency and extensibility of an ontology.

4 DESIGN OF BRICK+

Although Brick [3] establishes a formal class hierarchy and a set of descriptive relationships, it lacks the structure for inference of classes from tags and contains a number of design issues that impede this development. This motivates the design of *Brick+*, a drop-in replacement ontology for Brick that extends the hierarchy of described concepts to include fine-grained semantic properties and defines an explicit mapping from Brick concepts to sets of tags. Together, these enable the *programmatic interpretation* of tag sets, therefore eliminating the consistency and interpretability issues inherent to a tags-only design (§3).

4.1 Limitations of Brick

The design and implementation of Brick has several issues which inhibit formalizing the relationship between classes and tags.

No formal equivalence between tags sets and classes. Brick models a class hierarchy using a special construction called a TagSet. A TagSet has a definition, a set of related tags, and a name composed of each of the tags concatenated together. The Brick ontology defines which tags are used with which TagSets, but fails to capture bidirectional equivalency between the two definitions. Brick can retrieve the tags associated with a TagSet, but given a set of tags, Brick cannot infer the set of possible TagSets.

No modeling of function or behavior. The Brick class hierarchy relates different TagSets only by a "subclass" relationship; there is no semantic information to distinguish classes in terms of their behavior. The simple association of tags to TagSets also does not offer any semantic information. Enhancing the class definitions with more semantic information would increase the usability of Brick and the discoverability of concepts.

Inconsistent modeling and implementation. The implementation of the Brick ontology consists of a set of Turtle[2] files containing the ontology statements. These files are generated by a Python script that transforms an CSV-based specification into the Turtle syntax for RDF. This process is brittle, error-prone and difficult to test and extend.

4.2 Overview of Brick+

Brick+ has three components: a class lattice defining the family of equipment, points, locations, substances and quantities in buildings; a set of expressive relationships defining how entities behave and how they are connected, contained, used and located; and a family of tags defining the atomic attributes and aspects of entities

The implementation of Brick+ relies upon the use of a *semantic reasoner*, piece of software that materializes the set of facts deduced through the application of the logical rules contained within an ontology. An important implementation factor is the language used to define the ontology: more expressive languages can significantly increase the runtime complexity of the reasoning process (decreasing the utility of the system in an applied context), whereas less expressive languages may not be able define the necessary rules. The formal specification of Brick+ uses the OWL DL language to define rules for the operation and usage of Brick+ and to achieve the desired runtime properties.

4.3 Brick+ Class Lattice

Brick+ organizes all concepts into a class structure rooted in a small number of high-level concepts. Brick defines this structure as a tree-based hierarchy; Brick+ refines this structure into a *lattice*. Both the lattice and the hierarchy are defined in terms of a "subclass" relationship (§2), but differ in how they define relationships between concepts. A class hierarchy captures how concepts can be

[2]https://www.w3.org/TR/turtle/

Relationship	Definition	Domain	Range	Inverse	Transitive?
hasLocation	Subject is physically located in the object entity	`*`	`Location`	isLocationOf	yes
feeds	Subject conveys some media to the object entity in the context of some sequential process	`Equipment` `Equipment`	`Equipment` `Location`	isFedBy	no
hasPoint	Subject has a monitoring, sensing or control point given by the object entity	`Equipment` `Location`	`Point` `Point`	isPointOf	no
hasPart	Subject is composed – logically or physically – in part by the object entity	`Equipment` `Location`	`Equipment` `Location`	isPartOf	yes
measures	Subject measures a quantity or substance given by the object entity	`Sensor` `Sensor`	`Substance` `Quantity`		no no
regulates	Subject informs or performs the regulation of the substance given by the object entity	`Setpoint` `Equipment`	`Substance` `Substance`		no
hasOutputSubstance	Subject produces or exports the object entity as a product of its internal process	`Equipment`	`Substance`		no
hasInputSubstance	Subject receives the object entity to conduct its internal process	`Equipment`	`Substance`		no

Table 2: List of high-level relationships supported by Brick+.

specialized, but does not encode how these concepts behave and relate to one another. In contrast, a lattice captures how concepts can be composed from sets of properties. This offers greater flexibility in the definition of concepts in Brick+ and facilitates the tag decomposition and mapping to Haystack detailed in §5.

Brick+ has six primary concepts. **`Point`** is the root class for all points of telemetry and actuation. There are six immediate subclasses of `Point` categorized by the high-level semantics of how each point behaves: `Sensor`, `Setpoint`, `Command`, `Status`, `Alarm` and `Parameter`.

Brick+ refines the design of the Brick ontology to differentiate between parameters and setpoints. This avoids conflating the concepts of the minimum and maximum setpoints used in deadband control (such as to configure a thermostat to maintain a temperature within that band) and the minimum and maximum allowed values for a setpoint (for example to place a lower bound on permitted air flow setpoints).

`Equipment` is the root class for the lattice of mechanical equipment used in a building. The Brick+ equipment lattice covers equipment for HVAC, lighting, electrical and water subsystems. Brick+ extends the modeling of equipment in Brick to include how classes of equipment relate to substances and processes in the building.

`Location` is the root class for the lattice of spatial elements of a building. The lattice includes physical elements such as floors, rooms, hallways and buildings as well as logically-defined physical extents such as HVAC, lighting and fire zones.

`Substance` is the root class for the lattice of physical concepts that are measured, monitored, controlled and manipulated by building subsystems. Examples of physical substances are air, water and natural gas. These can be further subclassed by their usage within the building, for example "mixed air" is a subclass of "air" that refers to the combination of outside and return air in an air handler unit.

`Quantity` is the root class for the lattice of quantifiable properties of substances and equipment. Examples of physical properties include temperature, conductivity, voltage, luminance and pressure. Subclassing quantities enables differentiation between types of quantities, such as between `Dry Bulb Temperature` and `Wet Bulb Temperature`.

`Tag` is a root class for the flat namespace of atomic tags supported by Brick+. The majority of these tags are drawn from the Haystack tag dictionary, and are *instances* of the `Tag` class.

4.4 Brick+ Relationships

Relationships express how entities and concepts can be composed with one another; this is key to the consistent and extensible usage of Brick+. For entities – the "things" in a building – composition encapsulates functional relationships such as monitoring, controlling, manipulation, sequence within a process, and physical and logical encapsulation. Concepts are identified by classes and are organized into a lattice by relationships.

As in Brick, relationships in Brick+ exist between a *subject* (the entity possessing the relationship's indicated property) and an *object* (the entity that is the target of the property). Brick+ defines a set of constraints for each relationship to ensure correct and consistent usage between subject and object entities without constraining the application of the relationship to yet unknown scenarios.

All Brick relationships have at least one domain or range constraint determining the allowed classes for the subject or object. Domain constraints limit the class of entities that can be the subject of a relationship; range constraints limit the class of entities that can be the object of a relationship. Brick defines domains and ranges of relationships in terms of classes from the lattice. Brick+ supports these definitions (enumerated in Table 2) and extends them such that domains and ranges can be defined in terms of the properties of the subject and object, rather than which sublattice they belong to. This allows the definition of more fine-grained *sub-relationships* with additional semantics.

For example, as in Table 2, the `feeds` relationship indicates the passage of some substance between two pieces of equipment or between an equipment and a location. If the subject of the `feeds` relationship has the property that it outputs air, then the `feeds` relationship can be specialized to the `feedsAir` sub-relationship.

4.5 Brick+ Tags

Brick+ addresses the consistency and interpretability issues of tag-based metadata by explicitly binding Brick classes to sets of tags. In

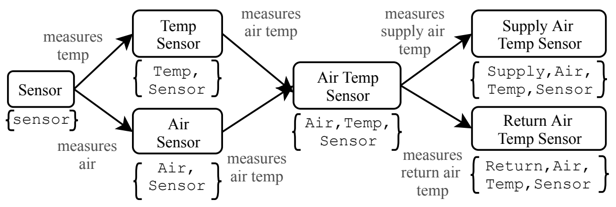

Figure 2: Portion of Brick+ class lattice illustrating the equivalence between tags and classes. The edges indicate which properties are added to each concept (class) to produce a new class. The reverse edges (not pictured) are the subclass relationships.

Brick, classes are human-interpretable because they have clear textual definitions; in Brick+, classes are additionally programmatically-interpretable because they are identified by their position in the class lattice and by the set of properties that define their behavior. Clear definitions promote consistent usage.

Binding classes to tag sets effectively bounds the family of possible tag sets to those that have clear definitions. *This removes the burden of definition, validation and interpretation from the tag structure by outsourcing it to the class lattice, which permits the inference of Brick classes from unstructured Haystack tags.*

Although Brick also defines tags, Brick+ advances the implementation in several ways. Firstly, Brick+ removes the need for tags to be lexically contained within the name of the class (the "TagSet" construct in Brick). This decoupling allows the definition of classes beyond what can be assembled through concatenation of tags, or classes that do not have a straightforward tag decomposition; for example, a Rooftop Unit equipment in Haystack has the `rtu` tag.

Secondly, Brick+ encodes tags so they can be inferred from a Brick class and vice versa, even if a given entity's definition is given only by one or the other. Figure 3 illustrates three different methods for instantiating an `Air Temperature Sensor` demonstrating the flexibility of the Brick+ implementation. The classification of an entity can be performed *explicitly* using the `a` or `rdf:type` predicate in conjunction with a Brick class, *implicitly* through annotating an entity with the set of tags equivalent to a Brick class, *descriptively* by annotating an entity with its behavioral properties, or through a combination of these.

Figure 2 illustrates how tags, classes and properties define the lattice for some subclasses of the `Sensor` class. Figure 4 shows the implementation of the `Supply Air Temperature Sensor` class: line 2 defines how `Supply Air Temperature Sensor` figs into the Brick class lattice. Lines 4-17 defines the `Supply Air Temperature Sensor` class as equivalent to entities that have the `sensor`, `temperature`, `air` and `supply` tags. Lines 18-25 define the `Supply Air Temperature Sensor` class as equivalent ot entities that measure the `Temperature` property of the `Supply Air` substance.

4.6 Brick+ Substances and Quantities

Brick+ defines a lattice of substances and quantities that can be used to describe the functionality of equipment and points. This permits inference of more fine-grained semantic information from existing

```
# instantiate class explicitly
:sensor1    a       brick:Air_Temperature_Sensor .

# instantiate a class implicitly through application of tags
:sensor1    brick:hasTag    tag:Air .
:sensor1    brick:hasTag    tag:Temp .
:sensor1    brick:hasTag    tag:Sensor .

# combination of explicit class and  tags
:sensor1    a       brick:Temperature_Sensor .
:sensor1    brick:hasTag    tag:Air .

# instantiation from behavior
:sensor1    a       brick:Sensor .
:sensor1    brick:measures      brick:Air .
:sensor1    brick:measures      brick:Temperature .

# alternative instantiation from behavior
:sensor1    a       brick:Temperature_Sensor .
:sensor1    brick:measures      brick:Air .
```

Figure 3: Five equivalent methods of declaring `sensor1` to be an instance of the Brick `Air Temperature Sensor` class.

```
1  brick:Supply_Air_Temperature_Sensor a owl:Class ;
2      rdfs:subClassOf brick:Air_Temperature_Sensor ;
3      owl:equivalentClass [
4          owl:intersectionOf (
5              [ a owl:Restriction ; owl:hasValue tag:Sensor ;
6                  owl:onProperty brick:hasTag ]
7              [ a owl:Restriction ; owl:hasValue tag:Temperature ;
8                  owl:onProperty brick:hasTag ]
9              [ a owl:Restriction ; owl:hasValue tag:Air ;
10                 owl:onProperty brick:hasTag ]
11             [ a owl:Restriction ; owl:hasValue tag:Supply ;
12                 owl:onProperty brick:hasTag ]
13         ) ], [
14         owl:intersectionOf (
15             [ a owl:Restriction ; owl:hasValue brick:Temperature ;
16                 owl:onProperty brick:measures ]
17             [ a owl:Restriction ; owl:hasValue brick:Supply_Air ;
18                 owl:onProperty brick:measures ]
19         ) ] .
```

Figure 4: OWL DL-compatible definition of the Brick `Supply Air Temperature Sensor` class showing the explicit class structure, tag equivalence and the use of substance and quantity classes to model behavior

Brick models and allows equipment and points to be classified by their behavior rather than by explicit classification.

The Brick+ substance class lattice is based upon the hierarchy developed by Project Haystack. It classifies substances by phase of matter (`Gas`, `Liquid`, `Solid`) and supports substances qualified by their usage within a process: `Air` is a subclass of `Gas`, and `Outside Air` and `Mixed Air` are subclasses of `Air`. This construction can be extended to include new substances and subclasses of those substances as used in different processes.

A key principle of the Brick+ implementation is every property associated with a class must be inferrable from instances of that class. Properties associated with classes include the set of tags that are equivalent to the class (indicated by the `hasTag` relationship) and the behavioral annotations of the class (indicated by relationships like `measures`).

5 BRICK+ IMPLEMENTATION

Recall that the Brick+ class lattice models concepts by their behavior and related tags as well as by explicit subclass relationships. The lattice is defined by a family of relationships, supported by a set of constraints that ensure correct and consistent usage between

```
1  id: 'd83664ec RTU-1 OutsideDamper'      6  outside: ✓
2  air: ✓                                   7  point: ✓
3  cmd: ✓                                   8  regionRef: '67faf4db'
4  cur: ✓                                   9  siteRef: 'a89a6c66'
5  damper: ✓                               10  equipRef: 'd265b064'
```

(a) Original Haystack entity from the Carytown reference model

```
1  :d83664ec      brick:hasTag     tag:Command . # cmd
2  :d83664ec      brick:hasTag     tag:Damper .
3  :d83664ec      brick:hasTag     tag:Outside .
4  :d83664ec      brick:hasTag     tag:Point .
```

(b) Intermediate RDF representation of the Haystack entity; Haystack software-specific tags (e.g. cur, tz) are dropped.

```
1  :d83664ec_point     brick:hasTag      tag:Damper .
2  :d83664ec_point     brick:hasTag      tag:Command .
3  :d83664ec_point     a                 brick:Damper_Position_Command . # inferred
4  :d83664ec_equip     brick:hasTag      tag:Air .
5  :d83664ec_equip     brick:hasTag      tag:Outside .
6  :d83664ec_equip     brick:hasTag      tag:Damper .
7  :d83664ec_equip     a                 brick:Outside_Damper . # inferred
8  :d83664ec_point     brick:isPointOf      :d83664ec_equip . # inferred
9  :d83664ec_point     brick:isPartOf       :d265b064 # inferred
```

(c) Brick inference engine splits the entity into two components: the explicit point and the implicit outside damper equipment.

Figure 5: The three stages of inferring a Brick model from a Haystack model.

subject and object entities without constraining the application of the relationship to yet unknown scenarios. This enables Brick+ to define a formal inference from Haystack's informal tags to formal Brick classes.

To facilitate the development, testing and debugging of Brick+, we created a Python framework that interprets a structured and extensible abstract ontology specification into a Turtle-based implementation. The framework is open source but its discussion is beyond the scope of this paper.

This section presents an overview of the implementation of the Brick+ ontology with a focus on the implementation of substances and the inference procedure for converting Haystack tags to Brick classes.

5.1 Substance Implementation

The inferred properties concerning substances are more complex to account for the differences in usage: Because substances are classes, it is possible to associate instances of substances with Brick entities. This is useful for applications that want to model how entities behave in relation to the same substance instance. For example, a `Mixed Air Temperature Sensor` and a `Mixed Air Damper` could be related through their respective measurement and regulation of the same instance of `Mixed Air`. However, if a shared instance is not given in the definition of the Brick model, an OWL DL reasoner cannot infer the instantiation of an appropriate substance. Brick+ solves this with *punning* [32].

Punning is a mechanism by which classes can be treated as the canonical instance of their own class. This means that an OWL DL reasoner can relate a punned substance to a property of an equipment or point. Importantly, this does not prohibit the instantiation of substance instances if and when a Brick model supplies those. Line 15 of Figure 3 contains an example of an inferred substance for instances of the `brick:Air_Temperature_Sensor` class.

5.2 Inference Procedure

In order to apply the Brick+ inference to Haystack entities, some preprocessing is required. Firstly, the engine filters out Haystack tags that do not contribute to the definition of the entity, including data historian configuration (`hisEnd`, `hisSize`, `hisStart`), current readings (`curVal`) and display names (`disMacro`, `navName`). Figure 5a shows an example of a "cleaned" Haystack entity containing only the marker and `Ref` tags from the Carytown reference model.

Next, the engine transforms the Haystack entity into an RDF representation that can be understood by the inference engine. The engine translates each of the marker tags into their canonical Brick form: for example, Haystack's `sp` becomes `Setpoint`, `cmd` becomes `Command` and `temp` becomes `Temperature`. The engine creates a Brick entity identified by the label given by the Haystack `id` field, and associates each of the Brick tags with that entity using the `brick:hasTag` relationship. Figure 5b contains the output of this stage executed against the entity in Figure 5a.

At this stage, the engine naïvely assumes a one-to-one mapping between a Haystack entity and a Brick entity. This is usually valid for equipment entities which possess the `equip` tag, but Haystack point entities (with the `point` tag) may implicitly refer to equipment that is not modeled elsewhere. Figure 5a is an example of a Haystack point entity that refers to an outside air damper that is not explicitly modeled in the Haystack model. The last stage of the inference engine performs the "splitting" of a Haystack entity into an equipment and point.

First, the inference engine attempts to classify an entity as an equipment. The engine temporarily replaces all point-related tags from an entity – `Point`, `Command`, `Setpoint`, `Sensor` – with the `Equipment` tag, and finds Brick classes with the smallest tag sets that maximize the intersection with the entity's tags. This corresponds to the *most generic Brick class*. In our running example, the inference engine would transform the entity in Figure 5b to the tags `Damper`, `Outside` and `Equipment`. There are 12 Brick classes with the `Damper` tag, but only one class with both the `Damper` and `Outside` tags; thus, the minimal Brick class with the maximal tag intersection is `Outside Air Damper`. If the inference engine cannot find a class with a non-negligable overlap (such as the `Equipment` tag), then the entity is not equipment.

Secondly, the inference engine attempts to classify the entity as a point. In this case, the engine does not remove any tags from the entity, and finds the Brick classes with the smallest tag sets that maximize the intersection with the entity's tags. In our running example, the minimal class with the maximal tag intersection is `Damper Position Command`.

Figure 5c contains the two inferred entities output by this methodology. In the case where a Haystack entity is split into an eqiupment and a point, the Brick inference engine associates the two entities with the `brick:isPointOf` relationship (line 10 of Figure 5c). Additionally, the inference engine translates Haystack's `Ref` tags into Brick relationships using the simple lookup-table based methodology established in [4]. The inference engine applies these stages to each entity in a Haystack model; the union of the produced entities and relationships constitutes the inferred Brick model.

Site Name	Haystack Entities	Inferred Brick Entities	% Classified Entities	Unclassified Entities	Avg % Custom Tags per Entity	Unique Custom Tags
1	22	23	86.4%	3	7.4%	4
2	147	168	89.8%	15	5.0%	6
3	149	145	73.8%	39	6.6%	7
4	2183	1755	86.7%	290	17.6%	46
5	6474	6236	93.0%	451	19.5%	41

Table 3: Results of inferring Brick entities from tagged Haystack entities.

Ontology	Inferred Properties	
	(Total)	(Avg per entity)
Brick	122,552	2.94/35.44
Brick+	201,266	4.79/35.55

Table 4: Number of inferred properties for all entities across 104 Brick models in Brick and Brick+.

6 EVALUATION

To evaluate Brick+, we examine how well the Brick+ inference engine is able to extract and classify entities from a set of five Haystack models. Brick+ is open-source and is in the process of being adopted as the authoritative implementation of Brick. The Brick+ ontology, generation framework, source code of the inference engine, and the Haystack dataset are all available online at https://github.com/BrickSchema/Brick.

6.1 Source Haystack Models

We assemble a set of five Haystack models, each consisting of a set of tagged entities. The second column of Table 3 contains the number of entities for each Haystack model. Haystack model 1 is the "Carytown" reference model published by Project Haystack for a 3000 sq ft building in Richmond, VA. Haystack models 2 and 3 are sample Haystack data models with for complex buildings, and thus contain large numbers of specialized and non-standard tags [9]. Haystack models 4 and 5 represent two office buildings on the UC Davis campus. Together, these five Haystack models represent a diverse family spanning small to large buildings, differing numbers of custom tags, and different model modelers.

6.2 Haystack Inference Results

Table 3 contains the results of applying the Brick inference engine to the five Haystack models. Column 3 contains the number of inferred Brick entities. When the inference engine splits Haystack entities into equipment and a point, the number of inferred Brick entities can exceed the number of original Haystack entities

The *% Classified Entities* column indicates the percentage of Haystack entities that were successfully classified by the Brick inference engine; the *Unclassified Entities* column contains the number of entities that were not classified. The majority of unclassified entities were such due to the use of non-standard tags that have no provided definition, and thus were not included in the Brick tag structure. The lowest-performing Haystack model, Site 3, represents a data center and contained a number of specialized lighting, HVAC and datacenter equipment and points that are not covered by the existing Haystack tag dictionary.

To understand the impact of informal modeling practices on interpretability and consistency, we examine the occurence of non-standard tags in the five Haystack models; the results are contained in the *Avg % Custom Tags per Entity* column and *Unique Custom Tags* column, which shows the number of user-defined tags in each building, showing the same trend. Models 4 and 5 contain a higher incidence of custom tags because they contain detailed representations of HVAC systems, thus requiring additional vocabulary beyond what is defined in Haystack. The required vocabulary includes HVAC concepts not yet defined in Haystack (e.g., `differential` for `differential pressure`) and functional relationships outside the Haystack's scope, such as capturing spatial relationships.

Examination of the Haystack models reveals three patterns of inconsistent tagging. Firstly, the lexical overlap of tags (detailed in Table 1) leads to one tag being used incorrectly in place of another; for example, using `heat` instead of `heating`. Secondly, because there is no notion of a "sufficient" tag set for a concept, several entities have ambiguous interpretations due to partial tagging. For example, several entities have a `differential` tag, but not a `pressure`, `temperature` or other tag to clarify the quantity. Thirdly, the lack of compositional rules resulted in the ad-hoc creation of site-specific "compound" tags: models 4 and 5 use a custom `spMax` tag instead of the Haystack-defined `sp` and `max` tags to differentiate between setpoints and parameters.

6.3 Brick Inference Results

To complete our evaluation of Brick+, we measure the number of properties that can be inferred from the entities in existing Brick models. Because Brick models already have a formal representation, the inference engine does not need to apply the cleaning or splitting phases of the inference procedure (§5) and can rely entirely upon the existing features of the OWL DL reasoner.

We executed the HermiT [12] OWL reasoner on 104 existing Brick models from the Mortar testbed [10] using the existing Brick ontology and our proposed Brick+ ontology, and computed the number of inferred properties. The results are summarized in Table 4: Brick+ was able to infer almost 80,000 more properties than Brick over the 42,681 entities contained in the Brick models. Brick+ was able to infer all the same properties as Brick, but was able to infer tags and behavioral properties as well.

6.4 Discussion

Our results demonstrate that Brick+ is able to infer 73-93% of entities in Haystack models that follow a canonical tagging scheme, and can infer more semantic properties about entities in Brick models than the previous release of Brick. Recall that Brick+'s inference engine does not currently infer all possible classes from a Haystack model; rather, it formalizes a particular *interpretation* and *organization* of Haystack tags applied to entities. Haystack tags in real-world Haystack models are highly idiosyncratic, due in part to site-specific invention of tags to cover concepts and relationships not defined in the Haystack tag dictionary. This suggests that Brick+'s inference engine will not be able to fully classify each Haystack entity without additional automated metadata construction techniques [7, 17]. Our results support this hypothesis: an ontology-based inference engine demonstrates decent performance against the informal Haystack data model, but, as expected, custom tags inhibit inference.

7 CONCLUSION

Interoperability for building applications requires metadata standards that are semantically sound, rich and extensible. Tags provide an intuitive and informal model, but lack rules for composition and validation that enable consistent, interpretable usage. Brick+ constructs a compositional model of metadata where tags are part of a type system with an underlying formalism based on lattice theory. This enables new algorithmic methods for checking validity, consistency and compositional correctness that is necessary for building a new class of scalable and portable building applications.

This paper has presented a qualitative analysis of the popular Haystack tagging system and demonstrated how its ad-hoc nature inhibits the consistent description of building systems. To address these issues, we have introduced Brick+, a refinement of the Brick ontology with clear formal semantics that permits the inference of well-defined classes from unstructured tags. Brick+ helps to bridge the gap between existing ad-hoc, informal metadata practices and interoperable formal systems; this establishes a foothold for the continued co-development of the Brick and Haystack metadata standards.

8 ACKNOWLEDGEMENTS

This research is supported in part by California Energy Commission EPC-15-057, Department of Energy grant EE-0007685, NSF grants CNS-1526841 and CSR-1526237, and the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The opinions expressed belong solely to the authors, and not necessarily to the authors' employers or funding agencies.

REFERENCES

[1] 2018. Project Haystack. http://project-haystack.org/.

[2] American Society of Heating, Refrigerating and Air-Conditioning Engineers. 2018. ASHRAE's BACnet Committee, Project Haystack and Brick Schema Collaborating to Provide Unified Data Semantic Modeling Solution. http://web.archive.org/web/20181223045430/https://www.ashrae.org/about/news/2018/ashrae-s-bacnet-committee-project-haystack-and-brick-schema-collaborating-to-provide-unified-data-semantic-modeling-solution.

[3] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, et al. 2016. Brick: Towards a unified metadata schema for buildings. In *Proceedings of the ACM International Conference on Embedded Systems for Energy-Efficient Built Environments (BuildSys). ACM.*

[4] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, et al. 2018. Brick: Metadata schema for portable smart building applications. *Applied energy* 226 (2018), 1273–1292.

[5] Bharathan Balaji, Chetan Verma, Balakrishnan Narayanaswamy, and Yuvraj Agarwal. 2015. Zodiac: Organizing Large Deployment of Sensors to Create Reusable Applications for Buildings. ACM, 13–22.

[6] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. 2004. *OWL Web Ontology Language Reference.* Technical Report. W3C, http://www.w3.org/TR/owl-ref/.

[7] Arka A Bhattacharya, Dezhi Hong, David Culler, Jorge Ortiz, Kamin Whitehouse, and Eugene Wu. 2015. Automated metadata construction to support portable building applications. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments.* ACM, 3–12.

[8] Alfonso Capozzoli, Marco Savino Piscitelli, Alice Gorrino, Ilaria Ballarini, and Vincenzo Corrado. 2017. Data analytics for occupancy pattern learning to reduce the energy consumption of HVAC systems in office buildings. *Sustainable Cities and Society* 35 (2017), 191–208.

[9] Patrick Coffey. 2019. Project Haystack Example Data Models. http://web.archive.org/web/20190626161742/https://patrickcoffey.bitbucket.io/.

[10] Gabe Fierro, Marco Pritoni, Moustafa AbdelBaky, Paul Raftery, Therese Peffer, Greg Thomson, and David E Culler. 2018. Mortar: an open testbed for portable building analytics. In *Proceedings of the 5th Conference on Systems for Built Environments.* ACM, 172–181.

[11] Jingkun Gao, Joern Ploennigs, and Mario Berges. 2015. A data-driven meta-data inference framework for building automation systems. ACM, 23–32.

[12] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. 2014. HermiT: an OWL 2 reasoner. *Journal of Automated Reasoning* 53, 3 (2014), 245–269.

[13] Ramanathan Guha and Dan Brickley. 2014. RDF Schema 1.1. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/

[14] Dave Hardin, Eric G Stephan, Weimin Wang, Charles D Corbin, and Steven E Widergren. 2015. *Buildings interoperability landscape.* Technical Report. Pacific Northwest National Lab.(PNNL), Richland, WA (United States).

[15] Dezhi Hong, Hongning Wang, Jorge Ortiz, and Kamin Whitehouse. 2015. The Building Adapter: Towards Quickly Applying Building Analytics at Scale. ACM, 123–132.

[16] Marco Jahn, Tobias Schwartz, Jonathan Simon, and Marc Jentsch. 2011. EnergyPULSE: tracking sustainable behavior in office environments. In *Int. Conf. on Energy-Efficient Computing and Networking.* ACM, 87–96.

[17] Jason Koh, Bharathan Balaji, Dhiman Sengupta, Julian McAuley, Rajesh Gupta, and Yuvraj Agarwal. 2018. Scrabble: transferrable semi-automated semantic metadata normalization using intermediate representation. In *Proceedings of the 5th Conference on Systems for Built Environments.* ACM, 11–20.

[18] Ora Lassila and Ralph R Swick. 1999. Resource description framework (RDF) model and syntax specification. (1999).

[19] Adam Mathes. 2004. Folksonomies - Cooperative Classification and Communication Through Shared Metadata. (2004), 14. http://adammathes.com/academic/computer-mediated-communication/folksonomies.html

[20] Natalie Mims, Steven R Schiller, Elizabeth Stuart, Lisa Schwartz, Chris Kramer, and Richard Faesy. 2017. Evaluation of U.S. Building Energy Benchmarking and Transparency Programs: Attributes, Impacts, and Best Practices. (2017). https://doi.org/10.2172/1393621

[21] OSTI. 2016. *The National Opportunity for Interoperability and its Benefits for a Reliable, Robust, and Future Grid Realized Through Buildings.* Technical Report. https://doi.org/10.2172/1420233

[22] Alexandre Passant. 2007. Using ontologies to strengthen folksonomies and enrich information retrieval in weblogs. In *International Conference on Weblogs and Social Media.*

[23] Alexandre Passant and Philippe Laublet. 2008. Meaning Of A Tag: A collaborative approach to bridge the gap between tagging and Linked Data. *LDOW* 369 (2008).

[24] Mary Ann Piette, Girish Ghatikar, Sila Kiliccote, Ed Koch, Dan Hennage, Peter Palensky, and Charles McParland. 2009. *Open automated demand response communications specification (Version 1.0).* Technical Report. Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US).

[25] Samuel Privara, Jiří Cigler, Zdeněk Váňa, Frauke Oldewurtel, Carina Sagerschnig, and Eva Žáčeková. 2013. Building modeling as a crucial part for building predictive control. *Energy and Buildings* 56 (2013), 8–22.

[26] Project Haystack. 2019. Project Haystack Documentation: Defs. http://web.archive.org/web/20190629183024/https://project-haystack.dev/doc/docHaystack/Defs.

[27] Project Haystack. 2019. Project Haystack Documentation: VFDs. http://web.archive.org/web/20190629182856/https://project-haystack.org/doc/VFDs.

[28] Mads Holten Rasmussen, Pieter Pauwels, Christian Anker Hviid, and Jan Karlshøj. 2017. Proposing a central AEC ontology that allows for domain specific extensions. In *2017 Lean and Computing in Construction Congress.*

[29] S Roth. 2014. Open Green Building XML Schema: A Building Information Modeling Solution for Our Green World, gbXML Schema (5.12). (2014).

[30] Jeffrey Schein, Steven T Bushby, Natascha S Castro, and John M House. 2006. A rule-based fault detection method for air handling units. *Energy and Buildings* 38, 12 (2006), 1485–1492.

[31] David Sturzenegger, Dimitrios Gyalistras, Manfred Morari, and Roy S Smith. 2012. Semi-automated modular modeling of buildings for model predictive control. ACM, 99–106.

[32] W3C. [n.d.]. Punning. https://www.w3.org/2007/OWL/wiki/Punning

[33] Rudolf Wille. 1992. Concept lattices and conceptual knowledge systems. *Computers & mathematics with applications* 23, 6-9 (1992), 493–515.

[34] Rudolf Wille. 2009. Restructuring lattice theory: an approach based on hierarchies of concepts. In *International Conference on Formal Concept Analysis.* Springer, 314–339.