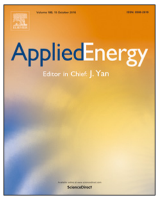
Applied Energy 226 (2018) 1273–1292

Contents lists available at ScienceDirect

Applied Energy

journal homepage: www.elsevier.com/locate/apenergy



Brick : Metadata schema for portable smart building applications

Bharathan Balaji[a], Arka Bhattacharya[b], Gabriel Fierro[b], Jingkun Gao[c], Joshua Gluck[c], Dezhi Hong[d], Aslak Johansen[e], Jason Koh[f,*], Joern Ploennigs[g], Yuvraj Agarwal[c], Mario Bergés[c], David Culler[b], Rajesh K. Gupta[f], Mikkel Baun Kjærgaard[e], Mani Srivastava[a], Kamin Whitehouse[d]

[a] *UCLA, United States*
[b] *UC Berkeley, United States*
[c] *Carnegie Mellon University, United States*
[d] *University of Virginia, United States*
[e] *University of Southern Denmark, Denmark*
[f] *UC San Diego, United States*
[g] *IBM Research, Ireland*

HIGHLIGHTS

- Deployment of building energy applications is inhibited by the heterogeneity of metadata.
- Brick introduces a semantic model to comprehensively describe building infrastructure.
- Schema development based on six commercial buildings, eight energy applications.
- Brick uses standard RDF and SPARQL tools to support machine readability and querying.
- Brick is an open-source effort; we present a software ecosystem for encouraging the adoption of Brick.

ARTICLE INFO

Keywords:
Smart buildings
Building management
Metadata
Schema
Ontology

ABSTRACT

Buildings account for 32% of worldwide energy usage. A new regime of exciting new “applications” that span a distributed fabric of sensors, actuators and humans has emerged to improve building energy efficiency and operations management. These applications leverage the technological advances in embedded sensing, processing, networking and methods by which they can be coupled with supervisory control and data acquisition systems deployed in modern buildings and with users on mobile wireless platforms. There are, however, several technical challenges to confront before such a vision of smart building applications and cyber-physical systems can be realized. The sensory data produced by these systems need significant curation before it can be used meaningfully. This is largely a manual, cost-prohibitive task and hence such solutions rarely experience widespread adoption due to the lack of a common descriptive schema.

Recent attempts have sought to address this through data standards and metadata schemata but fall short in capturing the richness of relationships required by applications. This paper describes *Brick*, a uniform metadata schema for representing buildings that builds upon recent advances in the area. Our schema defines a concrete ontology for sensors, subsystems and the relationships between them, which enables portable applications. We demonstrate the completeness and effectiveness of Brick by using it to represent the entire vendor-specific sensor metadata of six diverse buildings across different campuses, comprising 17,700 data points, and running eight unmodified energy efficiency applications on these buildings.

1. Introduction

Buildings account for 32% of the energy and 51% of the electricity demand worldwide as of 2010 [1]. Improving the energy efficiency of buildings can reduce energy demand by up to 90%, will help reduce operational cost, curb carbon emissions, improve indoor air quality, and keep occupants healthy and productive [1]. Driven by the availability of inexpensive embedded sensing and networking devices, modern buildings are being integrated with a variety of networked sensors and equipment for centralized operation and management.

* Corresponding author.
E-mail address: jbkoh@eng.ucsd.edu (J. Koh).

https://doi.org/10.1016/j.apenergy.2018.02.091
Received 25 September 2017; Received in revised form 11 February 2018; Accepted 12 February 2018
Available online 23 February 2018
0306-2619/ Published by Elsevier Ltd.

Technological innovations in what is now called the “Internet of Things” (IoT) have led to connected lights, power meters, occupancy sensors and electrical appliances that are capable of interfacing with the underlying SCADA (supervisory control and data acquisition) systems used in building automation. These technological improvements hold the promise of significant advances in energy efficient operations [1–3]. For example, research shows that up to 40% of HVAC energy use can be reduced by mitigating faults in these systems [4] and there are hundreds of Automated Fault Detection and Diagnosis (AFDD) algorithms available in the literature that could be used to identify these faults [5]. As of 2012, 14% of the commercial buildings in the U.S. had deployed Building Management Systems (BMS) to manage data collection and remote actuation of the connected building infrastructure [6]. Newer buildings are equipped with BMS by design, and many older buildings are being retrofitted with networked systems for improved efficiency. Furthermore, integration with the Internet presents an exciting possibility for value-creation through a network of buildings that can actively participate in smart grids. Leveraging these technologies a number of innovative software applications have emerged that pose to transform building energy dynamics such as model predictive control [7], automated demand response [8], occupancy based control [9], energy apportionment [10], fault diagnosis [11], participatory feedback [12], and architectural design iterations [13].

These emerging applications present an excellent opportunity for creating an “app store” like those available for smartphones to provide new capabilities to building operators and occupants alike. In this scenario, an energy solution can be deployed across a multitude of buildings containing the requisite infrastructure with minimal configuration. Yet, this vision is far from realization: deploying energy applications in buildings requires significant manual effort and building specific domain expertise. Even the most modern BMS present a cacophony of data and information flows that vary across buildings, vendors and locations. Unlike the mobile phone landscape, there is no standardized operating system or hardware abstraction layer for building applications.

The lack of a common data representation prevents interoperability between buildings and limits deployment of energy applications as developers need to map the heterogeneous data of each building to a common format. This problem has been recognized for a while now. NIST in 2004 estimated that the U.S. building industry lost $15.8 billion annually due to lack of interoperability standards [14]. Attempts have been made to address this problem. Building Information Models (BIM) [15] were introduced to address the interoperability concerns both for the design and operation of buildings. Schemata such as the Industry Foundation Classes (IFC) [16], and more recently the Green Building XML (gbXML) [17], are useful but they remain largely oriented towards design and construction efforts. As a consequence, only limited support is provided for BMS operations, energy management and data analysis. More recently, several other schemata (Project Haystack [18], SAREF [19]) have emerged to highlight the importance and use of building operations *metadata*, i.e., the information that captures the properties of different equipment, sensors and controls used in buildings. Brick builds upon these efforts to devise a practical schema that demonstrates use of several energy applications in a number of buildings across the U.S. and Europe.

The technical challenge here is to design a schema that can, at the very least, capture the information that the building engineers and facilities managers chose to put into real-life deployments across a diverse set of buildings. The schema needs to be expressive enough to capture the contextual information for building subsystems, the sensors installed and the data they generate so that canonical energy applications such as fault detection/diagnosis [20] and demand response [21] can be easily developed and deployed. Recent work has shown that the existing schemata fall short in capturing the important relationships and concepts necessary for applications for even one real building BMS [22].

Designing a comprehensive schema for the emerging IoT universe in order to run *any* conceivable application in *any* context is a difficult task but unnecessary for the current scope of creating a usable platform spanning commercial buildings. Therefore, we focus on creating an information exchange platform focused on commercial buildings where interactions among devices and building spaces are core to sophisticated applications. In developing such a platform, we are guided by the sensors, attributes and relationships that have been shown to be useful in the published literature with a view towards composability and extensibility. In designing Brick, we ask the following important questions and seek answers with demonstrated effectiveness:

- **Completeness:** Can Brick represent all the metadata information (such as a sensor’s location, type, etc.) contained in a building’s BMS?
- **Expressiveness:** Can Brick capture all important relationships between building entities that are (a) explicitly or implicitly mentioned in a building’s BMS, and (b) expressed in canonical energy applications in published academic literature?
- **Usability:** Can Brick represent the information in a way that is easy to use for both the domain expert and the application developer unambiguously? Can the schema support automation with machine readable data formats and querying tools? Can it be extended for new concepts in a unified way?

Due to the highly diverse and changing nature of buildings across the world, these questions can only be answered with a representative sample of current buildings, and it is important for our schema to remain extensible and open in order to accommodate the evolving BMS landscape. Thus, our design of Brick is grounded by the information from BMS across six buildings spread across two continents, comprising more than 630,000 sq-ft of floor space. The information in a BMS is characterized by *data points* that correspond to values reported by sensors, configuration parameters such as a temperature setpoint and status of equipment. Brick design is based on more than 17,700 data points supplied by BMS from six different vendors, and have vastly varying subsystems and sensors. We further refine our design requirements using eight canonical energy applications that require integrated information across commonly isolated building subsystems: air conditioning, heating, lighting, spatial and power infrastructure.

We demonstrate that 98% of BMS data points across our six buildings can be mapped to Brick, and our eight applications can easily query the mapped building instances for required information. We open source the Brick schema files, the BMS metadata from our buildings, the application queries that run on top of Brick and tutorials on how to map existing building metadata to Brick. Brick schema and documentation can be found at http://brickschema.org/.

This paper is based on our earlier work [23] where we presented the initial version of the Brick schema and how it modeled building equipment, locations, sensors and the relationship between them. This paper extends the work by presenting methodologies actually needed for deploying such metadata schema in real systems. First, we show methodologies to instantiate Brick in large scale by exploiting existing information sources including raw point names in building management systems and other schemata as Project Haystack and IFC to ease the adoption of Brick. Second, we propose an architecture for the integration of Brick with actual building operating systems as well as two concrete open-source implementations, XBOS [24] and BuildingDepot 3.0 [25]. Third, We validate the extensibility of our model by our community contribution model and the integrations with other schemata for diverse aspects beyond Brick’s original coverage.

2. Background

2.1. Building applications and energy efficiency

U.S. Department of Energy reports that the commercial sector, mostly accounted by its buildings, consumed 18% of the primary energy and 47% of the electricity in the country in 2017 [26] and that, on average, 30% of this energy is wasted [27]. By taking advantage of the proliferation of sensing and control devices in buildings, many approaches have been developed to reduce this waste, ranging from naïve occupancy-driven control [28] and AFDD [11] to model-predictive control [29]. Because these approaches are, for the most part, algorithms and heuristics that can be implemented in software, we refer to them as building applications. For example, Agarwal et al. developed a building application and showed that occupancy-based-control of HVAC of a single floor in a four floors building reduces 9–15% electrical and 7–12.85% thermal energy consumption of the entire four-story building with minimal hardware installations [28]. If this application was ported to the other 20 buildings on the same campus, the accumulated annual savings would be nearly 40,000 MBTU. Similarly, faults also account for 4–18% of the entire energy consumption in commercial buildings [30], and up to 40% of HVAC energy use [4], which various fault diagnostic algorithms can detect to mitigate.

The largest barrier for the adoption of energy efficient applications is cost [31]. As of 2012, 86% of the buildings in U.S. have no BMS that controls buildings in a centralized way [6]. Adoption of a naïve BMS costs between 2.50 USD to 7.00 USD per square foot as of 2016, which sums up to a minimum of 250,000 USD for a 100,000 square feet building [32]. In a case study of retrofitting a 20,500 square feet medium-sized building in 2010, the labor cost for designing and engineering the system takes the largest portion with 35% other than any other set of devices just to enable scheduled air conditioning [33]. Even the remaining 14% of buildings with BMSes require significant engineering effort even without adding new hardware. Upgrading a BMS or deploying a new application over a BMS requires significant financial investment for paying the BMS vendor. Five of the architectural requirements for BMSes listed by the report are interoperability, scalability, deployment, open, and plug-n-play for the large deployment of energy efficiency applications [33]. These are closely related to metadata organization in buildings addressed by our paper.

2.2. Metadata in buildings

Many large commercial buildings today have monitoring and actuation sensor networks that are accessed through the BMS or through SCADA (Supervisory Control and Data Acquisition) systems. BMSes capture the building infrastructure information in terms of different subsystems such as lighting, electric power, water, and heating, ventilation and air conditioning (HVAC). BMSes describe the equipment in each domain, how they connect with each other, monitor their operation through networked sensors and actuate them through remote commands. BMSes typically have programmable interfaces for higher level control, store historical data and provide visualization. We refer to each of the sensor or control point in the BMS as a “data point”. Their metadata consists of “labels” that describe the many aspects of a data point such as its function, type, location and relationships to different subsystems. Labels in some buildings are simply terse alphanumeric representations, while in other buildings they are long-form and human readable. Typically, these labels are attached to various user interfaces of the specific BMS/SCADA systems, so that engineers and operators can check status and plot trends.

Since BMS metadata information is neither standardized nor designed to be machine readable, “label” naming is heterogeneous and inconsistent across commercial vendors, between the buildings set up by the same vendor, and even within a building. For example, a Zone Temperature Sensor may be referred to as a `ZNT`, `Zone Temperature`, or even by an opaque numerical identifier. Even with programmatic access to labels, data, and other descriptive information, scaling analytics or intelligent control across commercial buildings remains

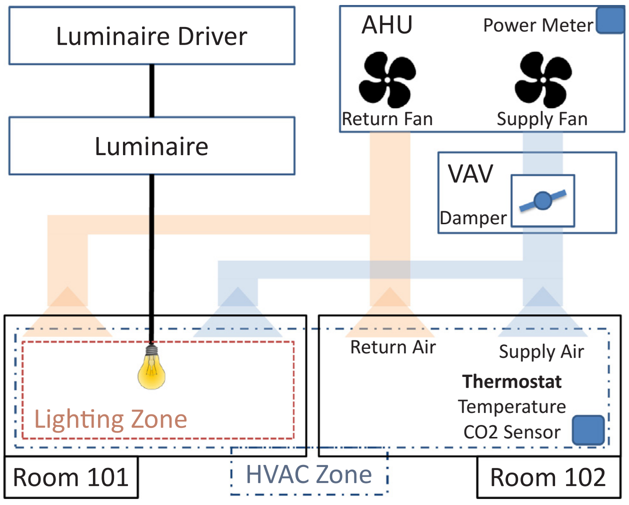


Fig. 1. A simple example building that highlights the components to be modeled in a building schema.

challenging. This is likely to be the case as long as the basic steps in interpreting the metadata involve labor intensive efforts by trained professionals with deep knowledge of building operations and specifics of each building.

Brick directly addresses this problem of building-specific labeling by compiling a normalized list of domain terms that we refer to as our *vocabulary* and devising canonical *relationships* that capture dependencies and connections in and between building subsystems. Brick describes a building in a machine readable format to enable programmatic exploration of different facets of a building. Hence, building managers can represent diverse set of BMS information using Brick, and applications developed based on the Brick schema can be directly deployed on those buildings.

2.3. An example model building

We start with a hypothetical model building to understand the requirements of a uniform building data representation, outlining the current state of the art. Fig. 1 shows the major components of two building subsystems that are commonly found in a modern BMS: HVAC system and lighting system. In the HVAC system, an Air Handler Unit (AHU) supplies conditioned air to a Variable Air Volume Box (VAV), which modulates the air provided to an HVAC Zone consisting of two rooms. The HVAC Zone is a portion of the building that maintains a uniform temperature and uses a thermostat with a temperature and CO_2 sensor for feedback. The luminaire driver in the building only controls the luminaire in the Lighting Zone of Room 101. The lighting system for Room 102 is omitted for readability. Lighting Zones may or may not be overlapped with HVAC Zones as they are defined by different subsystems.

At the very minimum, a schema should be able to model the components illustrated in Fig. 1 as well as their relevant sensors (e.g. temperature sensor) and their related control parameters. Data points and subsystems (such as structures, HVAC, lighting, electrical, and water systems) have complex interrelationships. For example, the AHU in an HVAC system can consist of equipment such as fans, cooling coils, humidifiers, valves and dampers. Each component could have further types; for example, fans could be of type supply fan, return fan or exhaust fan, and each fan could have its associated sensors measuring speed, air flow and power consumption.

2.4. Current state of the art

2.4.1. Project Haystack

Project Haystack [18] aims to address heterogeneity in buildings using *tags* to label different entities, and is the current de facto standard. Using Haystack, the temperature sensor in Fig. 1 is associated with the tags: `[zone, temp, sensor]`. Tags provide a flexible and scalable framework for annotating metadata to building data points. Haystack defines a vocabulary of tags describing building equipment, weather, units and data types. However, the current set of tags lacks or does not fully describe key aspects of buildings such as spatial elements, lighting equipment and electrical subsystems [22].

While deficiency of tags can be addressed by future updates, a subsequent challenge is that use of tags can be ambiguous and inhibit application portability. For example, a user may annotate a temperature sensor as `[zone, temp]` and omit `sensor`. A simple query searching for list of all sensors will not work with this annotation. We cannot enforce the correct grouping of tags to annotate the entities, and hence, users will invariably create multiple variations for the same entity. Therefore, Haystack earns the flexibility of use at the expense of ambiguity in tagging scheme.

Furthermore, Haystack relationships lack the expressive power required of an effective building metadata schema. Haystack represents relationships using a `ref` tag, which is enough to associate two pieces of equipment, but cannot express the nature of that association. For example, a VAV may have an `ahuRef` tag with a value of its parent AHU and a `equipRef` tag with the value of a supply air flow sensor. The `equipRef` tag does not capture whether the supply air flow sensor occurs before or after the VAV in the HVAC system. Further, Haystack does not model "reverse" tags, which makes it difficult to enumerate sequences of equipment.

Project Haystack defines a REST API and a filtering query language.[1] The query language provides basic mechanisms for identifying timeseries points using the associated tags, but does not clearly define a way of traversing `ref` tags for the purpose of exploring the structure of a Haystack model. At the time of writing, there are no open source server implementations available.

Haystack Tagging Ontology (HTO) [34] maps the Haystack tags to an ontology, with each tag corresponding to an ontology class. Thus, HTO is able to combine the flexibility of tags and the formal modeling of ontologies to define essential BMS metadata and the relationships between entities. However, HTO confines the ontology to the defined tags, and does not model the building entities which are a collection of tags (e.g. zone temperature sensor). HTO also does not provide a way to compose complex subsystems in a building and relies on Haystack tagging for mapping raw metadata to the ontology. Brick follows a similar methodology to combine tags and semantic models, but overcomes HTO's limitations with a vocabulary based approach. Thus, Brick provides a direct mapping to the data points and metadata exposed in a BMS and an enriched ontology that can be queried with ontology tools.

2.4.2. Industrial foundation classes

IFC [16] is a standardized Building Information Model (BIM) that developed from the need to have a common exchange model for 3D architectural drawings needed for a building's construction. IFC is good at capturing space-related information such as floors, rooms and zones, but also exhaustively describes the mechanical composition of building subsystems: not just AHUs and VAVs, but also ducts, flanges and other mechanical components not directly measurable or controllable.

IFC lacks much of the vocabulary for describing the necessary subcomponents needed for building operation. Recent versions of the IFC standard include references to generic sensor types (`IfcSensorTypeEnum`) which can be associated with the spaces the sensor covers. However, the IFC standard does not include explicit mechanisms for describing the functional role of sensors, such as whether a temperature sensor measures supply, return or exhaust air. There is also no common way of adding new vocabularies compliant to existing ones. The IFX 2×2 schema also contains descriptors for building controllers,[2] which describe at a high level the existence of alarms, events and schedules.

2.4.3. Semantic web and ontologies

Semantic Web is a framework promoting common data formats, exchange protocols and vocabularies with which machines can process contents' meanings without human interruptions on the Web. Its relevant standards include RDF, SPARQL, and Turtle [35]. It was originally advocated for annotating the Web documents [36], and since then it has seen adoption in multiple domains such as biology [37], IoT [38], and energy management [39] to control the complexity of the domain information.

In Semantic Web, ontologies define formal naming of entities and their properties. An ontology is formally defined as "an explicit specification of a conceptualization [40]. Ontologies are often represented as a directed, labeled graph. Different systems can refer to an ontology for the set of definitions and relationships in a target domain, which helps to reduce ambiguity.

A number of ontologies have been proposed for smart homes and buildings. Most of these ontologies focus on realizing specific applications like controlling things [41], energy management [42], or automated design and operation [43]. Ontology representations of IFC [44] and Haystack [34] also exist. Daniele et al. [19] combined these ontology modeling efforts in collaboration with industry to create a simple but unified model called Smart Appliances REFerence (SAREF). They identify 20 recurring concepts in homes and buildings across these ontologies, and lay out the steps to convert SAREF to a custom ontology. These common concepts, however, do not effectively cover the diversity of devices and equipment in buildings [22] because their goal was to capture generic sensor and smart devices rather than building operations where domain-specific information is required. Brick adopts similar design principles as SAREF, but our vocabulary and concepts are based on ground truth BMS deployments and representative smart building applications and systems.

The BOnSAI [45] smart building ontology describes the functionality of sensors, actuators and appliances as well as how they interact and effect their physical environment. However, they fail to capture the interactions and relationships between the sensors and other building assets. Hence, it lacks a system-level view of the building infrastructure necessary for many applications [22]. Further, the vocabulary does not describe the mechanical or functional compositions of critical building subsystems like HVAC and lighting.

2.4.4. Ontologies and energy management

Energy management systems commonly consist of heterogeneous systems where a standardized way of retrieving and processing complex information is much desired. Tahir et al. present an ontology and actual system supporting decision makers to emulate various electricity generation mixes scenarios [46]. Even though the ontology was not fully evaluated, it shows the importance of having a good schema for applications and the suitability of ontologies as a meta model. Eco-industrial park is a community of manufacturing and service businesses sharing the environment, such as energy, water, and materials, to improve productivity in synergy. Various subsystems from different participants need to cooperate in EIP, which in turn increases the complexity in organizing the required information [39]. They extend a

[1] http://project-haystack.org/doc/Ops.

[2] http://www.buildingsmart-tech.org/ifc/IFC2x4/rc2/html/schema/ifcbuildingcontrolsdomain/content.htm.

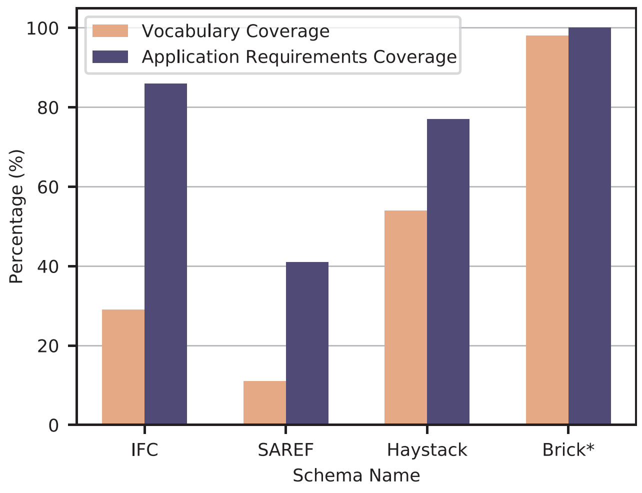


Fig. 2. Comparison of different schemata for buildings [22]. The paper used 89 applications (apps) and three buildings to evaluate IFC, SAREF and Haystack. * We show Brick's result for eight representative apps from each of the eight categories of the 89 apps and six buildings we described in this paper. Two of the buildings are the same as that used by Bhattacharya et al.

couple of existing ontologies for processes and industrial symbiosis with domain vocabularies and necessary relationships for their energy applications. However, none of the existing work in energy management has shown the complete evaluation on the real systems and the extensibility as we present in this paper.

2.4.5. Analysis of existing schemata

Bhattacharya et al. [22] performed a comparison of IFC, SAREF, and Project Haystack. The paper uses 89 building applications among eight categories published in the literature as a baseline to compare different schemata and shows that relationships between different pieces of information are essential to enable interoperability and portability of building applications over three buildings. The paper compares the capabilities of Haystack [18], IFC [16] and SAREF [19] using three metrics to measure the effectiveness of a schema: (i) the ability to completely map BMS metadata from three existing buildings to the schema, (ii) ability of the schema to capture the relationships required by applications, and (iii) the flexibility of the schema to deal with uncertainty as well as their extensibility to new concepts. Fig. 2 presents the comparison across Haystack, IFC, SAREF and Brick for metrics (i) and (ii). We evaluated Brick based on eight representative applications and two of the three buildings used by Bhattacharya et al. Among the three existing schemata, Haystack shows the best vocabulary coverage as it is a tag-based model where tags can be arbitrarily combined. IFC is the most complete in describing application relationships as its model captures the building subsystems and the dependencies between them. SAREF scored the lowest for both metrics because it models the common concepts across different models and systems instead of comprehensively modeling buildings. In comparison, Brick has complete coverage of both vocabularies and application requirements.

Brick builds upon these works in several ways to achieve both extensibility and expressibility. We utilize the tagging concept of Haystack and extend it with mechanisms to model relationships and entities. We use the location concepts from IFC. We use a semantic representation to utilize its flexibility and extensibility properties. The semantics allows us to formalize, restrict, and verify the usage of tags, entities, and relationships.

This paper is an extension of our BuildSys 2016 conference paper [23], covering several significant additions. Specifically, in this manuscript:

- We have updated the Brick schema based on lessons learned and feedback from the community. We have incorporated modeling of resources such as air, water, gas in the schema. We have introduced representation of control sequences of building subsystems in Brick. These capture the dependencies such as how a sensor value is used to adjust the speed of a fan or a position of a valve.
- We present how Brick can be incorporated into a BMS. The Brick model of a building serves to bind contextualized physical resources with logical resources such as API endpoints, timeseries streams, controllers and other processes. Specifically, we examine how Brick has been integrated into two academic BMS: XBOS [24] and BuildingDepot [25].
- We describe a general methodology for creating Brick models from the entities and relationships captured in BMS point names, Haystack models and IFC models. We have developed software to automatically convert Haystack and IFC building models to the Brick schema. We demonstrate their operation on generating Brick models for three buildings.
- We describe our extensibility and collaboration model for Brick. We follow an open source model and have created a framework for discussions, feedback and iterative improvements to the Brick schema. It is our hope to create a community of Brick users to help Brick evolve to meet the needs of application developers, building managers and occupants. We also show integrations with other existing ontologies to augment Brick's functionality without losing the integrity of each model.

3. Schema design

3.1. Design principles

Brick's design focuses on data points, their metadata found in real building deployments and requirements defined by end use applications for operations and managements. Brick is separated into a core ontology defining the fundamental concepts and their relationships as discussed below and a domain specific taxonomy expanding the building specific concepts. This allows users to extend new concepts as well as the taxonomy with the concepts. We obtain ground truth information from six diverse buildings across the US and Europe, which have 17,700 data points and five different vendors in total (Table 4). We pick eight representative application categories from the list of smart building applications compiled by Bhattacharya et al. [22], and formulate metadata queries for these applications to drive the basic requirements of Brick as well as evaluate how well our building metadata can be mapped to Brick. Section 7 contains our findings for the six buildings. We use existing standards in ontology development such as Turtle [47] for data formatting and SPARQL [48] for querying. Users can exploit existing tools such as ontology visualization tools and querying engines.

Brick is distinguished from the other building schemata as follows:

- **Completeness**: The current version of Brick covers the 98% of the vocabularies found in six buildings in different countries. (Section 7)
- **Vocabulary Extensibility**: The structure of Tags/TagSets allow easy extensions of TagSets for newly discovered domains and devices while allowing inferences of the unknown TagSets with Tags. (Section 3.2)
- **Usability**: Brick represents an entity as a whole instead of annotating it. It promotes consistent usages by different actors. Furthermore, its hierarchical TagSets structure allows user queries more generally applicable across different systems. (Section 3.2 and 3.3)
- **Expressiveness**: Brick standardizes canonical and usable relationships, which can be easily extended with further specifications. SPARQL facilitates all the possible combinations of the relationships required by queries of the eights application categories in the literature. (Section 3.4 and 6)

- **Schema Interoperability**: Using RDF enables straightforward integration of Brick with other ontologies targeting different domains or aspects. (Section 11)

3.2. Tags and tagsets

We borrow the concept of *tags* from Project Haystack [18] (Section 2.4) to preserve the flexibility and ease of use of annotating metadata. We enrich the tags with an underlying ontology that crystallizes the concepts defined by the tags and provides a framework to create the hierarchies, relationships and properties essential for describing building metadata. With an ontology, we can analyze the metadata using standard tools and place restrictions to prohibit arbitrary tag combinations or relationships. For example, we can restrict the units of temperature sensors to Fahrenheit and Celsius or prevent sensor and setpoint from occurring together in a tags combination for a data point. An ontology also enables property inheritance in the hierarchy. A subconcept of a concept preserves the original characteristics with more specifications.

We introduce the concept of a *tagset* that groups together relevant tags to represent an entity. With Haystack and related tagging ontologies [34], an entity such as `Zone_Temperature_Sensor` from Fig. 1 is defined by its individual tags, so its properties and relationships with other entities can only be specified at the tag level. A user should assume that the other users would have exactly used zone, temperature, and sensor for annotating the sensor to look for zone temperature sensors. Thus, the way of annotating the same type of sensors in tagging scheme may differ across different buildings. On the contrary, with tagsets based on tags, we have a cohesive concept of a `Zone_Temperature_Sensor` that can be consistently used to represent actual instances of zone temperature sensor. We can further provide its semantics as the temperature is maintained between the zone's `Cooling_Setpoint` and `Heating_Setpoint`. The concept of tagsets works well with an ontology class hierarchy - a `Zone_Temperature_Sensor` is a subclass of a generic `Temperature_Sensor`, and will automatically inherit all its properties. Further, we avoid use of complex tags such as the `chilledWaterCool` and `hotWaterReheat` tags in Haystack. The vocabulary of Brick is defined by its list of tagsets.

3.3. Class hierarchies

We define several high level concepts that provide the scaffolding for Brick's class hierarchy. As the central emphasis of our design is on representing points in the BMS, we introduce *Point* as a class, with subclasses defining specific types of points: Sensor, Setpoint, Command, Status, Alarm. Each point can have several *relationships* that relate the data point to other classes such as its location or equipment it belongs to. Bhattacharya et al. [22] recognize that building metadata has several dimensions, which we carry forward into the design of Brick. We define three dimensions as high level classes to which a Point can be related to: *Location*, *Equipment* and *Resource* (Fig. 3). We define each category as follows:

- Point: Points are physical or virtual entities that generate timeseries data. Physical points include actual sensors and setpoints in a building, whereas virtual points encompass synthetic data streams that are the result of some process which may operate on other timeseries data, e.g. average floor temperature sensor.
- Equipment: Physical devices designed for specific tasks controlled by points belonging to it. E.g., light, fan, AHU.
- Location: Areas in buildings with various granularities. E.g. room, floor.
- Resource: Physical resource or materials that are controlled by equipment and measured by points. An AHU controls resources such as water and air, to provide conditioned air to its terminal units.

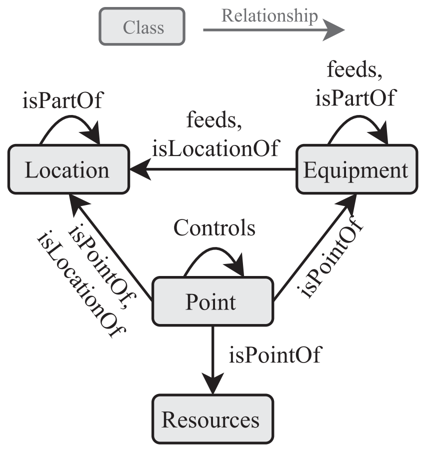


Fig. 3. Information concepts in Brick and their relationship to a data point.

We can expand these concepts in future versions to expand the metadata covered by Brick (e.g. Network). Each concept has a class hierarchy to concretely identify each entity in the building. For example, the Equipment class has subclasses HVAC, Lighting and Power, each of which have their own subclasses. Fig. 4 showcases a sample of Brick's class hierarchy.

It is common in a domain to use multiple terminologies for the same entity. For example, in HVAC systems, `Supply_Air_Temperature` and `Discharge_Air_Temperature` are used interchangeably. We identify these synonyms from our ground truth buildings, and mark the corresponding tagsets as being equivalent classes in Brick. Note that the class hierarchy does not strictly follow a tree structure, and we use multiple inheritance when appropriate. For example, a desk lamp can be a subclass of both the lighting system and office appliance classes.

3.4. Fundamental relationships

Relationships connect the different entities in the building and are essential to providing adequate context for many applications. For instance, to diagnose a VAV, a fault detection application running on our example building (Fig. 1) needs to know the room to which the VAV supplies air, the temperature sensor located in the room, other operational data points in the VAV, and the AHU that provides air to it. However, Bhattacharya et al. establish that current industrial standards lack the ability to sufficiently describe all the relationships required for modern applications [22].

We construct essential relationships by pulling a representative example from each of the eight common application dimensions identified by Bhattacharya et al. [22] as summarized in Table 2. The categories of quintessential relationships we extract from the applications are:

- **Taxonomy:** what class or classes of things define an entity

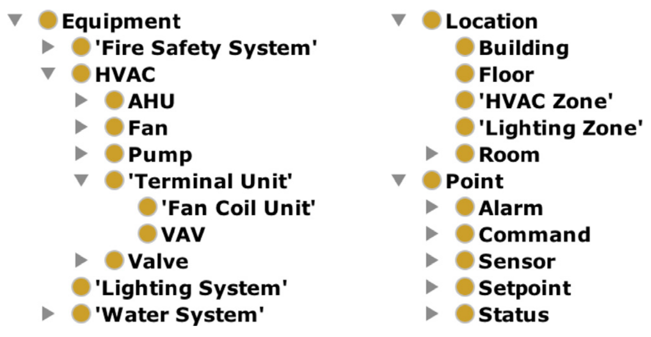


Fig. 4. A subset of the Brick class hierarchy.

- **Location:** which building, floor and room an entity is in, but also where in the room it is
- **Equipment Connections:** what equipment an entity is connected to, and how it is connected
- **Equipment Composition:** what equipment an entity is a part of, or what equipment is a part of it
- **Point Connections:** what points affect the behavior of other points
- **Monitoring:** what measures the entity or what it measures

Portability and orthogonality are two primary concerns in designing the set of relationships. When describing or reasoning about a building, the set of possible relationships between any two entities should be small enough and well-defined such that the correct relationship should be obvious. This *orthogonality* reduces the risk of inconsistency across buildings. Taken to its extreme, orthogonality informs a set of relationships that are specific and non-redundant, which can lead to overfitting the set of relationships for a particular building or subsystem. To support the goal of designing a unified metadata schema across many buildings, these relationships must also be sufficiently generic to be *portable* to many buildings.

Resolving these two tensions leads to the set of relationships listed in Table 1. The specific entities and relationships each application category requires are listed in Table 2. We provide relationships together with their inverse relationships so that users can express them in any direction they prefer. SPARQL queries can accommodate both directions to be compatible with any choices of inverse relationships. The left side of Endpoints column defines the possible subjects and the right side defines the possible objects that the relationship can have, which can provide a guideline for users not to improperly use them. The `isPartOf` relationship captures the compositions among the entities in the building. For example, a room `isPartOf` a floor and a return fan `isPartOf` an AHU. The `feeds` relationship captures the different *flows* between entities such as equipment or locations in the building, such as the flow of air from AHU to VAV, the flow of water from a tank to a tap, or the flow of electricity from a circuit panel to an outlet. Each of these relationships can have sub-properties. For instance, `feeds` can be extended to `feedsAirTo`, `feedsWaterTo`, etc. Fig. 5 shows the relationships for a subset of the example building in Fig. 1.

Brick uses the possible subjects/objects defined in Endpoints column of Table 1 as a guideline when users add relationships. Using ontology property restrictions, we provide rules for certain properties to have precise subjects and objects. For instance, the object of `hasPoint` must be an instance of a class in the `Point` hierarchy. Likewise, the subject of `isLocationOf` must be an instance of a class in the `Location` hierarchy. These can be exploited by a user interface to guide users while tagging raw metadata or while establishing relationships between entities. We define these restrictions as a set of guidelines for Brick model developers to aid in keeping Brick usage consistent between building models.

Table 1
List of the Brick relationships and their definitions. All definitions follow the form `A <relationship> B`, where `relationship` is the first one listed, not the inverse. All Brick relationships are asymmetric, and transitive where marked. If a relationship $\rightarrow$ is transitive, then if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$ is a valid relation. Asymmetric simply means that if $A \rightarrow B$, then $B \rightarrow A$ is invalid.

Relationship (Inverse)	Definition	Endpoints
isLocationOf (hasLocation)	`A` physically encapsulates `B`	Loc./Point Loc./Equip.
controls (isControlledBy)	`A` determines or affects the internal state of `B`	Point/Point
hasPart (isPartOf)	`A` has some component or part `B` (typically mechanical)	Equip./Sensor Equip./Equip. Loc./Loc.
hasPoint (isPointOf)	`A` is measured by or is otherwise represented by point `B`	Equip./Point Loc./Point Resource/Point
feeds (isFedBy)	`A` "flows" or is connected to `B`	Equip./Location Equip./Equip.

3.5. Control sequences

A control sequence is the logic determining an equipment's behavior. Select variables of such logic are exposed as points in building systems. Some points' values are measurements of physical properties, some are results of calculations and others are configuration parameters used to control physical devices. The flow of these control signals is key for understanding buildings operations. Users rely on this to interpret values (e.g., is `Air_Flow_Setpoint`'s value correct given `Zone_Temperature_Sensor`?) or to properly control equipment (e.g., what `point` should I change to achieve certain temperature in this `room`?).

Representations of control logic vary wildly between buildings. In many older buildings, control logic is embedded in physical controllers distributed throughout a building. Some vendors will provide visualization tools to represent their proprietary control logic, and others use proprietary programming languages. Some compelling representations of control logic include Simulink Simscape and Modelica. Simulink Simscape [49] provides multi-domains simulation of exact control logic with mathematical models, but is designed for simulations rather than for integrating with real physical systems. Modelica [50] is an object-oriented language and execution environment for modular simulations and has current development efforts focusing on building control and simulation [51]. These software, however, are only used in simulation and not designed for BMS operation. MLE+ [52] and BCVTB [53] have created co-simulation environments where control simulation logic can be deployed in real buildings with BMS. They are designed for experimental evaluation of control algorithms and are not meant for production operation of buildings.

Brick does not currently attempt to model the control logic in building systems; rather, it describes the dependencies between sensors, actuators, commands, setpoints and related equipment and spaces. We model the control dependencies using the `controls` relationship between points. When a point's value is used for another point's value determination, we say that the former one `controls` the later one. Fig. 6 is an example with a simplified version of VAV control. An AHU provides temperature-controlled air to VAVs, which control their associated zones' temperature by changing the amount of air flow. When the zone's temperature is lower than its corresponding setpoint, the VAV increases the supply air flow controlled by its damper. To be more specific, `Cooling_Command` increases proportionally to the difference between `Zone_Temperature_Sensor` and `Zone_Temperature_Setpoint`. `Cooling_Command` determines `Supply_Air_Flow_Setpoint` and the difference between `Supply_Air_Flow_Setpoint` and `Supply_Air_Flow_Sensor` determines the value for `Damper_Command`. `Damper_Command` affects its damper's state that controls actual air flow. We model these dependencies with `controls` such as "`Zone_Temperature_Setpoint controls Cooling_Command`" and "`Zone_Temperature_Sensor controls Cooling_Command`". We know from the two triples that if we want to change `Cooling_Command`, we have to change `Zone_Temperature_Setpoint`. `Zone_Temperature_Sensor` is not considered as it is a sensor that cannot be controlled arbitrarily.

While the exact mathematical relationships between control points are not included, dependencies modeled as `controls` relations give us enough insights for causal analysis and identify pieces of logic that an application is interested in. To analyze `Cooling_Command` is working properly in terms of control logics, we can easily find what points affect it with `controls` relationships and compare the data from the three data points to find an anomaly. We can also easily find the high-level commands/setpoints to properly control the equipment, which is `Zone_Temperature_Setpoint` in the example. If needed, the Brick

Table 2
This table shows at a high level which entities and relationships are required by each of the eight representative applications.

	Entities	Occupancy Modeling [65]	Energy Apportionment [10]	Web Displays [66]	Model Predictive Control [7]	Participatory Feedback [12]	Fault Detection and Diagnosis [11]	NILM [64]	Demand Response [21]
Points	Temp Sensor	X					X		
	CO$_2$ Sensor	X							
	Occ Sensor	X	X			X			
	Lux Sensor		X			X			
	Power Meter	X	X	X		X		X	X
	Airflow Sensor			X					
Equipment	*Generic*							X	X
	HVAC	X	X	X			X		
	Lighting	X	X			X			
	Reheat Valve			X			X		
	VAV			X	X				
	AHU				X		X		
	Chilled Water			X			X		
	Hot Water			X			X		
Locations	Building				X		X		
	Floor				X	X		X	
	Room	X	X	X	X	X		X	
	HVAC Zone	X		X	X				
	Lighting Zone	X				X			
Relationships	Sensor `isLocIn` Loc.	X	X			X		X	
	Equip `isLocIn` Loc.		X			X		X	X
	Loc. `hasPart` Loc.		X		X	X			
	Loc. `hasPoint` Sensor	X	X			X		X	X
	Equip `hasPoint` Sensor	X		X			X	X	X
	Equip `hasPart` Sensor			X			X	X	X
	Equip `feeds` Zone	X			X	X			
	Equip `feeds` Room	X			X			X	
	Equip `feeds` Equip			X	X			X	
	Zone `hasPart` Room	X			X	X			

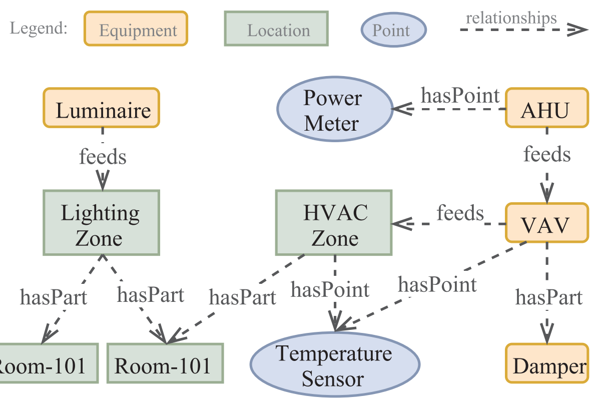


Fig. 5. Brick classes and relationships for a subset of the example building in Fig. 1.

model can be extended to incorporate more detailed control characteristics such as exact math equations. For example, Ploennigs et al. model linear time invariant dependencies for fault diagnosis [54].

4. RDF and SPARQL

4.1. Representing knowledge in RDF

Brick adheres to the RDF (Resource Description Framework) data model [55], which represents knowledge as a graph expressed as tuples of *subject-predicate-object* known as *triples*. All buildings in Brick are represented as a collection of such triples. A triple states that some *subject* entity has some relationship *predicate* to some other entity *object*, which is node/directed-edge/node in the graph theory. RDF enables easily composing different kinds of information in buildings such as hierarchical location information (e.g., room-101 is a part of the first floor) and interconnected equipment (e.g., a VAV is fed by an AHU).

All entities and relationships exist in some namespaces, indicated by a `namespace:` prefix. This enables distinguishing and reusing entities in different namespaces. Brick especially exploits well-defined standard vocabularies from RDF [56], RDFS [57] and OWL [58] to express common relationships. For example, RDFS defines subClassOf relationship to represent super-sub-concepts such as ”`sensor rdfs:subClassOf temperature sensor`”. A user can define multiple namespaces to reduce complexity in allocating unique names to entities especially when she handles many buildings. If a user defines two namespaces as `bldg1` and `bldg2`, she can easily append namespaces to `rm-101` to distinguish the rooms in two buildings with the same name as `bldg1:rm-101` and `bldg2:rm-101`.

The triples in Fig. 7 represents the connection of the VAV to the temperature sensor using the `hasPoint` relationship from the example building in Fig. 5. Line 5 declares an entity identified by the label `building:myVAV`, this creates the `myVAV` entity in the `building` namespace. `brick:VAV` is a TagSet defined by the Brick to represent any variable air-volume boxes. `rdf:type` declares `building:myVAV` to be an instance of `brick:VAV`. Similarly, line 6 instantiates a `Zone_Temperature_Sensor` with the label, `building:myTempSensor`. Line 7 uses the Brick relationship `brick:hasPoint` to declare that `building:myVAV` is functionally associated with the given temperature sensor.

4.2. Querying knowledge with SPARQL

Applications query the Brick graph for entities and relationships using SPARQL (SPARQL Protocol and RDF Query Language) [48].

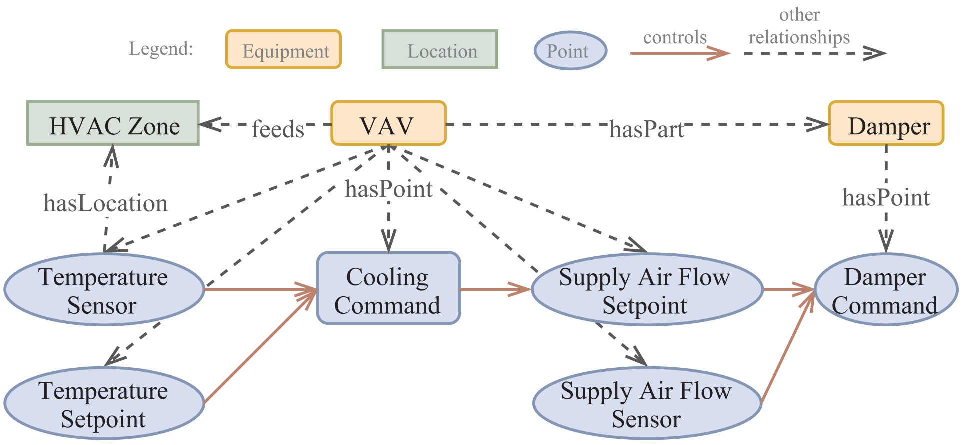


Fig. 6. Control flow example of a simplified VAV. A VAV has points related to equipment control to adjust its feeding zone's temperature. A point's value is often determined by other points' values. Such dependencies are modeled as `controls`.

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX brick: <http://brickschema.org/schema/Brick#>
PREFIX building: <http://example.com/building#>

building:myVAV rdf:type brick:VAV
building:myTempSensor rdf:type brick:Zone_Temperature_Sensor
building:myVAV brick:hasPoint building:myTempSensor
```

Fig. 7. RDF triples instantiating a VAV and a Temperature Sensor and declaring that the VAV measures temperature via that sensor.

SPARQL queries specify constraints and patterns of triples, and traverse an underlying RDF graph to return those that match. For Brick applications, this underlying graph consists of all the entities and relationships in buildings.

Fig. 8, a query for retrieving all rooms which are connected to a given AHU, contains a representative example of each of these features. Lines 1–3 declare the prefixes for the various namespaces to shorten the references to entities; for brevity, we omit these from all later queries in this paper. Line 4 contains the `SELECT` clause, which states that the variables `?ahu` and `?room` should be returned (the `?` prefix indicates a variable). The `WHERE` clause determines the types and constraints on these variables. Line 6 states that `?zone` is any entity in the graph that is an instance of the class `brick:HVAC_Zone`. Likewise, line 7 declares `?room` to be an instance of a `brick:Room`.

Brick provides both generic (such as AHU) and specific classes of equipment (such as a RoofTop-Unit AHU). A building represented in Brick can specify the specific subclasses, or if that information is not available, instantiate a generic class. Line 8 is a common construct in Brick queries which accounts for this type of uncertainty in how Brick represents buildings. This sub-query returns all entities `?ahu` that are either an instance of a subclass of `brick:AHU` or an instance of `brick:AHU` itself. An application that does not require specific features of such subclasses may want to query for the generic class rather than exhaustively specify every possible subclass.

After declaring the types of the entities involved, the query restricts the set of relationships between the entities on lines 9 and 10 to determine which pairs of entities are connected. Line 9 finds all HVAC zones downstream of a particular AHU by following a chain of `brick:feeds` relationships (the + indicates that 1 or more edges can be traversed as long as the edges are of type `brick:feeds`). Line 10 links the identified HVAC zones with the rooms they contain. The correct relationships to use can be determined from the Brick relationship list (Table 1).

This example query illustrates an important quality of Brick queries: establishing a link between two entities (even across different subsystems such as HVAC and spatial) does not require explicit knowledge of all intermediary entities. Rather, the query denotes the relevant entities and relationships: the query in Fig. 8 is indifferent to whatever building-specific equipment and details lie between an Air Handler Unit and the end zones. This is possible because the relationships between those entities all use Brick's `brick:feeds` relationship. Furthermore, the query is concise enough to return the answer only with a few expressions.

5. Brick development process

Brick development was a collaborative effort from sixteen

```
1  PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3  PREFIX brick: <http://brickschema.org/schema/Brick#>
4  SELECT ?ahu ?room
5  WHERE {
6      ?zone rdf:type brick:HVAC_Zone .
7      ?room rdf:type brick:Room .
8      ?ahu rdf:type/rdfs:subClassOf* brick:AHU .
9      ?ahu brick:feeds+ ?zone .
10     ?zone brick:hasPart ?room .
11 }
```

Fig. 8. A simple SPARQL query for retrieving all rooms connected to a given Air Handling Unit (AHU).

Table 3
Number of matching triples in each building for the SPARQL queries consisting the eight applications. A non-zero number indicates that the application successfully ran on the building. Buildings with '–' did not have any relevant points exposed in the BMS.

	Building					
Application	EBU3B	GTH	GHC	IBM	Rice	Soda
Occupancy [65]	261	245	366	821	265	232
Energy Apportionment [10]	–	302	–	397	4	–
Web Displays [66]	699	81	65	835	106	605
MPC [7]	482	69	428	324	110	482
Participatory Feedback [12]	–	253	–	386	–	–
FDD [11]	229	29	229	728	–	136
NILM [64]	6	82	–	1348	–	–
Demand Response [21]	2300	24	2490	608	4	152

researchers across seven institutions across the U.S. and Europe. From our experience on working with building systems, we identified that a common expressive schema was an essential step towards deployment of energy efficiency applications in buildings on a large scale. Together we contributed BMS data from six buildings to bootstrap the schema development. Initially, we compiled the vocabulary of terms from information in two buildings. We gradually expanded the vocabulary by adding information from two more buildings. Throughout this process, our focus was to capture information for building operations, and we excluded detailed information covered in building models for design, construction (e.g. IFC [16]) and energy modeling (e.g. gbXML [17], EnergyPlus [59]).

We concluded that a semantic model would best capture the complexity of building vocabulary we gathered as exemplified by models in other fields such as social networks [60], web search [61] and biology [62]. Hence, we organized the terms into a class hierarchy and identified relationships that would be essential but sufficient to capture the dependencies that existed between building equipment, locations and data points. We decided to use tags to support keyword search and ease of compatibility with tagging models such as Haystack [18]. We chose eight canonical energy efficiency applications from the list of applications compiled by Bhattacharya et al. [22] in their literature review. We identified the information that each of these applications will require from a building model. Given the class hierarchy and relationships in Brick, we formulated the application requirements into SPARQL queries. The SPARQL queries clearly laid out the expected relationships between different equipment and points, and acted as the specification for the semantic graph modeling of our testbed buildings to the Brick schema.

As our initial vocabulary was based on BMS information from four buildings, there was a risk that the vocabulary was not general enough to capture information found in other buildings which differ in usage or BMS vendor. To evaluate the effect of such "over-fitting" of Brick's tagsets to the set of known BMS points, we examined the percentage of BMS points covered by Brick's tagsets for the last two buildings - Rice Hall and Soda Hall - both before and after we incorporated their specialized points into Brick. Using an unaltered Brick, we matched 93.5% and 93.1% of Rice and Soda Hall's BMS points respectively, giving us confidence that Brick vocabulary does indeed capture the diversity of data points available across many buildings. After incorporating the BMS-specific points, they scored 98.5% and 98.7% respectively, using Brick's class hierarchy to avoid compromising generalizability. Table 4 contains a summary of the configuration of the six buildings, and how well Brick covers their BMS points. Examining Table 4, we can see that Brick matches the majority of points in all six buildings.

6. Applications

Applications interact with buildings through either reading or writing to the necessary data points' either historical or the most current data. However, as the timeseries data are in different structures compared to the metadata, the interactions are often separated into the following two steps. First, an application finds the names or the identifiers of the data points of interest with their metadata. Then, it retrieves or changes the data points' timeseries data in a BMS or a data historian. The application will run a fault detection algorithm or change a temperature setpoint with the retrieved data. We show how Brick and SPARQL together standardize the first step, of which typical systems lack. Brick excludes modeling the second interaction with BMS for timeseries data retrieval because each system has a unique interface. The two-steps interaction still could be further standardized through federating metadata query and data query [63]. The federated query is out of scope in this paper, but could be implemented upon Brick.

We consider eight applications — one from each of the application categories compiled by Bhattacharya et al. [22]. Research has shown that each of these applications can have a significant impact on improving building energy efficiency [64,65,10,66,7,12,11,21]. There have been hundreds of papers published that discuss how to design each of these applications so as to maximize their energy savings and we have seen several industry startups that have started to deploy them in real buildings [67–71]. If Brick successfully models different buildings in a uniform manner and enables portability of these applications, it can have a large impact on the building energy efficiency efforts.

6.1. Application coverage

We implemented these applications as a set of SPARQL queries identifying the relationships in Table 2. Brick allows applications to write *portable queries* that identify relevant resources in a building-agnostic manner. An application can then adapt its behavior to the set of returned resources, likely using some API to interact with the required points. For this reason, we implement each of the applications as a set of SPARQL queries that return the set of relevant entities and relationships. Table 3 contains the results of running these queries over the six buildings for each of the applications. Applications such as Occupancy, Web Display, Model-Predictive Control (MPC), and Demand Response run on most buildings as they are mostly related to HVAC systems, which are common in buildings. Such applications require VAVs, AHUs, HVAC zones, relevant sensors, and their relationships to each other. The Participatory Feedback application is designed for lighting controls. It shows relatively low coverage of buildings as many of the BMSes in our buildings do not expose points related to lighting systems. However, the

Table 4
Case study buildings information.

Building Name	Location	Year	Size (ft^2)	# Points	% Tagsets Mapped	# Relationships Mapped
Gates Hillman Center (GHC)	Carnegie Mellon Univ., Pittsburgh, PA	2009	217,000	8292	99%	35,693
Rice Hall	Univ. of Virginia, Charlottesville, VA	2011	100,000	1300	98.5%	2158
Engineering Building Unit 3B (EBU3B)	UC San Diego, San Diego, CA	2004	150,000	4594	96%	8383
Green Tech House (GTH)	Vejle, Denmark	2014	38,000	956	98.8%	19,086
IBM Research Living Lab	Dublin, Ireland	2011	15,000	2154	99%	14,074
Soda Hall	UC Berkeley, Berkeley, CA	1994	110,565	1586	98.7%	1939

```
1  SELECT ?airflow_sensor ?room ?vav
2  WHERE {
3    ?airflow_sensor rdf:type/rdfs:subClassOf*
4      brick:Supply_Air_Flow_Sensor .
5    ?vav rdf:type brick:VAV .
6    ?room rdf:type brick:Room .
7    ?zone rdf:type brick:HVAC_Zone .
8    ?vav brick:feeds+ ?zone .
9    ?room brick:isPartOf ?zone .
10   ?airflow_sensor brick:isPointOf ?vav .
11 }
```

Fig. 9. Genie query for airflow sensors and rooms for VAVs. The query returns all relevant triples for Genie to bootstrap itself to a new building.

relationships used in the application is generic for other types of systems too. The NILM application needs power meters to dissect energy usage into multiple subsystems, and power meters may not be integrated into the BMS as in the half of our testbed buildings.

We instantiate models from the target buildings' BMSes, so the coverages depend on how many data points the BMSes expose is the primary limiting factor for whether each application runs on a building. In addition, applications have to account for the diversity of points across buildings: Brick defines synonym tagsets where possible, but there will always be a degree of disambiguation specific to applications.

The primary challenge in developing portable queries was accounting for the variance in relationships across buildings. For example, a zone temperature sensor may have either an `isPointOf` relationship with an HVAC zone or a VAV. These inconsistencies arise from differences in building construction and the representation of the points in the BMS. It is possible to account for these differences in SPARQL to construct truly portable queries with using UNION operations that allow the temperature sensor be associated with either a zone or a VAV.

6.2. Example application: Genie

We show an example application from the perspective of Brick. The Genie [66] application incorporates monitoring and modeling of HVAC zone behavior and power usage with occupant feedback to provide a platform for occupants to directly contribute to the efficacy and efficiency of a building's HVAC system. Genie requires the following relationships:

- the mapping of VAVs to HVAC zones and rooms
- the heating and cooling state of all VAVs in the building
- the mapping of VAV airflow sensors to rooms
- all available power meters for heating or cooling equipment

Immediately, the requirements of this application outstrip the features provided by other metadata solutions. Genie needs to relate entities across subsystems typically isolated or ignored in modern BMS: the spatial construction of the building, the functional construction of the HVAC system, and the positioning of power meters in that infrastructure. Brick simplifies this cross-domain integration and makes it possible to retrieve all relevant information in a few simple queries.

To identify the airflow sensors and rooms served for each VAV, the application uses the query in Fig. 9. Lines 3–4, 5, 6, 7 find all the `Supply_Air_Flow_Sensor`s, `VAV`s, `Room`s and `HVAC_Zone`s in the building respectively. Line 8 identifies the `VAV`s that feed the respective `HVAC_Zone`s and line 9 identifies the `Room`s that are part of the corresponding `HVAC_Zone`s. Line 10 finds the `Supply_Air_Flow_Sensor`s that are part of the corresponding `VAV`s. The application uses Brick's synonyms to capture both `Discharge_Air_Flow_Sensor`s as well as `Supply_Air_Flow_Sensor`s. The "Web Displays" row of Table 3 contains the results of running Genie over the six buildings.

7. Case studies

We showcase the effectiveness of our schema by converting six buildings with a wide range of BMS, metadata formats, and building infrastructure into Brick. We discuss the challenges faced in converting the buildings into Brick as well as to provide guidance for using Brick. We also discuss how we can map labels of BMS points to Brick in Section 8.1 at scale.

7.1. Gates Hillman Center at CMU

The Gates and Hillman Center (GHC) at Carnegie Mellon University is a relatively new building, completed in 2009, with 217,000 square feet of floor space, 9 floors, and 350+ rooms of various types (offices, conference rooms, labs), and contains over 8000 BMS data points for HVAC. CMU contracts with Automated Logic[3] for building management.

The GHC includes 11 AHUs of different sizes serving multiple zones: three small AHUs serve a giant auditorium, a big laboratory and three individual rooms respectively. Eight large AHUs supply air to more than 300 VAVs. GHC's HVAC system also contains computer room air conditioning (CRAC) systems which are equipped with additional cooling capacity to maintain the low temperature in a computer room and fan coil units systems to provide cooling and ventilation functions. Brick matched 99% of GHC's BMS points, with the remaining points being too uncommon to be required by most applications (such as a `Return Air Grains Sensor` which measures the mass of water in air).

The major challenge in GHC was determining the relationships between pieces of equipment not encoded in the BMS labels. While the information is available through an Automated Logic GUI representation of the building, there was no machine readable encoding of which VAVs connected to which AHUs. This required examining the building plans directly to incorporate more than 400 relationships Brick representation, instead of being reliant upon manually examining a GUI to determine relationships between equipment, is more amenable for applications in both human and machine readable formats.

7.2. Rice Hall at UVA

Rice Hall hosts the Computer Science Department at the University of Virginia. The building consists of more than 120 rooms including faculty offices, teaching and research labs, study areas and conference rooms distributed over 6 floors with more than 100,000 square feet of floor space. The building contracts with Trane[4] for building management.

Rice Hall contains four AHUs associated with more than 30 Fan Coil Units (FCU) and 120 VAVs serving the entire building. Besides the conventional HVAC components, the building features several different

[3] Automated Logic, http://www.automatedlogic.com/.
[4] Trane, https://www.trane.com/.

new air cooling units, including low temperature chilled beams and ice tank-based chilling towers, an enthalpy wheel heat recovery system, and a thermal storage system. The building also contains a smart lighting system including motorized shades, abundant daylight sensors and motion sensors. Rice Hall's BMS points are easily interpretable for conversion to Brick despite it containing some uncommon equipment such as a heat recovery and thermal storage systems as part of the building design as an energy-efficient "living laboratory". However, the relationships defined by Brick sufficiently captured their relationships to the other parts of the system. They also have points specific to Rice Hall such as `ice tank entering water temperature sensor`. Brick's structure allows the clean integration of such new tagsets into the hierarchy without disrupting the representation of existing buildings.

7.3. Engineering Building Unit 3B at UCSD

The Engineering Building Unit 3B (EBU3B) at University of California, San Diego hosts the Department of Computer Science & Engineering and contains offices, conference rooms, research laboratories, an auditorium and a computer room. The building was constructed in 2004 and has 150,000 square feet of floor space with over 450 rooms. The BMS of EBU3B is provided by Johnson Controls,[5] and contains more than 4500 data points, most of which related to the HVAC system and power metering infrastructure.

The HVAC system consists of a single AHU that supplies conditioned air to 200+ VAV units and some FCUs. There are exhaust fans for all kitchens and restrooms and a CRAC system serving the computer room. The HVAC system also has Variable Frequency Drives (VFD), valves, heat exchangers and cooling coils to facilitate operation of AHU and CRAC. Brick's schema provides the necessary tagsets and relationships for all of these components. The university central power plant provides the hot and cold water for domestic medium temperature water system and controlling air temperature in the HVAC. The corresponding sensors that measure the hot and cold water use such as flow rate and temperature were modeled in Brick, but the central plant was left out as it was not part of the building.

An issue in mapping EBU3B to Brick is that the AHU discharge air is divided into two parts for two wings of the building. Brick currently does not model how the discharge air in the AHU is divided into two wings but describe the connections to other equipment such as VAVs. Additionally, EBU3B's BMS contains data points related to Demand Response (DR) events such as load shedding for hot water, which exposes an interesting conflation of the representation and operation of the building, while Brick does not model DR events as points. Because BMSes have been typically written as monolithic applications over vendor-specific interfaces, they must incorporate external signals such as DR into the set of BMS points directly. On the other hand, Brick decouples the resources and infrastructure of a building from the building operation so that any application can operate on top of Brick representation.

7.4. Soda Hall at UC Berkeley

Soda Hall, constructed in 1994, houses the Computer Science Department at UC Berkeley. It mostly consists of closed small to medium sized offices, where either faculty or groups of graduate students sit. The BMS system, provided by the now-defunct Barrington Systems, exposes only the data points in the HVAC system.

The HVAC system of the building runs on pneumatic controls, and comprises 232 thermal zones. Each zone has a VAV and especially VAVs for the zones on the periphery of the building have reheat mechanism. For a VAV with reheat, the same control setpoint indicates both the amount of reheat and the amount of air flowing into a zone. While such combination is building-specific, Brick can express the fact that the same sensor controls both the reheat and air flow by labeling the point as a subclass of both `reheat command` and `air flow setpoint` tagsets. The logic of the setpoint also can be described with control relationships in Brick for dependencies to other setpoints related to actual reheat and air flow rate.

Unique to the other buildings presented here, the operational set of Soda Hall's HVAC components is not static. Soda Hall contains a redundant configuration of chillers, condensers and cooling towers. At any point of time, one of these systems is operational while the others are kept as standby. An isolation valve setpoint indicates which of the redundant subsystems is currently operating. Brick completely expressed the redundant subsystem arrangement, but the equipment contained several unique points such as `on timer` for the chiller subsystem that had to be added to Brick's tagsets.

7.5. Green Tech House

The Green Tech House (GTH) was constructed in 2014 as a 38,000 square feet office building in Vejle, Denmark. It contains 50 rooms spanning three stories and functions as office spaces, a cafeteria, meeting rooms and bathrooms. GTH is controlled by the Niagara BMS,[6] but to protect basic building functionality only a subset of the BMS points are exposed via oBIX. As the oBIX points do not include AHU nor VAV points, the Brick representation was constructed from a combination of BMS points, BMS screen shots and technical documents.

Compared to the rest of the case study buildings, the thermal conditioning of GTH is reversed: Air is heated centrally in a single AHU and distributed to VAVs with cooling capabilities. The AHU uses a rotary heat exchanger to recovers heat from the return air. The pressure of the AHU return and supply air for the north and south side of the building is measured separately. Additionally, most rooms have radial heating on either walls or in the floor. These are supplied by two independent hot water loops – one for wall-mounted heaters and one for floor heaters – heated by district heating.

The two main challenges were to (i) find, extract and merge information from diverse sources, and (ii) to map this to Brick. Although equivalents are present neither the BMS nor the technical documentation of GTH refers to AHUs and VAVs. These equivalents are not named.

7.6. IBM Research Living Lab

The IBM Research building in Dublin was retrofitted as modern 15,000 m^2 office in 2011 from an old factory. The building serves as living laboratory for IBM's Cognitive Building research and is heavily equipped with modern building automation technology to provide a rich data source for research.

The building has been renovated multiple times and new systems were installed by different companies. The heterogeneity of systems became very high in the building. The building contains 2154 points collected from 11 different systems. The building is served by 4 AHUs with 115 points but also has old disconnected legacy systems in the point list. Unlike the other buildings, it contains 250 smart meters and 150 desk temperature sensors. It has 1000 points for 161 FCUs as well as 350 points on the lighting system including 150 PIR sensors and door with people counters.

The configuration of the FCUs connected to different AHU, boilers and chillers are unique for this building while terminal units such as VAVs and FCUs are connected to a single central unit such as an AHU in the other buildings. It shows importance of the relationship modeling and the capability of Brick.

[5] Johnson Controls, http://www.johnsoncontrols.com/.

[6] Tridium, https://www.tridium.com/.

8. Converting BMS points to Brick

Existing building metadata and BMS points need to be converted to the Brick schema for use by applications. The ease with which this conversion can be performed will have a significant impact on its adoption. Some buildings describe the metadata using vendor specific nomenclature while others use schemata such as Haystack and IFC. Though existing schemata do not capture all of the entities and relationships that Brick can express, it is possible to automate the conversion of a subset of each schema. We describe a general approach for converting building metadata to Brick, present initial conversion techniques for the popular Haystack and IFC building schemata, and demonstrate these techniques on three real buildings.[7] Finally, we describe methods to convert vendor specific BMS metadata to Brick.

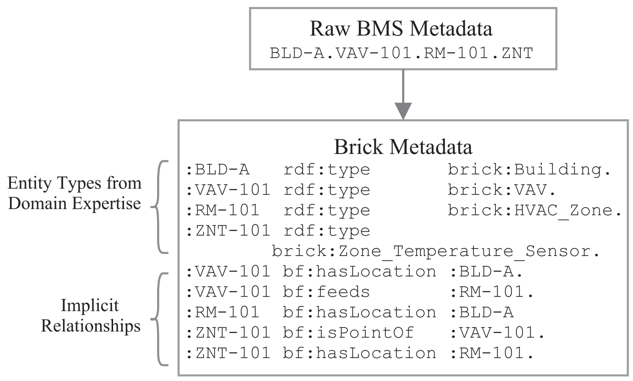
The general approach is to parse the given building metadata into sets of entities that have obvious relationships between them and then add these entities and relationships to a Brick model. The success of a conversion depends on what information is captured in a schema and how structured that information is. For unstructured metadata, the conversion implementation is often site-specific. For structured metadata, the conversion implementation is more portable.

8.1. BMS point conversion

Metadata in traditional BMSes is commonly represented as unstructured or semi-structured strings identifying points of measurement or actuation. Converting BMS metadata to Brick requires extracting the semantic information, i.e. entities and relationships, from these labels. Because point names are inconsistently named across and within buildings, this conversion requires tremendous manual effort, domain expertise and in-depth knowledge of the target building.

Fig. 10 shows an example of a BMS point and the equivalent Brick representation. To convert the BMS point to Brick, we need to infer that `ZNT` corresponds to `Zone_Temperature_Sensor`, `RM-101` corresponds to the room (and HVAC zone) where the sensor is located in a building named `BLD-A` and `VAV-101` is a VAV feeding conditioned air to this HVAC zone. Domain experts need to provide this mapping to automate this conversion of point name to entities. Once the entities are extracted, we can infer the relationships between them using the fact that there are only a limited number of possible relationships between types of entities. For example, a zone temperature sensor and a VAV probably have a `isPointOf` relationship between them. However, BMS point names do not always contain all the information necessary to fully populate a Brick model, such as which VAVs are downstream of a particular AHU. This information could be obtained through interviews with building managers. For the six case study buildings described in Section 7, we generate Brick instances using Python scripts which parse point names to generate Brick entities and infer the relationships between them.[8]

Several frameworks have been proposed to reduce the effort of converting BMS metadata to standardized vocabularies. Because these vocabularies are well-structured, they are simple to convert to Brick. Most of the frameworks focus on identifying point types. Methodologies include clustering BMS metadata with similarities to reduce the number of inputs to learn a model inferring point types [72,73]. Gao et al. extract features from time series data to learn a model for point types [74]. Pritoni et al. propose to learn the relationships between AHUs and VAVs by observing reactions of devices to artificial perturbations [75], which correspond to `feeds` relationship in Brick. Bhattacharya et al. propose a framework to construct synthesis rules from examples presented to building managers and domain experts [76]. The synthesis rules extract all possible relationships in BMS metadata which usually covers Equipment, Point, Location and relationships among them like `hasLocation` and `isPartOf`.



```
:BLD-A   rdf:type       brick:Building.
:VAV-101 rdf:type       brick:VAV.
:RM-101  rdf:type       brick:HVAC_Zone.
:ZNT-101 rdf:type
        brick:Zone_Temperature_Sensor.
:VAV-101 bf:hasLocation :BLD-A.
:VAV-101 bf:feeds       :RM-101.
:RM-101  bf:hasLocation :BLD-A
:ZNT-101 bf:isPointOf   :VAV-101.
:ZNT-101 bf:hasLocation :RM-101.
```

Fig. 10. An example of mapping raw metadata of a BMS point to Brick. The abbreviations inside the raw metadata represent some entities in the BMS and the mapping can be given by a domain expert or inferred by an automated inference algorithm. The relationships between entities in the raw metadata are implicit but the number of possible relationships are limited as shown above.

While this family of techniques is effective for creating part of a Brick model, the synthesis of a complete Brick model (including control relationships, spatial information and the full specification of building subsystems) has not yet been fully automated. The integration of many sources of building metadata for the creation of a complete Brick model is the subject of future work.

8.2. Haystack model conversion

We have developed a simple Haystack to Brick converter. It has two components: a translator module maps Haystack entities to Brick tagsets, and a relationship module infers a possible set of Brick relationships between those entities using contextual information and a set of base assumptions.

Haystack entities, like Brick entities, refer to equipment, sensors, setpoints and other physical objects and are described using a combination of Haystack tags and tag-value references to other entities. To convert these entities to Brick, the translator module takes advantage of the fact that Brick tagset names are based on Haystack tags. For each Haystack entity, the translator finds the Brick tagset with the largest intersection with the entity's Haystack tags and adds a corresponding Brick entity to the output Brick model. For example, a Haystack entity with the tags `air`, `exhaust`, `flow`, `his`, `lab`, `sensor` and identifier "OEA-F-3133-Lab" shares the most tags with the Brick tagset/class `Brick.Exhaust_Air_Flow_Sensor`, so the translator would output the following triple:

```
1 Bldg.OEA-F-3133-Lab rdf:type
  Brick.Exhaust_Air_Flow_Sensor
```

This technique captures the majority of Haystack entities, and requires only minor additions to account for rare or site-specific equipment. These additions can be carried forward for future translations.

The relationship module uses the tag-value pairs each Haystack entity contains to populate the set of relationships around the translated Brick entity. Some of these tag-value pairs describe aspects of the entity such as engineering units or square footage, which could be captured in future Brick extensions. Other tag-value pairs use "-Ref" tags to relate entities (e.g. `equipRef`, `siteRef`, `elecMeterRef`). These references do not capture the full set of relationships required by Brick, but usually imply a few obvious relationships. For example, the `elecMeterRef` implies a Brick `hasPoint` relationship between an equipment entity and an electric meter entity. The ubiquitous `equipRef` tag requires more context. If the owner (the entity that has the tag) and the target (the value of the `equipRef` tag) are both equipment, then we infer a

[7] These are different from the six buildings we present in our case studies (Section 7).
[8] https://github.com/BuildSysUniformMetadata/Brick.

```
area: "546 sq ft"
associatedRooms: "3165, 3240"
canopyHoodSignage: true
code: "VAV-1D"
dcv: 1.0
done: true
equip: true
equipParent: "AHU 04"
equipRef: "AHU 04"
id: "VAV 4_16 Rm 3167"
navName: "VAV 4_16 Rm 3167"
priorityTwo: true
siteRef: "Ghausi"
vav: true
```

Fig. 11. Haystack VAV entity, with spatial information encoded in the entity identifier string. Note that only boldface tags are standardized in Project Haystack. The original author of this metadata needed to add the other tags.

```
Ghausi.AHU_04             rdf:type        brick:Air_Handling_Unit .
Ghausi.VAV_4_16           rdf:type        brick:VAV .
Ghausi.Room3167           rdf:type        brick:Room .
Ghausi.Room3165           rdf:type        brick:Room .
Ghausi.Room3240           rdf:type        brick:Room .
Ghausi.HVAC_Zone_4_16     rdf:type        brick:HVAC_Zone .
Ghausi.HVAC_Zone_4_16     bf:hasPart      Ghausi.Room3167 .
Ghausi.HVAC_Zone_4_16     bf:hasPart      Ghausi.Room3165 .
Ghausi.HVAC_Zone_4_16     bf:hasPart      Ghausi.Room3240 .
Ghausi.AHU_04             bf:feeds        Ghausi.VAV_4_16 .
Ghausi.VAV_4_16           bf:feeds        Ghausi.HVAC_Zone_4_16 .
```

Fig. 12. The Brick triples (entities and relationships) generated from the Haystack entity in Fig. 11. Boldface relationships and TagSets are standardized by Brick and a user only needs to define the identifiers. The annotative information such as duplicated names and code is omitted from Fig. 11.

Brick `feeds` relationship between the entities. If the owner is a sensor and the target is equipment, then we infer a Brick `isPointOf` relationship. With these simple contextual assumptions, the Haystack identifiers of the owner and target of a "-Ref" tag are enough to generate the requisite Brick triples. Furthermore, We use the same technique as BMS point conversion for the relationships that are implicit in Haystack names and identifiers, which requires a more site-specific implementation.

Fig. 11 contains the Haystack representation of a VAV entity in Ghausi Hall on the UC Davis campus (described below). The non-standard `associatedRooms` tag and the room number in the entity identifier string ("VAV 4_16 Rm 3167") describe the set of rooms in the HVAC zone conditioned by the VAV. From this entity, we can instantiate three rooms, an air handling unit, a VAV, an HVAC zone, and the set of Brick relationships connecting all of them. The resulting triples are in Fig. 12.

We have implemented our Haystack-to-Brick converter script in Python, totaling 350 lines of code.[9] We apply this technique to two Haystack models from the UC Davis campus and were able to successfully translate air handling units, VAVs, dampers, HVAC zones, rooms, setpoints and electric meters as well as temperature, humidity and occupancy sensors. Ghausi Hall is a 66,000 sq ft engineering building with 2183 Haystack entities; the translated Brick model contains 4135 triples. PES is a 90,000 sq ft office and lab building with 6475 Haystack entities; the translated Brick model contains 15,561 triples.

8.3. Converting IFC

The IFC building information model captures a very different set of relationships than Brick. However, it is still possible to generate a partial Brick model from an IFC representation of a building. IFC models mostly consist of spatial information useful for construction such as the size and position of walls, dampers and ducts, but also includes semantic groupings of these entities into floors, rooms and HVAC zones. The IFC schema encodes information as "objects", which correspond to equipment, spaces and other infrastructure. Objects can also refer to groups of objects.

We have implemented a simple converter that exports spatial information in IFC models to Brick. The converter first scans an IFC model for all instances of `IFCZONE` objects, which can correspond to an HVAC Zone, and `IFCSPACE` objects, which correspond to rooms. `IFCRELASSIGNSTOGROUP` objects associate zones (using a "RelatingGroup" attribute) with a list of rooms (using a "RelatedObjects" attribute). `IFCRELAGGREGATES` objects associate rooms with floors (instances of `IFCBUILDINGSTOREY`).

Our IFC-to-Brick converter, implemented in 100 lines[10] of Python (not including an open-source IFC file parser[11]), converts IFC representations of floor, room and zones to their Brick equivalents. The converter currently makes the assumption that all zones are HVAC zones because there is not enough contextual information in the IFC model to determine the "kind" of zone without programmatically traversing the components of the HVAC system as represented in the IFC model. We have successfully tested the converter on an IFC model of a 7000 sq ft office building in downtown Berkeley. The textual IFC model totals some 150,000 lines and the exported Brick model contains 159 triples. This informally illustrates the expressive differences between IFC and Brick; the IFC model contains a very detailed description of the construction physical space, but the translated Brick model only

[9] https://github.com/gtfierro/BrickConvert/tree/master/haystack.

[10] https://github.com/gtfierro/BrickConvert/tree/master/ifc.

[11] https://github.com/mvaerle/python-ifc.

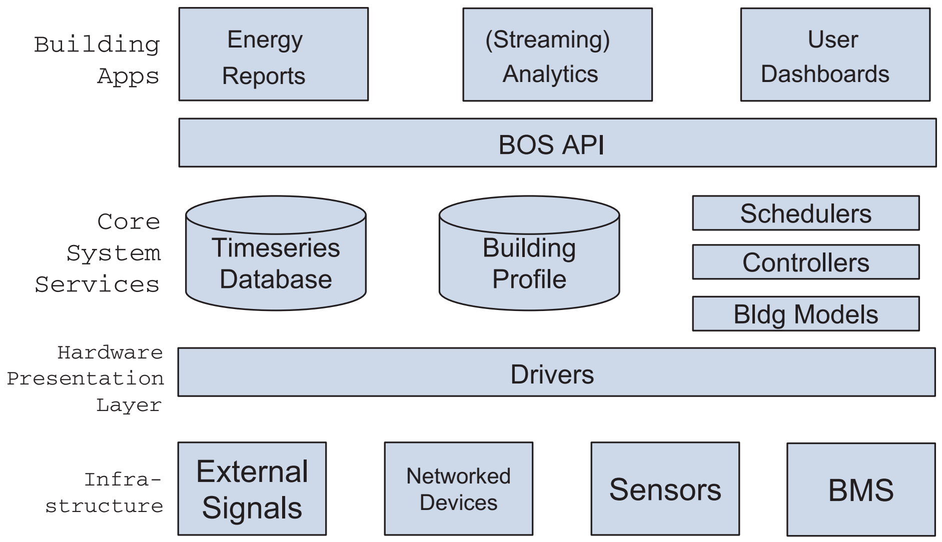


Fig. 13. Generic Building Operating System (BOS) Architecture, with the Brick building model stored in the Building Profile's metadata database.

represents the high-level spatial information required by building applications.

9. Building operating system integration

We have designed Brick so that it describes the essential components of buildings present in existing BMSes and supports applications on top of the building system infrastructure. In this section, we outline the role of Brick in building operating systems.

9.1. Building Operating System (BOS)

Traditional BMSes provide supervisory operation and maintenance of different building systems. However, to support third party applications, we need to have other functionalities such as management of metadata, data storage, search, authentication and access control. We use the term *Building Operating System* (BOS) to describe such a system. Fig. 13 shows the architecture for a generic BOS. We can think of a traditional BMS as a single "monolithic" application engineered for a specific building. It is possible to extend or augment a BMS's functionality, but this often requires intensive collaboration between a building manager and a BMS engineer. In contrast, BOS seek to be much more easily extensible. An BOS provides an application programming interface (API) that enables users such as building managers, controls engineers and even building occupants to integrate building components and data into novel applications. An effective metadata solution for buildings is crucial because it allows applications and users to easily find the necessary components for building controllers, schedulers, analytics and other software.

A BOS abstracts the different types of systems, equipment and sensors in a building with a hardware presentation layer that provides a common interface for interactions and abstracts away different communication protocols such as BACnet, LonTalk or ZigBee. The infrastructure components are represented in a canonical way using Brick and stored in a database. A BOS provides storage infrastructure and control system framework for applications to perform data analytics and control operations. All the interactions with a BOS are authenticated and access control permissions ensure that applications privileges are kept in check while building managers have supervisory access. For example, building managers would be allowed to update equipment model and create new sensor data streams, while a fault detection application can only read the data stream. Applications interact with a BOS using APIs and building managers will have graphical interfaces for overseeing building operations. A few examples of BOS include Tridium's NiagaraAX [77] and academic efforts such as XBOS and BuildingDepot.

9.2. Role of brick in a BOS

The building Brick model describes all the infrastructure components using the Brick schema. Hence, it acts as the common metadata layer that applications and users use to interact with the building infrastructure. This Brick building model is stored in a metadata database and a user can search the database for specific sensors, setpoints, equipment, etc using SPARQL. Any changes to building infrastructure can be updated by the building manager using the metadata database. Each Brick entity is associated with a corresponding API endpoint to retrieve historical data or send control commands. With this architecture, users can discover resources in a building, identify relationships between building entities, and access spatial, mechanical and control system context.

9.3. Case study: XBOS

The eXtensible Building Operating System (XBOS) is a distributed BOS composed of microservices[12] communicating over a secure message bus. The BOSSWAVE [78] message bus provides topic-based publish-subscribe functionality coupled with a fine-grained permission model. The hardware presentation layer in XBOS communicates with BMS points, equipment, devices, data sources and other external, networked resources using drivers. Drivers expose functionality through standardized interfaces accessed through publishing and subscribing on structured topic names. An archival service stores all produced data in a timeseries database.

The Building Profile stores Brick models for the buildings in an XBOS deployment and serves SPARQL queries against them. The Building Profile provides the necessary binding between physical entities (building subsystems, sensors, setpoints and equipment) and logical entities (drivers, services, controllers and streams of timeseries data). It also contains references to the timeseries data for Brick entities such as sensors, setpoints, commands and other networked devices. XBOS applications query the Brick model to find the equipment they need to operate as well as for the necessary identifiers to either interface with that device (by using API endpoints to communicate with

[12] Martin Fowler, Microservices, https://martinfowler.com/articles/microservices.html.

XBOS drivers) or query historical state (by using timeseries identifiers to communicate with the archiver). In XBOS, the Building Profile is implemented using HodDB, a specialized RDF/SPARQL database for Brick [79]. HodDB resolves SPARQL queries at interactive speeds (< 100 ms) and integrates with the BOSSWAVE message bus. On top of these base services, applications implement controllers, schedulers, analytics and dashboards for the buildings in the deployment. Low-latency Brick queries allow all components of XBOS to make use of building metadata without significantly impacting performance; user interfaces, controllers and alarms especially need fast queries.

Brick is an essential component of XBOS because it enables portable applications that can discover not only physical building resources through SPARQL queries, but also the related logical entities that provide BOS functionality.

9.4. Case study: BuildingDepot

BuildingDepot (BD) is a BOS designed for scalable and secure big data management across multiple buildings with a protocol-agnostic API for data storing and actuation [25]. It consists of two types of services: (i) a Central Service (CS) to manage metadata and (ii) a Data Service (DS) to manage timeseries data. The metadata of BD was originally designed with tag-value pairs stored in MongoDB, a document-oriented database. For example, a temperature sensor point can be associated with tag-value pairs such as `unit: Fahrenheit` and `location: Room-101`. Each point in CS can have an associated timeseries data stream stored in DS and they are linked with a unique identifier. An application can find points matching requested tag-value pairs from CS and then request corresponding data with the found identifiers to a DS. Users are also associated with tag-value pairs for access control. When a user tries to read data of some points from DS, her access permission is evaluated by what tag-value pairs the user is associated with. If a user is associated with `location: Room-101`, she can access data points in that room.

We implement Brick's functionality on top of tag-based scheme that BD was originally designed for. The data models and their examples are described in Fig. 14a. An entity's metadata are composed of its name and the tag-value pairs describing it. While tag-value pairs can be arbitrary, we adopt Brick TagSets as tags and entity names as values to emulate Brick's graph structure. `hasLocation` tag in Fig. 14b emulates a triple of `ZNT-1 brick:hasLocation RM-101` where `RM-101` is a Room. We introduce BrickTagSets to represent TagSets and they can have Brick relationships with each other using the tag-value pairs. While any relationship can be added to BrickTagSets, `subClassOf` and `superClassOf` relations are fully expanded for query optimization as it is common to exploit transitive queries such as finding an instance of all subclasses of a temperature sensor. We flatten such hierarchy and add all subclasses of an entity to the `superClassOf` field.

In addition, we implement a subset of SPARQL on top of tag-value querying capability supported by document-oriented databases. Each triple in WHERE clause of SPARQL filters candidates satisfying the triple which can be emulated by finding documents matching with given tag-value patterns. Recursive querying in a triple with * operator is executed with document match querying recursively. Note that we exploit MongoDB's document-oriented structure to minimize change to the original BD architecture. Although its speed is lower than querying a native RDF store, any tag-based system can adopt Brick with this framework.

10. Extensibility model

The Brick schema currently incorporates points, equipment and location entities that exist in modern BMS and are required for canonical applications. We envision new applications will emerge, and developers may want to model other aspects of a building such as its network infrastructure or security system. We have a well defined process to perform updates to Brick. We briefly outline the process here.

We use the Brick GitHub repository[13] as the main tool for update requests, discussions and development tracking. We will have *Brick maintainers* who manage the repository. Maintainers will also manage a development road map which would list all future milestones, i.e., anticipated changes to Brick. Community developers can take up the implementation of any of these milestones and perform a git pull request to merge the changes with the Brick schema. Maintainers will review the changes and perform the merge. Based on these merges, new versions of Brick will be released. Building applications and systems can still use the older version of the Brick and update to the newer version as per their needs. Brick version follows the Semantic Versioning v2 system[14]: backward compatible changes to Brick will be released as minor versions and non-compatible changes will be released as major versions.

Requested changes or updates are filed as issues on the GitHub issue tracker. Mature proposals can be submitted as Requests For Comment (RFCs), also using the issue tracker.[15] The issue tracker will serve as the primary forum for discussion of the submitted change, but RFCs will also be announced on the mailing list.[16] Community members are free to discuss issues and suggest modifications. Based on the discussions, the Brick maintainers may choose to accept or reject the RFC. Accepted RFCs will be added to the Brick development roadmap and included in a subsequent release.

Discussions in RFCs cover various aspects of schema development. An RFC for lighting systems filled the gap of domain expertise among the original Brick developers by providing exact hierarchy of necessary TagSets and expedited discussions to select common vocabularies across different domains (e.g., `occupancy sensor` in HVAC and `presence sensor` in lighting systems) and to find common concepts in the domains (e.g., `interface` for `thermostat` in HVAC and `LED touch panel` in lighting systems. We will discuss lighting systems and electrical power systems extended through RFCs in Section 10.1.

10.1. Ongoing extensions

Several external contributors have proposed extensions to lighting systems and electrical systems in the Brick schema. The essential concept in the lighting system is the lighting fixtures and its control system.[17] We introduce the TagSet `Luminaire` to represent lighting fixtures and `Luminaire_Driver` for the corresponding end point controllers. A `Lighting_Zone` is the area in the building that is controlled by a single `Luminaire_Driver`. We introduced the tagset `interface` to capture devices that are designed for human interaction. These are common across domains: `dimmer` in lighting systems and `thermostat` in HVAC. We reuse the `feeds` relationship to model the relationship of `luminaire` feeding light to `lighting zone`. The basic taxonomy is shown in Fig. 15a.

The proposed extensions to the electrical power system captures the concepts in the power distribution network and related equipment.[18] Each equipment is either used for isolation — `Bus, Transformer` or protection — `Circuit_Breaker`. Each of the categories has specialized equipment, e.g., `Panel_Board` is a type of `Bus`. Such hierarchical relationship is captured well with subclasses in Brick. We reuse `feeds`

[13] https://github.com/BuildSysUniformMetadata/Brick.
[14] http://semver.org/.
[15] https://github.com/BuildSysUniformMetadata/Brick/issues/25.
[16] https://groups.google.com/forum/#!forum/brickschema.
[17] Lighting System RFC: https://github.com/BuildSysUniformMetadata/Brick/issues/29.
[18] Electrical System RFC: https://github.com/BuildSysUniformMetadata/Brick/issues/28.

```
Entity = {
  "Name": String,
  "Tag-Value Pairs": List of (Tag String, Value String).
}
BrickTagSet = {
  "Name": String,
  "subClassOf": List of subclass Brick TagSets in String.
  "superClassOf": List of superclass TagSets in String.
}
```

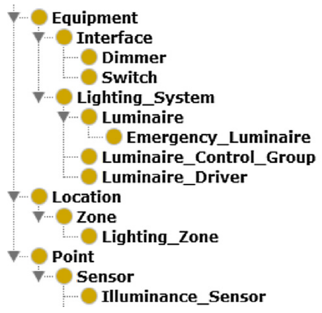
(a) Data Schema

```
ZNT-1 = {
  "Name": "ZNT",
  "Tags": [("type", "brick:Zone Temperature Sensor"),
           ("bf:hasLocation", "RM-101")]
}

Zone_Temperature_Sensor = {
  "Name": "Zone Temperature Sensor",
  "subClassOf": ["Temperature Sensor", "Sensor"],
  "superClassOf": ["Average Zone Temperature Sensor"]
}
```

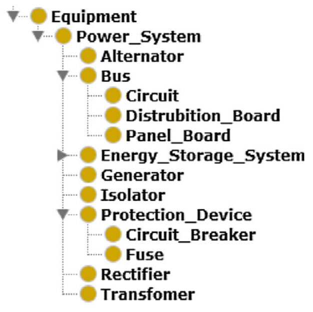
(b) Example entity and TagSet.

Fig. 14. Building Depot 3.0 Data Model.



- Equipment
  - Interface
    - Dimmer
    - Switch
  - Lighting_System
    - Luminaire
      - Emergency_Luminaire
    - Luminaire_Control_Group
    - Luminaire_Driver
- Location
  - Zone
    - Lighting_Zone
- Point
  - Sensor
    - Illuminance_Sensor

(a) Lighting system tagsets.



- Equipment
  - Power_System
    - Alternator
    - Bus
      - Circuit
      - Distrubition_Board
      - Panel_Board
    - Energy_Storage_System
    - Generator
    - Isolator
    - Protection_Device
      - Circuit_Breaker
      - Fuse
    - Rectifier
    - Transfomer

(b) Power system tagsets.

Fig. 15. Extended domains proposed and discussed through RFCs. The extended domains share the same characters of hierarchical relationships among tagsets and common relationships such as `feeds` already defined in Brick.

to model flow of electricity from `transformer` to `circuit breaker`, `circuit panel`, `isolator`, and then to end equipment like a refrigerator. Power meters are modeled equipment that `hasPoint` such as `power`, `voltage`, `current`. With the `power meter` class and `feeds` relationship, a user can easily query what loads a power meter is measuring.

11. Integration with other ontologies

There are various aspects of buildings that applications need to exploit and a single model cannot describe everything. Even though integrating different ontologies and standards for a system is a common practice, there has been little discussion in how to systematically integrate different models in buildings. In the RDF framework, it is easy to extend Brick to accommodate other ontologies by connecting relevant concepts via either predefined or custom relationships. Each ontology community can maintain and develop their own model without deteriorating the other models. As an example, we illustrate the integration of Brick with three ontologies covering different aspects in Fig. 16, showcasing Brick's flexibility and extensibility even for the scope outside Brick's original design.

11.1. Unit of measurement (QUDT)

Units of measurement vary across systems, e.g., Celsius and Fahrenheit for temperature measurements. They need to be explicitly specified so that applications can interpret corresponding data unambiguously without human input. QUDT is a representative ontology for quantities, units, and data types [80]. We link the vocabularies under `QuantityKind` and `Unit` it the QUDT ontology using the relationships `hasQuantityKind` and `hasUnit` respectively as shown in

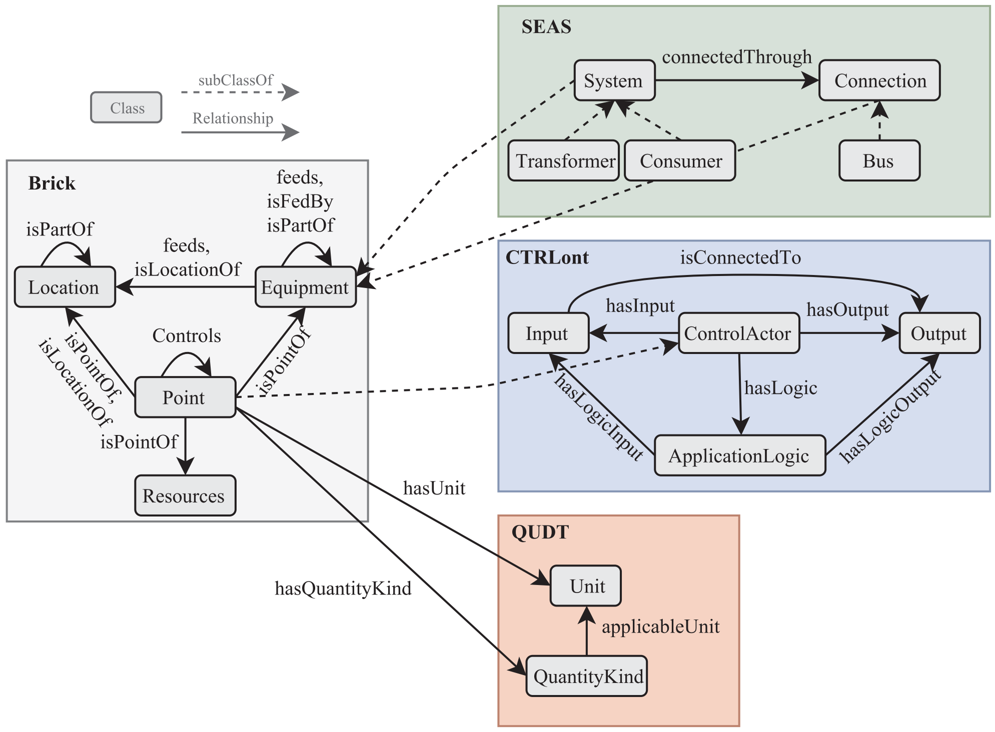


Fig. 16. Integration of Brick with other ontologies. Common concepts are linked through `subClassOf` relationships and auxiliary concepts are connected through new relationships. This integration provides all the functionalities without violating any models.

```
# query name: unit conversion
PREFIX unit: <http://qudt.org/vocab/unit/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX qudt: <http://qudt.org/schema/qudt/>
PREFIX bf: <https://brickschema.org/schema/BrickFrame#>
PREFIX building: <http://example.com/building#>
SELECT ?result
WHERE {
  VALUES (?currVal ?targetUnit) { (70 unit:DEG_C) } .
  building:ZNT-101 bf:hasUnit ?srcUnit. # Assume ?srcUnit is unit:DEG_F.
  ?srcUnit qudt:conversionMultiplier ?srcFactor. # ?srcFactor = 1.0
  ?srcUnit qudt:conversionOffset ?srcOffset. # ?srcOffset = 273.15
  ?targetUnit qudt:conversionMultiplier ?targetFactor. # ?targetFactor = 0.5556
  ?targetUnit qudt:conversionOffset ?targetOffset. # ?targetOffset = 255.372
  BIND ((((xsd:float(?currVal) * xsd:float(?srcFactor) + xsd:float(?srcOffset))
    - xsd:float(?targetOffset)) / xsd:float(?targetFactor)) AS ?result).# =21.11
```

(a)

```
# query name: unit validation
# Same namespace prefixes in the above query.
SELECT ?isApplicable
WHERE {
    VALUES ?target {building:ZNT-101} .
    ?target bf:hasQuantityKind ?qk .
    ?target bf:hasUnit ?targetUnit .
    ?qk qudt:applicableUnit ?applicableUnit .
    BIND (?applicableUnit = ?targetUnit AS ?isApplicable) .
}
```

(b)

Fig. 17. Example usages of QUDT with Brick. (a) Automated unit conversion: This query converts a temperature value from the sensor in an unknown unit into Celsius. The base unit of temperature units is Kelvin (retrieved from QUDT) and the parameters converting them into Kelvin can be automatically retrieved from QUDT and then used to produce a value in the target unit. The value in the source unit is converted into the base unit, Kelvin, and into the target unit, Celsius, in turn. This query returns the right conversion of 70°F in Celsius, 21.11. `bldg:ZNT-101`, the target value 70, and the target unit unit:`DEG_C` can be parameterized for more generic usage. In SPARQL, `VALUES` provides inline values to variables and `BIND` assign values in certain rules to a variable. (b) Automated unit validation: This finds a QuantityKind and a Unit corresponding to the sensor ZNT-101, and then checks if the unit is found in the QuantityKind's applicable unit set. `building:ZNT-101` can be parameterized.

Fig. 16. `QuantityKind` represents "any observable property that can be measured and quantified numerically [80]" such as temperature and energy. The vocabularies under `QuantityKind` can be automatically associated with Brick `Point` TagSets that contain Tags of what they measure. For example, `temperature sensor` contains `temperature` as a Tag and we can infer that any instances of `temperature sensor` should have `temperature` as a `QuantityKind`. `Unit` is "a particular quantity value that has been chosen as a scale for measuring other quantities the same kind [80]" such as Celsius and Joule. The vocabulary of units in the building domain can be extracted from BACnet vocabularies or directly adopted from QUDT in the future. QUDT defines extensive instances of both `QuantityKind` and `Unit`, and each instance of `QuantityKind` is associated with a set of units through the relationship, `applicableUnit`. Thus, we can systematically define the semantic relationships between Brick points and units through QUDT.

Given the explicit representation of units as an ontology, we can automate various use cases handling units [81]. We present two of the use cases in Fig. 17 for building applications. The first one (Fig. 17a) is to convert a value in a unit into a target unit automatically. An application does not need to know unit conversion rules for given values but just needs to submit the query with a value for a target unit; this is Celsius in the example. The second one (Fig. 17b) is to validate if the given unit for a point is correct. The validation query checks if the discovered unit is applicable to the corresponding quantity kind. Thus, QUDT integration enables automated functionalities with unit composition, conversion and validation.

11.2. Control Logic (CTRLont)

Even though Brick's `controls` relationships can represent control dependencies between Points, some applications may require full control logic such as PID controllers and state machines. CTRLont [82] is an ontology modeling control logic that can fully describe control actors and logic, and modularize the logic to ensure reusability and easy extension. We can easily integrate the CTRLont into Brick. The core concept of CTRLont is the "sense-process-actuate" model of control processes: `ControlActor` processes `Inputs` based on `ApplicationLogic` and produces `Outputs` that may actuate devices. Points in Brick receive inputs based on `controls` relationships from other points to produce their own output. This is essentially an abstraction of `ControlActor` in CTRLont. By making the Brick `Point` a subclass of CTRLont's `ControlActor`, every `controls` relationship can be further clarified using the Input-isConnectedTo-Output relationship and its logic can be specified by `ApplicationLogic` modules. As `Point` inherits the properties of `ControlActor`, the integration can inherit functionalities proposed by CTRLont such as the automated rule-based verification of control logic in BMS [82].

11.3. Electrical Power System (SEAS)

Smart Energy Aware Systems (SEAS) Knowledge Model [83] is an ontology aligning energy systems to existing ontologies such as SOSA (Sensor, Observation, Sample, and Actuator) ontology [84] and SAREF (Smart Appliances REFerence) ontology [19] and has several subdomains including electric power systems. Even though Brick has its own ontology for representing electrical power systems, we can increase the interoperability and portability of Brick by defining associations to other ontologies. In SEAS, `System`s are connected with each other through `Connection`s like a transformer is connected to a power consumer through a bus. In Brick's design, both `System` and `Connection` are a type of `Equipment` that can be monitored, controlled and functionally connected to each other.

Some inconsistencies can arise when integrating two different ontologies. For example, the SEAS ontology does not define a direction when connecting two entities (`connectedThrough`), whereas Brick does (`feeds/isFedBy`). A user may choose to keep both relationships in the unified model, which is enabled by a well-defined mapping of which concepts are common to the SEAS and Brick ontologies.

12. Conclusion

There are millions of buildings in the world and they constitute a major portion of the human energy footprint. With the growing efforts to mitigate climate change, numerous building energy efficiency solutions have been invented, tested and deployed. To have meaningful impact, these solutions must be deployed at a global scale; however, the heterogeneity of building representation presents a major bottleneck in fast and low cost deployment of energy efficiency measures.

We have designed the Brick schema as a strong candidate to solve this open problem. Brick builds upon prior work and introduces a number of novel concepts. Brick uses easy to understand Tags and TagSets to specify sensors and subsystems in a building. We define tags and tagsets in an ontology with class hierarchies. We define portable and orthogonal relationships between entities from an extensive list of smart-building applications. Relationships among entities are represented as triples, which allows users to leverage existing tools to build and query the resulting building representations. We showcase Brick's extensibility model as well as its interoperability with other existing schemata and ontologies. Finally, we have proposed practical methodologies to use Brick in the real world including how to convert existing unstructured and structured metadata into Brick and how to integrate Brick with actual building systems.

Brick is complete, capturing an average of 98% of BMS data points across six diverse buildings comprising almost 17,700 data points and 615,000 sq-ft of floor space. Brick is expressive, successfully running eight canonical applications on these buildings. Four applications ran on all the six buildings, while the remaining applications ran on buildings whose BMS exposed the requisite points. Brick is usable, as converting each of the buildings' legacy BMS metadata to the normalized schema took no more than 20 man-hours with semi-automated methods. Given structured metadata such as Haystack and IFC, the conversion process can be automatic. The resulting schema is understandable and easy to query as shown in Figs. 7–9. Brick was integrated with two example systems under a common BMS architecture while providing querying capability. Brick's extensibility model is already being put to the test by a growing user base.

Brick maintains orthogonality in describing tagsets and relationships, i.e. there is a single straightforward way to describe an entity, collection of entities and their inter-relationships. Open references of our six buildings provide a common platform to evaluate different schemata. The code, schema, and reference implementations of all the buildings in our testbed are available at http://brickschema.org/.

We hope that our solution to this well-defined open metadata problem lays the foundation for industry and academic collaboration to produce bona fide standards that could be transformative in producing energy efficient buildings and portable applications.

Acknowledgments

Funding: This work was supported by the National Science Foundation [Grant Nos. CPS-1239552, NSF-1636879, IIS-1636916, CSR-1526237, CNS-1526841, NSF-1305362]; the U.S. Department of Energy [Grant No. DE-EE0006353]; the King Abdullah University of Science and Technology award [Grant No. #OSR-2015-Sensors-2707]; the Innovation Fund Denmark [Grant No. 4106-00003B]; the EU H2020 [Grant No. 676760]; and Intel Corporation.

References

[1] Lucon O, Urge-Vorsatz D, Ahmed AZ, Akbari H, Bertoldi P, Cabeza L, et al. Climate Change 2014: mitigation of climate change – buildings. In: Contribution of working group III to the fifth assessment report of the intergovernmental panel on climate change.
[2] Granderson J, Piette MA, Ghatikar G. Building energy information systems: user case studies. Energ Effi 2011;4(1):17–30.
[3] Molina-Solana M, Ros M, Ruiz MD, Gómez-Romero J, Martin-Bautista M. Data science for building energy management: a review. Renew Sustain Energy Rev 2017;70:598–609.
[4] Roth KW, Westphalen D, Feng MY, Llana P, Quartararo L. Energy impact of commercial building controls and performance diagnostics: market characterization, energy impact of building faults and energy savings potential, Tech. Rep. TIAX LLC D0180, TIAX LLC, Cambridge, Massachusetts, USA, Aug. 2005.
[5] Liu X, Akinci B, Berges M, Garrett Jr. JH. An integrated performance analysis framework for HVAC systems using heterogeneous data models and building automation systems. Proceedings of the fourth ACM workshop on embedded sensing systems for energy-efficiency in buildings, BuildSys'12 New York, NY, USA: ACM; 2012. p. 145–52. http://dx.doi.org/10.1145/2422531.2422558.
[6] U.S. Energy Information Administration. User's guide to the 2012 CBECS public use microdata file, Commercial Buildings Energy Consumption Survey (CBECS); 2016, p. 33.
[7] Sturzenegger D, Gyalistras D, Morari M, Smith RS. Semi-automated modular modeling of buildings for model predictive control. BuildSys – ACM workshop on embedded sensing systems for energy-efficiency in buildings. ACM; 2012. p. 99–106.
[8] Piette MA, Ghatikar G, Kiliccote S, Koch E, Hennage D, Palensky P, et al. Open automated demand response communications specification (version 1.0), Tech. rep., Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US); 2009.
[9] Capozzoli A, Piscitelli MS, Gorrino A, Ballarini I, Corrado V. Data analytics for occupancy pattern learning to reduce the energy consumption of hvac systems in office buildings. Sustain Cities Soc 2017;35:191–208.
[10] Jahn M, Schwartz T, Simon J, Jentsch M. Energypulse: tracking sustainable behavior in office environments. Int. conf. on energy-efficient computing and networking. ACM; 2011. p. 87–96.
[11] Schein J, Bushby ST, Castro NS, House JM. A rule-based fault detection method for air handling units. Energy Build 2006;38(12):1485–92.
[12] Krioukov A, Dawson-Haggerty S, Lee L, Rehmane O, Culler D. A living laboratory study in personalized automated lighting controls. BuildSys - ACM workshop on embedded sensing systems for energy-efficiency in buildings. ACM; 2011. p. 1–6.
[13] Oti A, Kurul E, Cheung F, Tah J. A framework for the utilization of building management system data in building information models for building design and operation. Automat Construct 2016;72:195–210.
[14] NIST GCR. Cost analysis of inadequate interoperability in the US capital facilities industry. National Institute of Standards and Technology (NIST).
[15] Cerovsek T. A review and outlook for a 'building information model'(BIM): a multi-standpoint framework for technological development. Adv Eng Informat 2011;25(2):224–44.
[16] Bazjanac V, Crawley D. Industry foundation classes and interoperable commercial software in support of design of energy-efficient buildings. In: Building simulation 99, vol. 2; 1999. p. 661–7.
[17] Roth S. Open green building XML schema: a building information modeling solution for our green world, gbXML schema (5.12).
[18] Project Haystack. <http://project-haystack.org/>.
[19] Daniele L, den Hartog F, Roes J. Study on semantic assets for smart appliances interoperability: D-S4: Final report, Tech. rep., European Union; 2015.
[20] Schein J, Bushby ST, Castro NS, House JM. A rule-based fault detection method for air handling units. Energy Build 2006;38(12):1485–92. http://dx.doi.org/10.1016/j.enbuild.2006.04.014<http://www.sciencedirect.com/science/article/pii/S0378778806001034>.
[21] Weng T, Balaji B, Dutta S, Gupta R, Agarwal Y. Managing plug-loads for demand response within buildings. BuildSys - ACM workshop on embedded sensing systems for energy-efficiency in buildings. ACM; 2011. p. 13–8.
[22] Bhattacharya A, Ploennigs J, Culler D. Short paper: analyzing metadata schemas for buildings: the good, the bad, and the ugly. BuildSys - ACM conference on embedded sensing systems for energy-efficiency in buildings. ACM; 2015. p. 33–4.
[23] Balaji B, Bhattacharya A, Fierro G, Gao J, Gluck J, Hong D, et al. Brick: Towards a unified metadata schema for buildings. Proceedings of the 3rd ACM international conference on systems for energy-efficient built environments. ACM; 2016. p. 41–50.
[24] Fierro G, Culler DE. Xbos: An extensible building operating system. Proceedings of the 2nd ACM international conference on embedded systems for energy-efficient built environments. ACM; 2015. p. 119–20.
[25] BuildingDepot 3.0. <http://buildingdepot.org/>.
[26] U.S. Energy Information Administration. Monthly Energy Review. <https://www.eia.gov/totalenergy/data/monthly/#consumption>.

[27] Energy Star. Save energy. <https://www.energystar.gov/buildings/facility-owners-and-managers/existing-buildings/save-energy>.
[28] Agarwal Y, Balaji B, Dutta S, Gupta RK, Weng T. Duty-cycling buildings aggressively: the next frontier in hvac control. 10th Information Processing in Sensor Networks (IPSN). IEEE; 2011. p. 246–57.
[29] Goyal S, Ingley HA, Barooah P. Occupancy-based zone-climate control for energy-efficient buildings: complexity vs. performance. Appl Energy 2013;106:209–21.
[30] Roth KW, Westphalen D, Llana P, Feng M. The energy impact of faults in us commercial buildings. In: International refrigeration and air conditioning conference; 2004.
[31] Council on Finance, Insurance and Real Estate. Financing small commercial building energy performance upgrades: challenges and opportunities; 2016.
[32] Rawal G. Costs, savings, and roi for smart building implementation. <https://blogs.intel.com/iot/2016/06/20/costs-savings-roi-smart-building-implementation/>.
[33] Katipamula S, Underhill RM, Goddard JK, Taasevigen DJ, Piette M, Granderson J, et al. Small-and medium-sized commercial building monitoring and controls needs: a scoping study, Tech. rep., Pacific Northwest National Lab.(PNNL), Richland, WA (United States); 2012.
[34] Charpenay V, Kabisch S, Anicic D, Kosch H. An ontology design pattern for iot device tagging systems. 5th Int. conf. on the Internet of Things (IOT). IEEE; 2015. p. 138–45.
[35] W3C. Semantic Web. <https://www.w3.org/standards/semanticweb/>.
[36] Berners-Lee T, Hendler J, Lassila O, et al. The semantic web. Sci Am 2001;284(5):28–37.
[37] Ashburner M, Ball CA, Blake JA, Botstein D, Butler H, Cherry JM, et al. Gene ontology: tool for the unification of biology. Nat Genet 2000;25(1):25–9.
[38] Hachem S, Teixeira T, Issarny V. Ontologies for the internet of things. Proceedings of the 8th middleware doctoral symposium. ACM; 2011. p. 3.
[39] Zhang C, Romagnoli A, Zhou L, Kraft M. Knowledge management of eco-industrial park for efficient energy utilization through ontology-based approach. Appl Energy 2017.
[40] Gruber TR. Toward principles for the design of ontologies used for knowledge sharing. Int J Hum-Comput Stud 1995;43(5-6):907–28.
[41] Bonino D, Corno F. DogOnt – ontology modeling for intelligent domotic environments. In: ISWC – Int. Semantic Web Conf., vol. 5318; 2008. p. 790–803.
[42] Kofler MJ, Reinisch C, Kastner W. A semantic representation of energy-related information in future smart homes. Energy Build 2012;47:169–79.
[43] Ploennigs J, Hensel B, Dibowski H, Kabitzsch K. Basont-a modular, adaptive building automation system ontology. IECON – 38th an. conf. of ieee industrial electronics society. IEEE; 2012. p. 4827–33.
[44] Beetz J, Van Leeuwen J, De Vries B. IfcOWL: A case of transforming EXPRESS schemas into ontologies. Artif Intell Eng Des Anal Manuf 2009;23(01):89–101.
[45] Stavropoulos TG, Vrakas D, Vlachava D, Bassiliades N. Bonsai: a smart building ontology for ambient intelligence. Proceedings of the 2nd international conference on web intelligence, mining and semantics. ACM; 2012. p. 30.
[46] Tahir AC, Bañares-Alcántara R. A knowledge representation model for the optimisation of electricity generation mixes. Appl Energy 2012;97:77–83.
[47] Turtle. <https://www.w3.org/TR/turtle/>.
[48] SPARQL Query Language. <https://www.w3.org/TR/rdf-sparql-query/>.
[49] Simulink Simscape. <https://www.mathworks.com/products/simscape/features.html#multidomain-schematics/>.
[50] Fritzson P, Engelson V. Modelica a unified object-oriented language for system modeling and simulation. European conference on object-oriented programming. Springer; 1998. p. 67–90.
[51] Wetter M. Modelica-based modelling and simulation to support research and development in building energy and control systems. J Build Perform Simul 2009;2(2):143–61.
[52] Bernal W, Behl M, Nghiem TX, Mangharam R. Mle+: a tool for integrated design and deployment of energy efficient building controls. Proceedings of the fourth ACM workshop on embedded sensing systems for energy-efficiency in buildings. ACM; 2012. p. 123–30.
[53] Wetter M. Co-simulation of building energy and control systems with the building controls virtual test bed. J Build Perform Simul 2011;4(3):185–203.
[54] Ploennigs J, Clement J, Wollschlaeger B, Kabitzsch K. Semantic models for physical processes in cps at the example of occupant thermal comfort. 2016 IEEE 25th International Symposium on Industrial Electronics (ISIE). IEEE; 2016. p. 1061–6.
[55] Lassila O, Swick RR. Resource description framework (RDF) model and syntax specification.
[56] RDF Concepts Namespace. <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
[57] RDF Schema Namespace. <https://www.w3.org/2000/01/rdf-schema#>.
[58] OWL Namespace. <http://www.w3.org/2002/07/owl#>.
[59] Crawley DB, Lawrie LK, Winkelmann FC, Buhl WF, Huang YJ, Pedersen CO, et al. Energyplus: creating a new-generation building energy simulation program. Energy Build 2001;33(4):319–31.
[60] Brickley D, Miller L. Foaf vocabulary specification 0.91; 2007.
[61] Guha RV, Brickley D, Macbeth S. Schema. org: evolution of structured data on the web. Commun ACM 2016;59(2):44–51.
[62] Ashburner M, Ball CA, Blake JA, Botstein D, Butler H, Cherry JM, et al. Gene ontology: tool for the unification of biology. Nat Genet 2000;25(1):25.
[63] El Kaed C, Boujonnier M, Dillon S. Forte: A federated ontology and timeseries query engine. In: The 3rd IEEE international conference on smart data. IEEE, 2017.
[64] Marchiori A, Han Q. Using circuit-level power measurements in household energy management systems. BuildSys – ACM workshop on embedded sensing systems for energy-efficiency in buildings. ACM; 2009. p. 7–12.
[65] Jung D, Krishna VB, Khiem NQM, Nguyen HH, Yau DK. Energytrack: Sensor-driven energy use analysis system. BuildSys – ACM workshop on embedded sensing systems for energy-efficiency in buildings. ACM; 2013. p. 1–8.
[66] Balaji B, Koh J, Weibel N, Agarwal Y. Genie: a longitudinal study comparing physical and software thermostats in office buildings. Proceedings of the 2016 ACM international joint conference on pervasive and ubiquitous computing. ACM; 2016. p. 1200–11.
[67] Comfy. <https://www.comfyapp.com/>.
[68] Bidgely. <http://www.bidgely.com/>.
[69] SkyFoundry. <https://skyfoundry.com/>.
[70] EnerNOC. <https://www.enernoc.com/>.
[71] KGS Buildings. <http://www.kgsbuildings.com/>.
[72] Balaji B, Verma C, Narayanaswamy B, Agarwal Y. Zodiac: Organizing large deployment of sensors to create reusable applications for buildings. BuildSys – ACM conference on embedded sensing systems for energy-efficiency in buildings. ACM; 2015. p. 13–22.
[73] Hong D, Wang H, Whitehouse K. Clustering-based active learning on sensor type classification in buildings. Proceedings of the 24th ACM international on conference on information and knowledge management. ACM; 2015. p. 363–72.
[74] Gao J, Ploennigs J, Berges M. A data-driven meta-data inference framework for building automation systems. BuildSys – ACM conference on embedded sensing systems for energy-efficiency in buildings. ACM; 2015. p. 23–32.
[75] Pritoni M, Bhattacharya AA, Culler D, Modera M. Short paper: A method for discovering functional relationships between air handling units and variable-air-volume boxes from sensor data. Proceedings of the 2nd ACM international conference on embedded systems for energy-efficient built environments. ACM; 2015. p. 133–6.
[76] Bhattacharya A, Hong D, Culler D, Ortiz J, Whitehouse K, Wu E. Automated metadata construction to support portable building applications. BuildSys – ACM conference on embedded sensing systems for energy-efficiency in buildings. ACM; 2015. p. 3–12.
[77] Niagara AX. <https://www.tridium.com/en/products-services/niagara-ax>.
[78] Andersen MP, Fierro G, Culler DE. Enabling synergy in iot: Platform to service and beyond. J Network Comput Appl 2017;81:96–110.
[79] Fierro G, David C. Design and analysis of a query processor for Brick. In: BuildSys – ACM conference on embedded sensing systems for energy-efficiency in buildings. ACM; 2017.
[80] QUDT. <http://qudt.org/>.
[81] Steinberg MD, Schindler S, Keil JM. Use cases and suitability metrics for unit ontologies. International experiences and directions workshop on OWL. Springer; 2016. p. 40–54.
[82] Schneider GF, Pauwels P, Steiger S. Ontology-based modeling of control logic in building automation systems. IEEE Trans Indust Informat 2017;13(6):3350–60.
[83] Lefrançois M. Planned ETSI SAREF extensions based on the W3C&OGC SOSA/SSN-compatible SEAS Ontology Patterns. In: Proceedings of workshop on semantic interoperability and standardization in the IoT, SIS-IoT; 2017.
[84] W3C, SOSA Ontology. <https://www.w3.org/2015/spatial/wiki/SOSA_Ontology/>.