Formalizing Tag-Based Metadata with the Brick Ontology

Gabe Fierro [1,*], Jason Koh [2] Shreyas Nagare [3] Xiaolin Zang [3] Yuvraj Agarwal [3] Rajesh K. Gupta [2] and David E. Culler [1]

[1] *Computer Science, UC Berkeley, Berkeley, CA, USA*
[2] *Computer Science, UC San Diego, San Diego, CA, USA*
[3] *Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA*

Correspondence*:
Gabe Fierro
gtfierro@cs.berkeley.edu

ABSTRACT

Current efforts establishing semantic metadata standards for the built environment span academia, industry and standards bodies. 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, a popular tagging system for building metadata, and identify a family of inherent interpretability 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 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.

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

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 [39, 10, 31], modeling [32] and analysis [38, 20] — such approaches do not see widespread use due to the prohibitive cost of configuring their instantiation 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 [26]. Recent studies by the US Department of Energy [27, 18] 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 path 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 [4], industry [1, 37] and standards bodies [35, 3]. As applications developed for the built environment have 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 [4] 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 [3].

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 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 informal practices are not sufficient for the large-scale standardization of *consistent* semantic metadata necessary for the widespread deployment of energy efficiency applications. Consistent metadata 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, the formal representation of metadata enables a greater degree of consistency and consistency on behalf of the model, while enabling a family of supporting tooling that facilitate the production of Brick+ models from existing tag-based metadata, the systematic validation of those models, and the migration of existing Brick models to the proposed Brick+.

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

1.2 Overview

§3 presents an analysis of the systemic interpretability and consistency issues endemic to the Haystack metadata system, motivating the need for formal rules for composition. This is one of the first systematic evaluations of Project Haystack's approach to metadata: how it impacts consistency and to what extent it enables or inhibits semantic interoperability. §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 lattice 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. §8 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. §9 summarizes ongoing and future efforts to integrate the Brick and Haystack metadata standards and concludes.

Since publication of [13], Brick+ has been adopted into the release of Brick v1.1. This paper extends [13] with:

1. a deeper discussion of the challenges in formalizing metadata tags, and how the implementation of Brick+ resolves these issues (§5.2)
2. the design and implementation of a tool for validating usage of the Brick ontology using SHACL, and techniques for defining templates and idioms that assist in Brick usability (§6)
3. the design and implementation of a tool for migrating Brick models from older versions of the Brick ontology to newer versions (§7)

The production and evaluation of the above Brick+ tooling validates the choice of a formalized semantic metadata model. Not only is the tooling straightforward to construct given the Python-based implementation and formal construction, it also presents an opportunity to unify the Brick and Haystack metadata standards beyond a fragile "house of sticks" constructed from idiom and convention.

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 approach 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. These hint at relationships, but are entirely generic; the relationship is understood by convention. 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 [24], 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 2 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 [17] and OWL [7] 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 [16] 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-supervised learning approaches to learn parsing rules for unstructured labels [9, 22], classifying sensors by examining historical timeseries data [19, 15], or by combining timeseries analysis with label clustering [6]. 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 [28, 29] using ontologies to provide structure to tag-based folksonomies [25]. The approaches developed in these works, along with work on formal concept analysis [43] and concept lattices [42], form the theoretical basis for Brick+.

2.5 Relation to Building Information Modeling

Building information modeling (BIM) relates to the data exchanged for the design, construction and commissioning of a building. BIM models contain extensive lists of building assets in addition to 3D geometry, and there is active research into extending the use of BIM for operation and maintenance [40, 44]. However, BIM models lack direct representation of the contextual metadata described by Brick and Haystack [8, 23]. For example, a BIM model can represent a fan and describe its physical properties such as the shape of the blades, but does not explicitly label whether the fan is installed on the supply or return side of an HVAC system. Deriving that information requires traversing the complex objects and relationships that describe the ducts, connectors and other components of the HVAC system [12], which is difficult to do in an automated manner. Despite these difficulties in retrieving the contextual metadata required to run data-driven applications, BIM is largely complementary to Brick and Haystack. Recent work in representing BIM models using an OWL-based ontology [30] will enable well-defined mappings between the Brick+ and ifcOWL ontologies.

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. Here, we present one of the first analyses of the Haystack data model and how its tag-based implementation impacts consistency, interpretability and interoperability.

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: smaller tag sets 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.

	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)

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 using 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.[33]"

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.

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.

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.

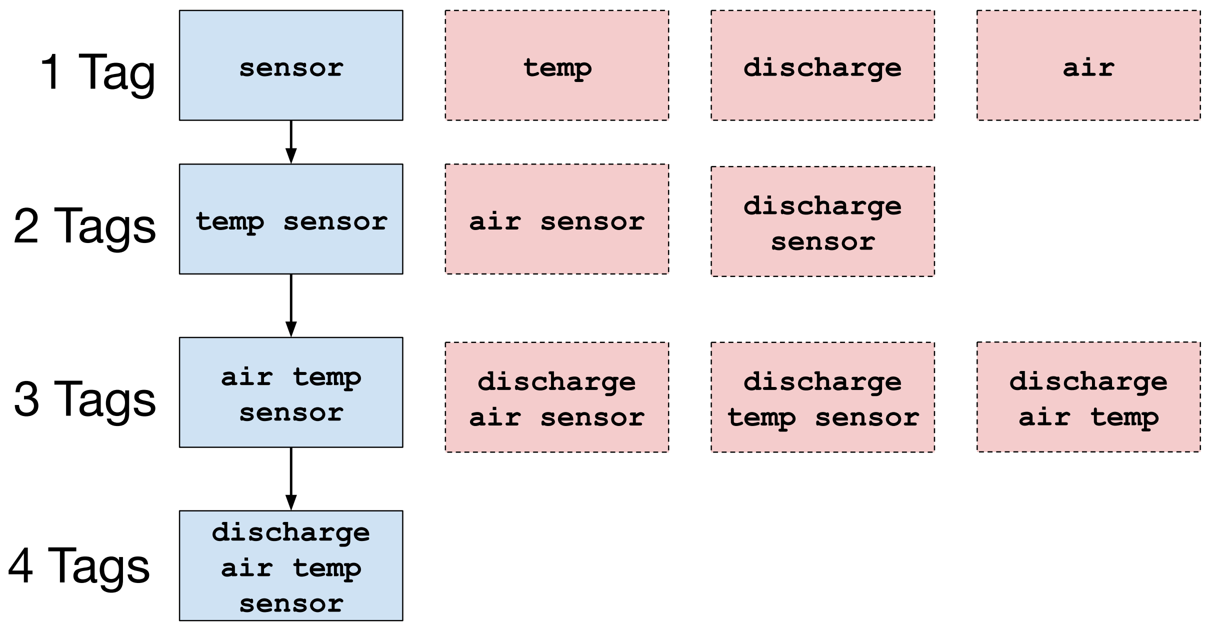

Figure 1. The set of valid (blue + solid outline) and invalid (red + dashed outline) tagsets for a set of four tags. The class hierarchy is established from top to bottom; subclass relationships are indicated by arrows.

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, given by one of `gasHeat`, `hotWaterHeat`, `steamHeat` or `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 meaningfully 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`.[34]"

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 Impact on Consistency

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 [4] establishes a formal class hierarchy and a set of descriptive relationships, it lacks the structure for inference of classes from tags and exhibits 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). In conjunction with the structured implementation (§5), the formal design of Brick+ also enables a suite of supporting tooling for the validation (§6), migration (§7) and construction (§5.3) of Brick+ metadata models.

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 tag 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 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` points are outputs of transducers recording the state of the physical world, e.g. `Air Temperature Sensor`
- `Setpoints` points are control points used to guide the operation of a feedback-driven control system, e.g. `Air Flow Setpoint`

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

- `Command` points are control points that directly affect the state of equipment, e.g. `Fan Speed Command`
- `Status` points report the current logical status of equipment, e.g. `Damper Position Status`
- `Alarm` points are high-priority indicators conveying non-nominal behavior, e.g. `Water Loss Alarm`
- `Parameter` points are configuration settings used to guide the operation of equipment and control systems, e.g. `Max Air Flow Setpoint`.

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, sequencing 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

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 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+.

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 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 2 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.

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

Figure 3. 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

Figure 4 illustrates how tags, classes and properties define the lattice for some subclasses of the `Sensor` class. Figure 3 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.

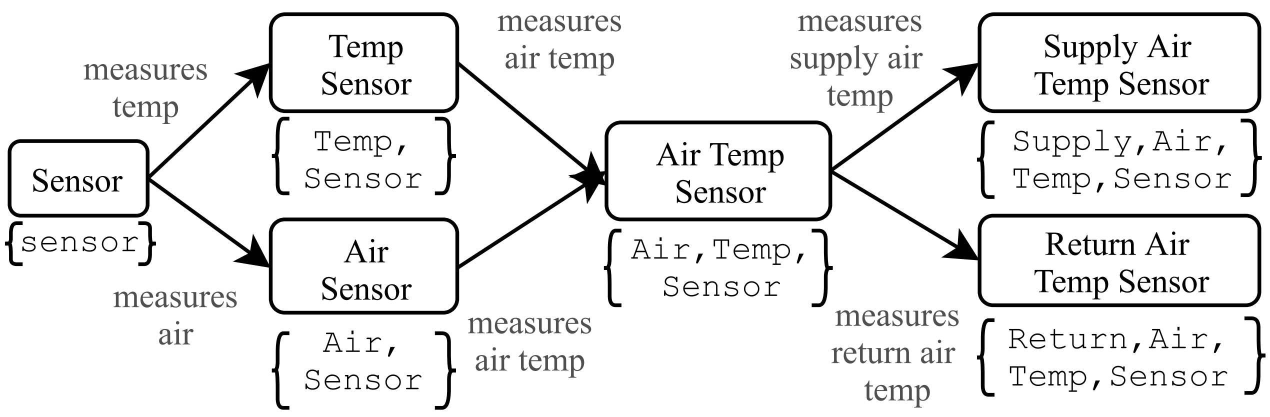

Figure 4. Portion of Brick+ class lattice illustrating the equivalence between tags and classes. Edges indicate which properties are added to each concept (class) to produce a new class. Reverse edges (not pictured) are the subclass relationships.

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 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 which are supported by a set of constraints that ensure correct and consistent usage between subject and object entities without constraining the application of the relationship to yet unknown scenarios. This enables Brick+ to define a formal mapping 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 and available online[3]. The implementation of the inference engine described in §5 and the validation tool described in §6 are published as part of the open source `brickschema`

[3] `https://github.com/BrickSchema/Brick`

Python package[4]. The migration tool described in §7 is distributed as part of the Brick+ source code, and is also open source.

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 association helps applications 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* [41].

Punning is a mechanism by which a class name can represent a canonical instance of that class. This allows 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 2 contains an example of an inferred substance for instances of the `brick:Air_Temperature_Sensor` class.

5.2 Formalizing the Tag-Class Equivalence

There are several possible approaches for formalizing the mapping between tag sets and classes, each with non-obvious tradeoffs. Fundamentally, the formalization has to grapple with the same issue captured in Definition 3.1. The use of the subset relationship between the set of tags describing an entity and the set of tags defining a class can produce ambiguous or erroneous results. In fact, the process described in Definition 3.1 is only sufficient if there is no pair of disjoint classes whose tags are a subset of each other. This implies that under such semantics the only permitted subset relationships between tag sets are those that also obey the subclass relationship between the corresponding classes.

To see why this is the case, consider the following (informal) definitions:

- All `Setpoint` classes are disjoint from all `Parameter` classes; that is, the set of instances of the `Setpoint` type is disjoint with the set of instances of the `Parameter` type.
- The `Air Flow Setpoint` class (a subclass of `Setpoint`) is defined by the tags `air`, `flow` and `setpoint`.
- The `Supply Air Flow Setpoint` class (a subclass of `Setpoint`) is defined by the tags `supply`, `air`, `flow` and `setpoint`.
- The `Max Air Flow Setpoint Limit` class (a subclass of `Parameter`) is defined by the tags `max`, `air`, `flow`, `setpoint` and `limit`.

The tags for `Air Flow Setpoint` are a subset of the tags of *both* `Supply Air Flow Setpoint` and `Max Air Flow Setpoint Limit`. The former is permissible because `Supply Air Flow Setpoint` is a subclass of `Air Flow Setpoint`. However, the subset relationship incorrectly implies that `Max Air Flow Setpoint Limit` is a subclass of `Air Flow Setpoint`, which is a violation of the disjoint relationship between their respective parent classes (`Parameter` and

[4] `https://brickschema.readthedocs.io/`

`Setpoint`). The primary challenge in formalizing the mapping between tag sets and classes is how to overcome this limitation with respect to the following constraints:

1. **Build on the RDF data model and OWL ontology language.** The mapping between tag sets and classes should be expressed in the RDF data model for compatibility with the existing entities, annotations and properties already defined in the Brick+ ontology. Additionally, inferring a class from a set of tags (and vice versa) should be possible through the mechanisms in the OWL ontology language, i.e. via a reasoning engine. This keeps the formalism consistent with the design of the ontology and the mapping to the space of tags that Brick+ incorporates and also minimizes the amount of external machinery required to make use of the developed mappings.
2. **The set of tags mapped to a class should be sensible.** There is a strong usability argument for maintaining a "sensible" set of tags that maps to each class; a "sensible" set of tags is defined as a set of tags that "makes sense." This may include the descriptive terms in the class name or words commonly associated with the class concept. Importantly the interpretability of a "sensible" set of tags does not depend on the large class structure. As long as the set of tags for every class is unique, the mapping developed here is sound.

However, developing a "sensible" mapping that can be expressed in RDF and that is supported by OWL-based inference is a tricky task. Because the Brick+ formal model is built on the set logic of OWL, a naive approach to designing a mapping between tags and classes quickly encounters the tension between tag set overlap and the disjointness of the class hierarchy described above. It is tempting to design the tag sets associated with classes in a manner that sidesteps this issue; for example, one could define the tags associated with `Max Air Flow Setpoint Limit` to be `max`, `air`, `flow` and `limit` (excluding `setpoint`) to avoid the logical violation given by the subsumption of the tags for `Air Flow Setpoint`. This is possible due to the perscriptive nature of the mapping, but it may feel unnatural to users of Brick+ to avoid the `setpoint` tag when describing a concept related to a setpoint, even if the concept itself is not a setpoint.

How, then, can both properties be achieved in the same design? The current understanding of the authors is that implementing a sensible mapping is fundamentally incompatible with a solution expressed *entirely* in the set logic of OWL. A full proof of this assertion is beyond the scope of this paper, but we present the essential intuition in the course of explaining the design and implementation of the Brick+ tag-class mapping below.

The tag-class mapping in Brick+ decouples the informal specification of which tags are associated with which classes from how this mapping is formalized in the ontology. This allows the two representations to evolve independently. The Brick+ generation framework incorporates a list of tags for each defined class. A set of unit tests verifies that there is a unique set of tags for each class in the Brick+ class hierarchy and raises a warning to the developer if this is not the case. Invoking the framework generates Brick+ ontology, including the tag-class mapping. We discuss each direction of the mapping separately below.

Class to Tag Set Mapping: The Brick+ ontology defines a `hasTag` property which associates an entity (an instance of a class) with a tag. The mapping must enable the population of the appropriate `hasTag` properties for each instance of a Brick+ class. The key construct in the modeling approach involves the use of OWL Restriction classes (marked by `owl:Restriction`); these are an OWL construct which defines the members of a class as those that possess certain properties. Brick+ defines a Restriction class for each tag in the ontology that, by definition, counts its members as those entities that possess the given tag. The Restriction classes enable a semantic reasoner to generate a `hasTag` property with a given tag

```
 1 # OWL Restriction definition
 2 _:has_Temperature a owl:Restriction ;
 3     owl:hasValue tag:Temperature ;
 4     owl:onProperty brick:hasTag .
 5
 6 # model statement
 7 example:thing   a   _:has_Temperature .
 8
 9 # generated by the reasoner
10 example:thing   brick:hasTag    tag:Temperature .
```

Figure 5. An example of an OWL Restriction class encoding the association with the `Temperature` tag in Brick.

for each member of the class. Figure 5 demonstrates this mechanism: lines 2-4 contain the definition of a Restriction class for the `Temperature` tag. Line 7 declares an entity as an instance of the Restriction class; this allows a semantic reasoner to produce the association with the tag as captured on line 10.

Brick+ represents the set of tags associated with a class as the intersection of each of the constituent Restriction classes. The intersection is realized as a class formed by the `owl:intersectionOf` property. Lines 3-11 of Figure 3 contain an example of this construction. By definition, the intersection class has as its members the set of entities that fulfill the requirements of each of the constituent tag Restriction classes — that is, the set of entities that have the required tags. This construction means that any entities declared to be instances of this intersection class can, through the application of a semantic reasoner, have the required tags automatically associated.

The last piece of the implementation involves how to associate a Brick+ class with the anonymous intersection class that represents the set of tags (referred to as a *tag set class*). There are two available approaches: marking the Brick+ class as a *subclass* of the tag set class, or marking the Brick+ class as *equivalent* to the tag set class. Marking the tag set class as *equivalent* (using the `owl:equivalentClass` property) is not sufficient because it captures entities with a superset of the required tags; this is exactly the logical violation problem described above. Therefore, we use the other approach. Brick+ encodes a Brick+ class as a subclass (using the RDFS `subClassOf` property) of the tag set class. This means that an entity declared to be a member of a Brick+ class will inherit membership of each of the Restriction classes in the tag set class construct; the application of a semantic reasoner can then populate the required tag associations.

Tag Set to Class Mapping: Encoding the mapping from a set of tags to a Brick+ class requires reasoning about what tags an entity does *not* have in addition to which tags it does have. This is incompatible with the *open world assumption* [36], which is employed by the underlying set logic of OWL. Informally, the open world assumption tells us that a statement may be true regardless of whether or not that statement is included in a given knowledge base. In the context of tag-class mapping, the open world assumption means that the *absence* of a tag on an instance cannot be interpreted as the instance not having that tag; that statement may exist elsewhere. A deeper discussion about the utility of the open world assumption in the context of buildings — where a knowledge base may indeed be the authoritative data source — is beyond the scope of this paper and is the subject of ongoing work. To circumvent this issue, the tag set to

```
id: 'd83664ec RTU-1 OutsideDamper'
air: ✓
cmd: ✓
cur: ✓
damper: ✓
outside: ✓
point: ✓
regionRef: '67faf4db'
siteRef: 'a89a6c66'
equipRef: 'd265b064'
```

Figure 6. Original Haystack entity from the Carytown reference model

```
:d83664ec       brick:hasTag    tag:Command . # cmd
:d83664ec       brick:hasTag    tag:Damper .
:d83664ec       brick:hasTag    tag:Outside .
:d83664ec       brick:hasTag    tag:Point .
```

Figure 7. Intermediate RDF representation of the Haystack entity; Haystack software-specific tags (e.g. `cur`, `tz`) are dropped.

```
:d83664ec_point      brick:hasTag    tag:Damper .
:d83664ec_point      brick:hasTag    tag:Command .
:d83664ec_point      a         brick:Damper_Position_Command . # inferred
:d83664ec_equip      brick:hasTag    tag:Air .
:d83664ec_equip      brick:hasTag    tag:Outside .
:d83664ec_equip      brick:hasTag    tag:Damper .
:d83664ec_equip      a               brick:Outside_Damper . # inferred
:d83664ec_point      brick:isPointOf      :d83664ec_equip . # inferred
:d83664ec_point      brick:isPartOf       :d265b064 # inferred
```

Figure 8. Brick inference engine splits the entity into two components: the explicit point and the implicit outside damper equipment.

class mapping in Brick+ is accomplished using a simplified version of the inference procdure described in §5.3. The implementation is captured in the external `brickschema` Python package[5].

5.3 Brick-Haystack 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 6 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 7 contains the output of this stage executed against the entity in Figure 6.

[5] `https://brickschema.readthedocs.io/`

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 6 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 7 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 8 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 8). Additionally, the inference engine translates Haystack's `Ref` tags into Brick relationships using the simple lookup-table based methodology established in [5]. 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.

6 BRICK+ VALIDATION

While the formal definition of the Brick+ ontology enables the exact specification of concepts and the relationships between them, it does not directly provide a means for validating correct or idiomatic usage within a model. Recall that a *Brick+ model* (sometimes referred to as an *instance* of Brick+) is an RDF graph that uses the Brick+ ontology to represent and describe the entities and relationships within a building.

We begin by defining *correct* and *idiomatic* usage of the Brick+ ontology and provide examples of where the need for such validation arises in practice. We then show how we apply the SHApes Constraint Language to this task by defining "shapes" that encode correct and idiomatic practices. Finally, we describe the implementation of the Brick+ validation library and tool.

The implementation and execution of Brick+ validation is made possible because of the formalization of the underlying model. In Brick and Haystack, the lack of formal rules means that there is no specification of what "correct" or "idiomatic" usage looks like — this must instead be determined through forum posts and human-readable documentation. Brick+ advances the state-of-the-art of building metadata by enabling such a validation process to be performed and in an automatic manner.

```
 1 brick:isPointOf a owl:ObjectProperty ;
 2     rdfs:domain brick:Point ;
 3     owl:inverseOf brick:hasPoint .
 4
 5 brick:Point a owl:Class ;
 6     owl:disjointWith brick:Equipment .
 7
 8 building:ahu1    a   brick:Equipment .
 9 building:vav1    a   brick:Equipment .
10 building:ahu1   brick:isPointOf building:vav1 .
```

Figure 9. An example of a logical violation in an instance of a Brick+ model.

6.1 The Role of Validation

We define *correct* usage of an ontology to mean that the terms and properties defined in the ontology are used appropriately within an instance and do not result in any logical violations of the formal model. For example, consider the set of RDF statements in Figure 9. Lines 1-6 are a partial implementation of the Brick+ ontology. Lines 1-3 specify that any subject of the `brick:isPointOf` property is implied to be an instance of the `brick:Point` class. lines 5-6 state that the set of instances of `brick:Point` is disjoint with the set of instances of `brick:Equipment`; that is, no entity can be both a Point and an Equipment.

We now turn to the definition of a violating Brick+ model. Lines 8 and 9 define two pieces of equipment (we use the high-level `brick:Equipment` for illustrative purposes; in reality these instances would be members of more specific, descriptive classes within the Brick+ Equipment class lattice). Line 10 introduces the logical violation: the use of the `brick:isPointOf` property implies that `building:ahu1` is a member of `brick:Point` which conflicts with the statement on line 8 that `building:ahu1` is a member of the disjoint class `brick:Equipment`. Note that without external information — such as the domain knowledge that an entity named “ahu1” is likely an air handling unit and thus an equipment — it is impossible to tell which statement is erroneous.

Idiomatic usage of an ontology such as Brick+ differs from *correct* usage in that idiomatic violations are still logically valid. Instead, such violations are failures to meet structural and organizational expectations. The specification of modeling idioms is essential for normalizing the use of an ontology to a higher degree than can reasonably be provided by the ontology definition itself. Because the Brick+ ontology is meant to generalize to many different kinds of buildings, subsystems, equipment and organizations thereof, the ontology definition makes very few statements about what information is *required* to be present in a given building instance for it to be considered valid. Idioms fill this gap by encoding “best practices” of what *should* be contained in a given model.

Modeling idioms are diverse in form because they can fulfill many roles. For example, modeling idioms may include the expectation that

- all VAVs in an instance should refer to an upstream AHU and a downstream HVAC zone
- all VAVs of a particular make and model should have five associated monitoring and control points
- all temperature sensors should be reporting in Celsius

Because modeling idioms are not tied directly to the formal definition Brick+, their enforcement is not a requirement for the usage of Brick+. We expect modeling idioms to be defined, distributed and applied on

```
 1 bsh:isPointOfDomainShape a sh:NodeShape ;
 2     sh:targetSubjectsOf brick:isPointOf ;
 3     sh:message "Subject of isPointOf should be an instance of Point" ;
 4     sh:class brick:Point .
 5
 6 bsh:hasPointRangeShape a sh:NodeShape ;
 7     sh:targetSubjectsOf brick:hasPoint ;
 8     sh:property [
 9         sh:class brick:Point ;
10         sh:message "Object of hasPoint should be an instance of Point" ;
11         sh:path brick:hasPoint ] .
```

Figure 10. SHACL shapes defining correct usage of the `brick:isPointOf` and `brick:hasPoint` relationships

a per-project or per-building basis; in the future, equipment manufacturers may distribute Brick+-encoded modeling idioms for how their equipment should be represented in a Brick+ model.

6.2 Brick+ Validation with SHACL

The abstract Brick specification described in §5 lends a means to generate constraints enforcing correct and/or idiomatic usage. These constraints are defined using the SHApes Constraint Language (SHACL [21]), a W3C standard for validating RDF graphs against a set of conditions or constraints. The SHACL standard comprises a specification for a *shapes graph*, an RDF graph containing the constraint definitions — including but not limited to expected properties, values and types associated with properties and arity of properties — and a method for verifying if a target RDF graph meets those constraints.

A shapes graph contains a collection of *shapes*. A shape consists of a list of constraints and a *target declaration* which specifies which node or group of nodes the constraints apply to. SHACL constraints have many forms; rather than review the full range of possibilities (we refer the reader to [21] for detailed documentation on SHACL), we concentrate on the two main categories of SHACL shapes used in Brick+ validation: relationship shapes and class shapes.

Relationship shapes are constraints that validate use of Brick+ relationships enumerated in Table 2. There is one shape for each domain and range property defined for each Brick+ relationship. The domain and range properties (denoted by `rdfs:domain` and `rdfs:range`) imply the class of the subject and object of the relationship, respectively. Validating against relationship shapes can alert authors of Brick+ models of potential logical violations (see Figure 9).

Figure 10 contains two relationship shape definitions for the inverse relationships `brick:isPointOf` and `brick:hasPoint`. The top shape — `bsh:isPointOfDomainShape`, lines 1-4 — demonstrates the typical structure of a shape constraining the class of a relationship's subject. The implementation in SHACL is straightforward: line 2 indicates that the shape targets all nodes which are subjects of the `brick:isPointOf` relationship. The targeted nodes are called the *focus nodes* in SHACL parlance. Line 4 indicates that the focus node's class should be `brick:Point`.

The bottom shape — `bsh:hasPointRangeShape`, lines 6-11 — demonstrates the typical shape structure for constraining the type of the object of a relationship. Line 7 indicates that the shape targets all nodes which are subjects of the `brick:hasPoint` relationship. The composite structure on lines 8-11 states that the objects of the `brick:hasPoint` relationship should have the class `brick:Point`.

```
bsh:vavRelationshipShape    a    sh:Nodeshape ;
    sh:targetClass  brick:Variable_Air_Volume_Box ;
    sh:property [
        sh:path brick:feeds ;
        sh:message "VAV boxes should feed an HVAC Zone" ;
        sh:class brick:HVAC_Zone ] ;
    sh:property [
        sh:path brick:isFedBy ;
        sh:message "VAV boxes should be fed by an AHU" ;
        sh:class brick:Air_Handler_Unit ] .
```

Figure 11. A SHACL shape encoding the idiom that a VAV must be interposed between an AHU and an HVAC Zone through the use of the `brick:feeds` property

```
bsh:modelXYZ_VAVSHape    a    sh:Nodeshape ;
    sh:targetClass  brick:Model_XYZ_VAV ;
    sh:property [
        sh:path brick:hasPart ;
        sh:message "Model XYZ VAVs must have a damper" ;
        sh:class brick:Damper ] ;
    sh:property [
        sh:path brick:hasPoint ;
        sh:message "Model XYZ VAVs must have a supply air temp sensor" ;
        sh:class brick:Supply_Air_Temperature_Sensor ] ;
    sh:property [
        sh:path brick:hasPoint ;
        sh:message "Model XYZ VAVs must have a heating temp setpoint" ;
        sh:class brick:Heating_Temperature_Setpoint ] ;
    sh:property [
        sh:path brick:hasPoint ;
        sh:message "Model XYZ VAVs must have a cooling temp setpoint" ;
        sh:class brick:Cooling_Temperature_Setpoint ] ;
    sh:property [
        sh:path brick:hasPoint ;
        sh:message "Model XYZ VAVs must have an air flow sensor" ;
        sh:class brick:Supply_Air_Flow_Sensor ] ;
    sh:property [
        sh:path brick:hasPoint ;
        sh:message "Model XYZ VAVs must have an air flow setpoint" ;
        sh:class brick:Supply_Air_Flow_Setpoint ] .
```

Figure 12. A SHACL shape encoding the required parts and points for a theoretical "Model XYZ" variable air volume box.

Class shapes specify conditions on the properties and property values for a Brick+ class. Correspondingly, the focus nodes for a class shape are the set of instances of that class. We expect that most idiomatic shapes will be class shapes.

Figure 11 contains an example of an idiomatic class shape that encodes the requirement that all VAVs in a model instance must refer to a downstream HVAC zone and an upstream air handling unit. Validating a Brick+ model instance against this shape involves examining all of the instances of `brick:Variable_Air_Volume_Box` and its subclasses to see if the required properties exist and if the objects of those properties fulfill the class requirements.

Figure 12 is an example of a shape encoding the expected parts and points for a theoretical variable air volume box of a particular make and model. Applying this shape to a Brick+ model instance can help ensure that all instances of the equipment are modeled consistently.

Unlike the shapes in Figure 10, the shapes in Figure 11 and Figure 12 are not required for correct usage of Brick. Instead, adherence to these shapes might be a commissioning requirement for the Brick+ model produced for a site.

6.3 Implementation

We now describe how we have incorporated the SHACL standard into the development and usage of Brick+.

The abstract specification of Brick+ developed in §5 allows us to automatically generate relationship shapes for verifying correct usage of Brick relationships. There are currently 23 relationship shapes distributed with Brick+, but we expect this number will increase over time as the number of relationships and properties supported by Brick+ expands. These shapes are included as part of the Brick+ distribution and are organized under the abstract RDF namespace `https://brickschema.org/schema/1.1/BrickShape`, commonly abbreviated as `bsh`.

To perform validation of a model instance, we incorporate the excellent open-source PySHACL library [2] into a Brick-specific software module and augment it with some features specific to Brick+. The module exposes the validation functionality through a command-line tool, `brick_validate`, as well as a Python library. Shapes for validating correct usage of Brick+ are included in the library so validation against these shapes is always performed by default, without any additional configuration.

The primary feature offered by the Brick+ validation software module is a post-processing step applied to the output of the underlying PySHACL module. When the PySHACL validation process is complete, the Brick+ validation software attempts to find the offending triples and relevant context within the model instance. The software can then provide suggestions for how to repair the model to pass validation.

6.4 Evaluation of Validation Tool

To evaluate the efficacy of the validation approach and tools, we applied the Brick+ validation module to five reference models from the original Brick release [4]. Each of the models was converted to the Brick+ edition of Brick through the migration process described in §7. The validation process found correctness violations in each graph, including:

- Incorrect type of subject or object as required by the property: this is one type of error that can be found through the application of relationship shapes
- Incorrect use of a relationship; for example, `brick:hasLocation` is used where `brick:hasPart` is more appropriate.
- Using a class declared to be in the Brick+ namespace that is not actually defined in the official Brick+ release: this may not be a severe violation because we do expect that ad-hoc extensions to the Brick+ ontology will take place “in the field”, but it is good to raise a warning that potentially unsupported classes exist in a model instance
- Failure to declare a type for an entity: this is an example of a correctness constraint that does not fall under the relationship shapes defined above

Validation of Brick model instances has long been a desired feature, but has been difficult to implement due to the lack of formalization of the Brick model itself. The introduction of Brick+ and its abstract specification makes description of correct and idiomatic usage natural to express as “shapes” within the

SHACL language. The validation of a Brick+ model instance using these shapes is simple to perform through the Brick+ software library. The Brick+ shapes and the validation process are captured online at `https://brickschema.readthedocs.io/en/latest/validate.html`.

7 BRICK MODEL MIGRATION

As the Brick ontology evolves it becomes increasingly important to handle the migration of a particular building model from one version to another. The migration should fulfill the following properties:

- **Complete**: The migration should handle the translation of *all* classes and relationships from one version of the ontology to another.
- **Semantics-preserving**: The migration should preserve the semantics of the original model when updating it to the new ontology wherever possible; the extent to which this can be fulfilled is determined in part by how well the ontology itself preserves the semantics of the older version.
- **Automatic**: The migration should minimize the amount of input and manual translation effort required of the model developer.

In this section, we present the design, implementation and evaluation of a tool for migrating Brick model developed against the prior Brick 1.0.x ontology versions [4, 5] to the Brick+ ontology developed in this paper.

While the older versions of Brick have more structure than Haystack, we can still adopt a similar approach for formalizing the relationship between Brick concepts and Brick+ concepts. Both the Brick-migration described in this section and the Haystack-inference described in §8 describe how these non-formal metadata standards can be defined in terms of the formal Brick+ definition.

7.1 Migration Strategies for Brick

We adopt two strategies for migrating models developed against Brick to the newer Brick+ ontology: migration of classes and migration of relationships.

Class migration consists of referring instances of Brick classes to the most appropriate Brick+ class. Most of the class names stayed the same between Brick and Brick+, meaning the migration can be performed through a simple 1-to-1 mapping of namespaces. Cases where the name of the class changed while the role and definition stayed the same are handled through the same mechanism.

For cases where there is not a 1-to-1 mapping between classes in the two versions, we adopt a *parametric* approach to migration. The most common case where 1-to-1 mapping fails is that of so-called “equipment-flavored” classes. The original Brick class structure included many class names — the majority of them Point classes — that incorporated the name of equipment. For example, the Brick class `AHU_Zone_Air_Temperature_Sensor` represents the concept of an `Zone_Air_Temperature_Sensor` associated with an air handling unit. The existence of this class raised an issue for practitioners: should they use the `Zone_Air_Temperature_Sensor` class with an `brick:isPointOf` relationship to an instance of the `Air_Handling_Unit` class, or should they simply use the `AHU_Zone_Air_Temperature_Sensor` class? Brick+ addresses this issue by eliminating all “equipment-flavored” classes, preferring the explicit association of points to equipment through the `brick:isPointOf` relationship.

In order to preserve the semantics of the “equipment-flavored” classes, the migration tool must go beyond simply translating class names and now must add relationships as well. The migration tool adds the requisite relationships where the instance of the “equipment-flavored” point class already had a relationship

```
PREFIX brick_v_1_0_3: <https://brickschema.org/schema/1.0.3/Brick#>
PREFIX brick: <https://brickschema.org/schema/1.1/Brick#>
DELETE {
   ?subject ?predicate brick_v_1_0_3:AHU_Zone_Air_Temperature_Sensor
} INSERT {
   ?subject ?predicate brick_plus:Zone_Air_Temperature_Sensor
} WHERE {
   ?subject  ?predicate  brick_v_1_0_3:AHU_Zone_Air_Temperature_Sensor
}
```

Figure 13. An example of a SPARQL 1.1 UPDATE query migrating a Brick 1.0.3 class to a Brick+ class.

to an instance of the appropriate equipment class. When the instance does *not* have a relationship to an equipment instance, the migration module can either generate a temporary placeholder instance of the equipment or raise a flag to the user to indicate the lack of one.

The change in class structure from a strict hierarchy (Brick) to a lattice (Brick+) is transparent to the building models and thus does not need to be addressed by the migration process.

Relationship migration consists of replacing Brick relationships with the most appropriate Brick+ relationship. Brick+ preserves the relationships defined in Brick, so the migration tool only needs to handle the translation of the namespace. Although Brick+ incorporates some new relationships, these are either used solely within the definition of the class lattice (e.g. `brick:measures`), are not yet used by Brick instances, or can be added automatically (e.g. `brick:hasTag`); therefore, the migration tool does not need to handle these new relationships.

7.2 Implementation

The migration tool is implemented in Python and is included as part of the open source Brick+ distribution[6]. The tool includes a set of *conversion queries* that implement the translation from one version of Brick to another. A conversion query is phrased in the SPARQL 1.1 UPDATE language and is generated from an underlying dictionary of the simple and parametric migrations described above. Figure 13 contains an example of a conversion query. Each conversion query is parameterized by the source and destination version of Brick; the query converts a given term from the source version of Brick to its migrated form in the destination version. The conversion queries are incorporated into the migration algorithm, which consists of the following steps:

1. Accept as input the source model, the source version of Brick, and the intended output version of Brick.
2. If the migration tool contains a set of conversion queries for the provided source and destination version, then continue with the *direct* translation. Otherwise, perform an *indirect* translation (described below).
3. Ingest the source model into a database (such as a triplestore) that supports the required SPARQL 1.1 queries
4. Execute all of the conversion queries against the triple store
5. Serialize the edited model to the provided output file

[6] `https://github.com/BrickSchema/Brick`

Model Name	Classes	% Translated	Relationships	% Translated
Soda	34/34	100%	7/7	100%
GHC	79/80	98.75%	8/8	100%
Rice	57/57	100%	5/5	100%
EBU3B	217/217	100%	3/3	100%
GTC	570/582	97.94%	8/9	88.89%

Table 3. The completeness of the migration tool against five Brick reference models in terms of the *unique classes and relationships* in the source model.

Model Name	Classes	% Translated	Relationships	% Translated
Soda	1693/1693	100%	2078/2078	100%
GHC	9103/9112	99.90%	36458/36458	100%
Rice	632/632	100%	718/718	100%
EBU3B	6174/6174	100%	8392/8392	100%
GTC	1524/1526	99.21%	5309/5311	99.99%

Table 4. The completeness of the migration tool against five Brick reference models in terms of the *instances of classes and relationships* in the source model.

The migration tool maintains an RDF graph containing the details of all available conversions. In the case where a direct migration is not available, the migration tool runs a shortest-path algorithm to determine a sequence of intermediate versions through which the source model can be migrated so as to arrive at the desired output version. Currently, the shortest-path algorithm uses the number of intermediate versions as the distance metric.

7.3 Evaluation of Migration Tool

To evaluate the efficacy and completeness of the migration tool, we apply it to five of the original Brick reference models published as part of [4] to translate them from version 1.0.2 to Brick+.

The results of applying the migration tool to the source models are enumerated in Tables 3 and 4. The migration tool successfully converts 98% of the unique classes used in the models and 91% of the relationships. The unmapped classes — those for which no conversion query existed — were left as-is: no information was lost from the original models. These unmapped classes exist where the model authors defined their own extensions to Brick.

This evaluation demonstrates that the migration tool is effective in handling the translation of classes and relationships between past and current version of Brick. We believe that the methodology developed here will allow the migration tool to perform effective migrations of models through future versions of Brick.

8 HAYSTACK–BRICK+ INFERENCE

To further evaluate how well a formal approach to metadata enables consistency, we examine how well the Brick+ inference engine is able to extract and classify entities from a set of five Haystack models.

8.1 Source Haystack Models

We assemble a set of five Haystack models, each consisting of a set of tagged entities. 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 [11]. 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.

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 5. Results of inferring Brick entities from tagged Haystack entities.

8.2 Haystack Inference Results

Table 5 contains the results of applying the Brick inference engine to the five Haystack models. 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 data center 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 occurrence 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 the `differential` tag, but do not have a tag to clarify the quantity (e.g. `pressure`, `temperature`). 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.

8.3 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.

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

Table 6. Number of inferred properties for all entities across 104 Brick models in Brick and Brick+.

We executed the HermiT [16] OWL reasoner on 104 existing Brick models from the Mortar testbed [14] using the existing Brick ontology and our proposed Brick+ ontology, and computed the number of inferred properties. The results are summarized in Table 6: 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.

8.4 Discussion of Inference Results

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 [22, 9]. 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.

9 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 presents a qualitative analysis of the popular Haystack tagging system and demonstrates 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.

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`.

10 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.

CONFLICT OF INTEREST STATEMENT

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

AUTHOR CONTRIBUTIONS

The Author Contributions section is mandatory for all articles, including articles by sole authors. If an appropriate statement is not provided on submission, a standard one will be inserted during the production process. The Author Contributions statement must describe the contributions of individual authors referred to by their initials and, in doing so, all authors agree to be accountable for the content of the work.

FUNDING

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.

DATA AVAILABILITY STATEMENT

The datasets generated for this study can be found on GitHub at `https://github.com/BrickSchema/brick-examples`

REFERENCES

[1] [Dataset] (2018). Project Haystack

[2] [Dataset] (2020). PySHACL

[3] [Dataset] 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`

[4] Balaji, B., Bhattacharya, A., Fierro, G., Gao, J., Gluck, J., Hong, D., 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*

[5] Balaji, B., Bhattacharya, A., Fierro, G., Gao, J., Gluck, J., Hong, D., et al. (2018). Brick: Metadata schema for portable smart building applications. *Applied energy* 226, 1273–1292

[6] Balaji, B., Verma, C., Narayanaswamy, B., and Agarwal, Y. (2015). Zodiac: Organizing large deployment of sensors to create reusable applications for buildings (ACM), 13–22

[7] Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., et al. (2004). *OWL Web Ontology Language Reference*. Tech. rep., W3C, http://www.w3.org/TR/owl-ref/

[8] Bhattacharya, A., Ploennigs, J., and Culler, D. (2015). Short paper: Analyzing metadata schemas for buildings: The good, the bad, and the ugly. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments* (ACM), 33–34

[9] Bhattacharya, A. A., Hong, D., Culler, D., Ortiz, J., Whitehouse, K., and Wu, E. (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

[10] Capozzoli, A., Piscitelli, M. S., Gorrino, A., Ballarini, I., and Corrado, V. (2017). Data analytics for occupancy pattern learning to reduce the energy consumption of hvac systems in office buildings. *Sustainable Cities and Society* 35, 191–208

[11] [Dataset] Coffey, P. (2019). Project Haystack Example Data Models. `http://web.archive.org/web/20190626161742/https://patrickcoffey.bitbucket.io/`

[12] Dong, B., Lam, K., Huang, Y., and Dobbs, G. (2007). A comparative study of the ifc and gbxml informational infrastructures for data exchange in computational design support environments. In *Building Simulation 2007, BS 2007*

[13] Fierro, G., Koh, J., Agarwal, Y., Gupta, R. K., and Culler, D. E. (2019). Beyond a house of sticks: Formalizing metadata tags with brick. In *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*. 125–134

[14] Fierro, G., Pritoni, M., AbdelBaky, M., Raftery, P., Peffer, T., Thomson, G., et al. (2018). Mortar: an open testbed for portable building analytics. In *Proceedings of the 5th Conference on Systems for Built Environments* (ACM), 172–181

[15] Gao, J., Ploennigs, J., and Berges, M. (2015). A data-driven meta-data inference framework for building automation systems (ACM), 23–32

[16] Glimm, B., Horrocks, I., Motik, B., Stoilos, G., and Wang, Z. (2014). HermiT: an OWL 2 reasoner. *Journal of Automated Reasoning* 53, 245–269

[17] [Dataset] Guha, R. and Brickley, D. (2014). RDF schema 1.1

[18] Hardin, D., Stephan, E. G., Wang, W., Corbin, C. D., and Widergren, S. E. (2015). *Buildings interoperability landscape*. Tech. rep., Pacific Northwest National Lab.(PNNL), Richland, WA (United States)

[19] Hong, D., Wang, H., Ortiz, J., and Whitehouse, K. (2015). The building adapter: Towards quickly applying building analytics at scale (ACM), 123–132

[20] Jahn, M., Schwartz, T., Simon, J., and Jentsch, M. (2011). Energypulse: tracking sustainable behavior in office environments. In *Int. Conf. on Energy-Efficient Computing and Networking* (ACM), 87–96

[21] Knublauch, H. and Kontokostas, D. (2017). Shapes constraint language (shacl). *W3C Candidate Recommendation* 11

[22] Koh, J., Balaji, B., Sengupta, D., McAuley, J., Gupta, R., and Agarwal, Y. (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

[23] Lange, H., Johansen, A., and Kjærgaard, M. B. (2018). Evaluation of the opportunities and limitations of using ifc models as source of building metadata. In *Proceedings of the 5th Conference on Systems for Built Environments*. 21–24

[24] Lassila, O. and Swick, R. R. (1999). Resource description framework (RDF) model and syntax specification

[25] Mathes, A. (2004). Folksonomies - Cooperative Classification and Communication Through Shared Metadata , 14

[26] Mims, N., Schiller, S. R., Stuart, E., Schwartz, L., Kramer, C., and Faesy, R. (2017). Evaluation of U.S. Building Energy Benchmarking and Transparency Programs: Attributes, Impacts, and Best Practices doi:10.2172/1393621

[27] OSTI (2016). *The National Opportunity for Interoperability and its Benefits for a Reliable, Robust, and Future Grid Realized Through Buildings*. Tech. rep. doi:10.2172/1420233

[28] Passant, A. (2007). Using ontologies to strengthen folksonomies and enrich information retrieval in weblogs. In *International Conference on Weblogs and Social Media*

[29] Passant, A. and Laublet, P. (2008). Meaning of a tag: A collaborative approach to bridge the gap between tagging and linked data. *LDOW* 369

[30] Pauwels, P. and Terkaj, W. (2016). Express to owl for construction industry: Towards a recommendable and usable ifcowl ontology. *Automation in Construction* 63, 100–133

[31] Piette, M. A., Ghatikar, G., Kiliccote, S., Koch, E., Hennage, D., Palensky, P., et al. (2009). *Open automated demand response communications specification (Version 1.0)*. Tech. rep., Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US)

[32] Privara, S., Cigler, J., Váňa, Z., Oldewurtel, F., Sagerschnig, C., and Žáčeková, E. (2013). Building modeling as a crucial part for building predictive control. *Energy and Buildings* 56, 8–22

[33] [Dataset] Project Haystack (2019). Project Haystack Documentation: Defs. `http://web.archive.org/web/20190629183024/https://project-haystack.dev/doc/docHaystack/Defs`

[34] [Dataset] Project Haystack (2019). Project Haystack Documentation: VFDs. `http://web.archive.org/web/20190629182856/https://project-haystack.org/doc/VFDs`

[35] Rasmussen, M. H., Pauwels, P., Hviid, C. A., and Karlshøj, J. (2017). Proposing a central aec ontology that allows for domain specific extensions. In *2017 Lean and Computing in Construction Congress*

[36] Reiter, R. (1981). On closed world data bases. In *Readings in artificial intelligence* (Elsevier). 119–140

[37] Roth, S. (2014). Open green building XML schema: A building information modeling solution for our green world, gbXML schema (5.12)

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

[39] Sturzenegger, D., Gyalistras, D., Morari, M., and Smith, R. S. (2012). Semi-automated modular modeling of buildings for model predictive control (ACM), 99–106

[40] Tang, S., Shelden, D. R., Eastman, C. M., Pishdad-Bozorgi, P., and Gao, X. (2020). Bim assisted building automation system information exchange using bacnet and ifc. *Automation in Construction* 110, 103049. doi:https://doi.org/10.1016/j.autcon.2019.103049

[41] [Dataset] W3C (2007). Punning

[42] Wille, R. (1992). Concept lattices and conceptual knowledge systems. *Computers & mathematics with applications* 23, 493–515

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

[44] Yang, Q. and Zhang, Y. (2006). Semantic interoperability in building design: Methods and tools. *Computer-Aided Design* 38, 1099 – 1112. doi:https://doi.org/10.1016/j.cad.2006.06.003