UNIVERSITY OF CALIFORNIA SAN DIEGO

Metadata Models and Methods for Smart Buildings

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Jason B. Koh

Committee in charge:

Professor Rajesh K. Gupta, Chair
Professor Yuvraj Agarwal
Professor David E. Culler
Professor Farinaz Koushanfar
Professor Julian McAuley

2020

The dissertation of Jason B. Koh is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2020

DEDICATION

To my wife, parents, sister, and loving ones.

EPIGRAPH

Extraordinary claims require extraordinary evidence!

— Carl Sagan

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to first acknowledge Professor Rajesh Gupta and Professor Yuvraj Agarwal. Rajesh Gupta has shown tremendous support on my ideas whether they were a rough sketch or a solid picture. At the same time, he never let me be satisfied with only conceiving an idea but fully implementing it. His encouragement has led me to fearlessly study, implement, and verify all the ideas. Some ideas worked and others did not, but, regardless, I was able to learn every corner of the ideas and theories, which has become the foundation of my analytical skills along with this dissertation.

Yuvraj Agarwal was a foreseeing advisor. He has been serious about building things for research, and such experience has guided me to ask really important and interesting problems. Furthermore, I learned that any important problem can be expanded to other interesting problems, and my entire journey has been joyful with a series of undiscovered problems with a grand vision.

Special thanks to my Ph.D. dissertation committee members David Culler, Julian McAuley, and Farinaz Koushanfa for serving in my committee and providing invaluable advice in my dissertation. I will carry it over my future work as well.

Bharathan Balaji, who was a senior student in the lab, taught me every technical detail that I needed to know for my dissertation. He was unparalleled in patience when I ask questions. No matter how silly or how many the questions were, he stayed with me and clarified every bit of the questions for me. I promise I will do the same when I am asked.

When Dezhi Hong joined our lab as a postdoctoral scholar, I felt like I got a strong ally as well as a close friend. Fully equipped with the right experience and the necessary knowledge, he has helped me to do my research in the right manner. More importantly, he always listens to me as an advisor and a friend.

UCSD's Facilities Management has generously provided all the required resources for my building research. John Dilliott allowed me to continue my research in buildings even under difficult circumstances. Without Robert Austin, I would have been lost before finding the right person or resource to solve the problem in building systems. Michelle Perez gave me the real questions for campus energy management, which kept my feet on the ground.

I would like to thank all the Brick initiators and the community. My dissertation is all about how to put together commonly shared knowledge in buildings, and it would have been impossible without the team effort. I was really lucky to have the smart, diligent, and motivated colleagues for Brick. I will continue working with the team and devote myself to Brick's future.

My two internship mentors, Jack Hodges at Siemens and Zhongyi Jin at Johnson Controls, assured me that I was solving the right problems for the industry and the society. It drove me further to expand the ideas in the dissertation without hesitation.

I was also lucky to work with brilliant junior students: Honam Bang, Jinmou Li, Renxu Hu, Yutao Lee, Yiming Yang, Kuo Liang, Paul Minsub Song, Naveen Kashyap, Xiaohan Fu, and Tejaswi Tanikella. Not only they all did a great job at given tasks, but also I learned a lot from them about how to work as a supervisor with other people, based on their sincere feedback and active communications.

Since my third year in the Ph.D. program, I have been happily married to Eunjeong Koh, which was the best choice in my life. She has been a spouse, a friend, a supporter, a colleague, or everything I needed in the past years. She has been always on my side and I will do the same for her.

Most of all, I cannot thank enough my parents, Sungsune Wang and Kyungchul Koh. They made me trust myself as they have always trusted me with the highest bar. I cannot measure the love they gave me; I can only try to return what I received to them. Last but never least, I thank my sister, Hyeyoung Koh. We have been a companion to each other in the hardest times and will be, as a family.

I also acknowledge all the co-authors of the works included in this dissertation sincerely.

Chapter 3, in part, is a reprint of the material as it appears in Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments, 2016 by

authors Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjrgaard, Mani Srivastava, and Kamin Whitehouse. The dissertation author is one of the primary investigators of this paper.

Chapter 3, in part, is a reprint of the material as it appears in Applied Energy, 2018 by authors Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjrgaard, Mani Srivastava, and Kamin Whitehouse. The dissertation author is one of the primary investigators and the corresponding author of this paper. This is a journal extension of the above paper.

Chapter 3, in part, also contains material as it appears in Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation, 2019 by authors Gabe Fierro, Jason Koh, Yuvraj Agarwal, Rajesh K. Gupta, David E. Culler. The dissertation author was the second author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of the 5th ACM International Conference on Systems for Built Environments, 2018 by authors Jason Koh, Bharathan Balaji, Dhiman Sengupta, Julian McAuley, Yuvraj Agarwal and Rajesh Gupta. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, also contains the material as it appears in arXiv, 2016 by authors Jason Koh, Bharathan Balaji, Vahideh Akhlaghi, Yuvraj Agarwal and Rajesh Gupta. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, is a reprint of the material as it appears in Proceedings of the 5th ACM International Conference on Systems for Built Environments, 2018 by authors Jason Koh, Dezhi Hong, Rajesh Gupta, Kamin Whitehouse, Hongning Wang, and Yuvraj Agarwal. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in part, contains material as it appears in Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation, 2019 by authors Jason Koh, Dezhi Hong, Shreyas Nagare, Sudershan Boovaraghava, Yuvraj Agarwal and Rajesh Gupta. The dissertation author was the primary investigator and author of this paper.

VITA

2014	B. S. in Electrical and Computer Engineering *cum laude*, Seoul National University, Seoul, South Korea
2017	M. S. in Computer Science (Computer Engineering), University of California, San Diego
2020	Ph. D. in Computer Science (Computer Engineering), University of California, San Diego

PUBLICATIONS

Jason Koh, Dezhi Hong, Shreyas Nagare, Sudershan Boovaraghava, Yuvraj Agarwal and Rajesh Gupta, ”Who can Access What, and When?” In Proceedings of *the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation* (BuildSys), 2019.

Gabe Fierro, Jason Koh, Yuvraj Agarwal, Rajesh K. Gupta, David E. Culler, ”Beyond a House of Sticks: Formalizing Metadata Tags with Brick” In Proceedings of *the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation* (BuildSys), 2019.

Jason Koh, Dezhi Hong, Rajesh Gupta, Kamin Whitehouse, Hongning Wang, and Yuvraj Agarwal, ”Plaster: An Integration, Benchmark, and Development Framework for Metadata Normalization Methods.” In Proceedings of *the 5th ACM International Conference on Systems for Built Environments* (BuildSys), 2018.

Jason Koh, Bharathan Balaji, Dhiman Sengupta, Julian McAuley, Yuvraj Agarwal and Rajesh Gupta, ”Scrabble: Transferrable Semi-Automated Semantic Metadata Normalization using Intermediate Representation.” In Proceedings of *the 5th ACM International Conference on Systems for Built Environments* (BuildSys), 2018.

(Alphabetically) Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjærgaard, Mani Srivastava and Kamin Whitehouse, ”Brick : Metadata schema for portable smart building applications.”, *Applied Energy*, 2018.

Jason Koh, Steven Ray, and Jack Hodges, ”Information Mediator for Demand Response in Electrical Grids and Buildings.” In Proceedings of *the 11th International Conference on Semantic Computing* (ICSC), 2017.

(Alphabetically) Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjærgaard, Mani Srivastava, and Kamin Whitehouse, ”Brick: Towards a unified metadata schema for buildings.” In Proceedings of *the 3rd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments* (BuildSys), 2016.

Bharathan Balaji, Jason Koh, Nadir Weibel, and Yuvraj Agarwal, ”Genie: A Longitudinal Study Comparing Physical and Software-augmented Thermostats in Office Buildings.” In Proceedings of *ACM International Joint Conference on Pervasive and Ubiquitous Computing* (UbiComp), 2016.

ABSTRACT OF THE DISSERTATION

Metadata Models and Methods for Smart Buildings

by

Jason B. Koh

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2020

Professor Rajesh K. Gupta, Chair

Applications in smart buildings have shown potential for improving energy efficiency, automated operation, and for creating better living conditions for occupants. To achieve these goals requires effective collection and use of sensing data and collaboration among different subsystems such as Heating, Ventilation, Air-condition (HVAC), security, lighting and sensing subsystems. Data generated and used by these subsystems are heterogeneous and often contextualized to real-time conditions. Contextual information is often provided by subsystem vendors as "metadata", that is, the data about data. The multiplicity of vendors makes most metadata idiosyncratic without any consistent meaning or usage that can be directly inferred from such metadata. This makes them difficult to be useful. Vendors and building operators have to often

“guess” based on the unstructured text of metadata written by different engineers. Converting building metadata to a machine-readable format usually involves significant manual effort. We envision building systems that are able to seamlessly exchange data across subsystems as well as across various building services in a programming framework. Such information exchange is mediated by timely sensor information, its automated organization and navigation, thus creating a technical basis for future ‘smart buildings’.

Methods and tools for automated handling of metadata are crucial to this vision. Second, we present an application programming framework comprised of machine learning algorithms to help organize the current unstructured metadata information from existing buildings into a structured format such as Brick+. Third, we propose an application workflow that relies only on a standard information model for unified and secure application deployment. Using Brick+, Scrabble and Plaster programming support tools, we have built an end-to-end applications and services framework for smart buildings. Using buildings on the UC San Diego campus, we describe and demonstrate the effectiveness of the proposed methods. We demonstrate several new applications, such as a personal thermostat application called Genie and an energy dashboard, that can be built and deployed with minimal human effort. In addition to the demonstrated value of metadata models and methods in building portable applications for smart buildings in this dissertation, we continue to pursue building a community of system builders for the smart building environments.

Chapter 1

Introduction

The Internet of Things, or IoT, refers to embedded devices with sensing, computing and actuation capabilities that are accessible through a wired or wireless network connection. These devices are growing rapidly, and estimates place 20 to 30 billion devices deployed by 2020 [BPV14]. As these devices find ways into daily life, we envision a responsive living environment that provides necessary creature comforts as well as operational efficiencies and effective response to emergencies. For example, fire prediction algorithms utilize weather data from geographically distributed IoT sensors and satellite images to identify conditions when wildfires could break out and predict their progress to be useful in firefighting and eradication plans [GTA+15]. Thus, the potential impact of these IoT use cases is not just limited to gathering and analyzing data, instead in integrating these disparate data sources and systems to work together seamlessly for a larger goal. For example, once a fire is predicted, there are a number of key steps that need to happen. This includes fire stations need to be notified, contingency plans need to be executed over nearby building systems, traffic control systems should be managed to enable residents to evacuate in an orderly manner, etc. To address these real-world use cases, numerous applications (apps) have been proposed and developed such as automatic traffic control [ZWW+11] and intelligent control of smart buildings [BBF+18a]. These apps use various types of resources such as data collected from sensors (e.g an occupancy sensor) and configuration parameters (e.g. a user's temperature setpoint) that control certain devices (e.g. the air temperature and volume provided into the occupants' room). While there are numerous opportunities enabled by capable apps in IoT scenarios, their sheer number and heterogeneity poses many challenges.

A vast majority of these challenges are related to how different data sets – sensor generated, inferred or derived – are composed, and analyzed for useful inferences and control actions. Most sensory data are highly contextual. For instance, an airflow sensor's data is meaningful only when accompanied by the information about its location, time and other conditions. Contextual information is typically captured through additional data, attributes and annotations that are collectively referred to as "metadata", that is, data about data. Metadata is an old concept and its most recent use in software systems has been in building middleware systems such as CORBA [ZM95], DCOM [HK97].

In the IoT context, metadata often captures information about the sensor/actuator type, location, time and other conditions. Typically, such information is encoded in a textual string of characters and numbers. Lack of structure in metadata sometimes makes it hard or error prone to infer the real context intended by the metadata. For example, while a temperature sensor may produce a stream of data consisting of time-stamped values such as 70 at 2019-12-10T10:30:00, the values do not convey their actual meanings. Rather, the sensor's metadata can indicate that its measurement `type` is temperature, the `unit` is Fahrenheit, and it is `located in` a certain physical space in a building.

Metadata is present in diverse domains. For example, it is used for similar purposes in Web search. While a search engine algorithm, such as Google's, statistically exploits the links in the contents of websites to answer user queries [PBMW99], it also utilizes the websites' metadata to provide more precise results matching a user's queries using `schema.org` [Gooa, sch]. Google's search engine checks if the metadata of a target web page contains a specific keyword, "Movie" predefined at `https://schema.org/Movie`, which tells if the web page is unambiguously about movies. By leveraging metadata, the web search algorithm is more deterministic than the statistical analysis of the webpage contents alone, thereby providing more precise results to the user.

Effective use of metadata is even more critical in IoT devices because the contents (i.e., values) of IoT devices have less contextual information due to the limited computing/storage capacities of the IoT devices. These are also often much more ambiguous than, say, the descriptive text or videos used on websites. Unlike a website that can make inferences about user behaviors based on longitudinal data (using "cookies", for instance, which can be seen as carriers of metadata), IoT devices as sensor endpoints simply lack that capability. For instance, we may logically infer that the temperature data of a room would have a similar pattern to the adjacent room if the occupants in the two rooms have similar working schedules. However, such inferences by the endpoints are architecturally difficult or technically impossible in most deployments. A software and programming framework that uses metadata effectively can, in principle, help make such inferences possible.

Structure of metadata is important to its systematic (and automated) use by various applications. A structured metadata conforms to a well-defined *schema*. A schema consists of a set of *types* associated with the objects and *rules* that govern the use of these objects in a particular scheme. For instance, a website consists of webpages as objects that are composed through hypertext links and `schema.org` provides vocabularies and rules to annotate such objects and the rules. In general, a schema refers to a pattern of use of concepts in a given model though well-defined syntax and rules that conform to semantics derived from the target application domain.

In reality, the syntax of metadata may vary across systems as well as the vocabularies in the metadata may be specific to the target system. Metadata could be human-readable text or a predefined codebook that is specific to a broad class of systems. For instance, the term "T" may stand for temperature in the context of building applications. Real-life practice of metadata leaves much to be desired. Different vendors and system integrators have used their own schemata, at

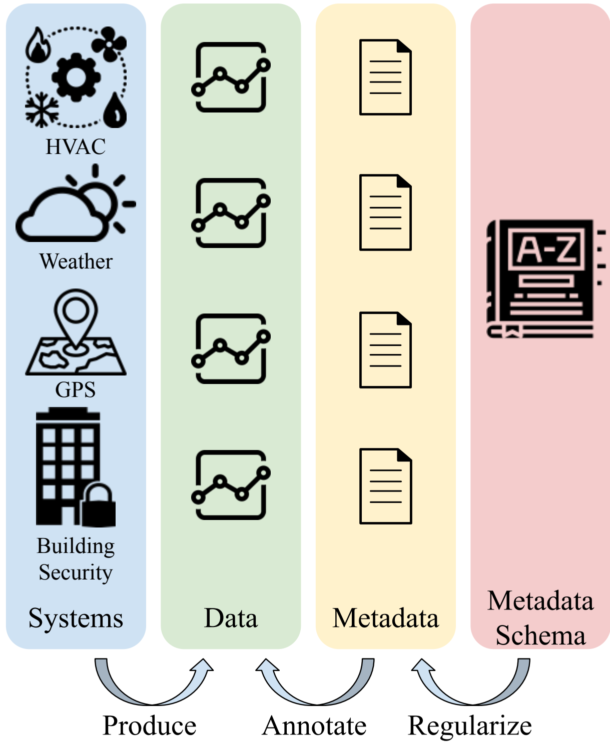


Figure 1.1: Relationships between systems, data, metadata, and metadata schema.

best, to represent metadata, which creates vendor lock-in. However, the recent change towards open systems and interoperability have brought to bear the problems with both textual description and vendor-specific schemata. Moreover, different systems need different types of metadata depending on the system requirements. For example, while image data-sets need to present the images' encoding schemes (e.g., JPEG, PNG, etc.), temperature sensor data do not need such information but rather units of the values and the location of the measurement. Given the diversity of applications, and ambiguity inherent in natural language descriptions, a metadata schema is an important need. While metadata schemata have various formats, their common goal is to bind the information of interest such as image encoding schemes or data units to a designated part of metadata, and users can uniformly refer to the part to obtain necessary information.

Figure 1.1 describes the relationships among systems, data, metadata, and metadata schema. A metadata schema regularizes metadata which users would refer to for understanding the meaning of the underlying data. In other words, metadata schema is the starting point to address the deluge of data produced by different systems. This observation leads to our key argument: to make use of the large data produced by IoT devices, we need to treat a metadata schema as a first-class object in the system design and application workflows.

There are a number of questions that must be answered to design an effective metadata schema: (a) What is the right metadata schema for a system composed of a set of IoT devices? (b) What is the right methodology to organize existing data and metadata with the metadata schema? and (c) What are the right means to access data based on the metadata schema?

This dissertation explores the above questions within the context of *commercial buildings*. Buildings are a good example of emerging cyber-physical systems that provide a platform for interaction between physical systems (such as HVAC), humans and computing systems. These are also significantly energy-intensive thus providing a good application platform for a substantial impact on societal energy use: buildings account for 40% of the entire energy use of the United States [U.S] and people spend 87% of their time indoors [KNO+01]. Occupants interact with buildings in numerous ways. For example, office workers interact with other workers in office buildings and patients are treated by health care providers in hospitals. To support various usage modalities of different types of buildings, they have diverse subsystems such as Heating, Ventilation, and Air Conditioning (HVAC), lighting, elevators, and security systems, which are often very energy-intensive.

Pursuing the vision of “smart buildings”, the holy grail of CPS computing infrastructure supported by application software (app) is one in which developers write apps that are independent of a specific building, and rather the same app can be run on *any* building without much customization, if any at all. However, there is a large gap between this vision and reality. Porting an app written for a specific building to run on another building is currently a complicated process requiring domain-specific knowledge about the target system and significant manual effort by domain experts and building managers. Traditionally, the purpose of metadata in building systems was not for programs to interface with the Building Management Systems (BMS's) but for building managers to read through the metadata to identify the location of resources when a fault occurred in a system or its configuration needed to be changed. Thus, existing metadata are usually only in human-readable formats where conventions vary a lot based on how and when a building was commissioned. Metadata authors commonly use technical jargon that is often abbreviated into a couple of characters. Moreover, the jargon is specific to the target building because buildings are different in terms of functionality and implementation. For instance, buildings in Southern California region often have simpler or nonexistent heating systems while those in Wisconsin have elaborate heating systems due to the harsher winters. Even with a similar weather condition, buildings may implement cooling systems with different types of configurations such as centralized and decentralized HVAC systems from different vendors.

Such heterogeneity in metadata and systems results in manually processing existing metadata into a standard format that an app can understand even when a common schema is available. Managers or developers deploying a smart building app need to understand the target building system configuration (e.g., how are the subsystems interconnected?), the source of the information the app needs (e.g., in which metadata is measurement type encoded?), actual meaning of the codes used (e.g., what does "T" stand for?), and relationships across resources (e.g., which room is this temperature sensor associated with?). Existing metadata schemes are further exemplified in Section 2.2. As is common today, a domain expert would provide a large set of rules to parse the existing metadata and the interpreted results are hard-coded per app. This practice not only involves extensive upfront manual effort but is also unsustainable as the understanding of metadata could drift over the lifetime of the building as it pertains to different apps and system components — the system could have been updated or human knowledge corrected.

These observations inform our working thesis; *a large scale building application deployment and management over heterogeneous systems and devices can be built upon structured* *metadata designed for the context of applications and metadata management tools.* Buildings and associated systems are inherently heterogeneous, for which applications should be adaptive enough to be portable and interoperable with each other. A metadata schema should be the base reference for such adaptation as it is needed for software to be operational automatically. An end-to-end smart building infrastructure entirely based on a metadata schema, without hand-crafted data/system integration, requires advances in three key dimensions: (a) design of a correct and comprehensive schema for large-scale application development; (b) methods to organize building metadata around this schema, and algorithms to translate existing building metadata into this schema with minimal human effort; (c) and finally, a workflow to control applications' access of resources with the proposed metadata schema.

1.1 Related Work

Ontologies have been proposed as a means to present information that can be used across systems. Ontologies are defined as "an explicit specification of a conceptualization [Gru95]". We can also use ontologies as a metadata schema, thus, we use metadata and ontologies interchangeably. Researchers have studied ontologies for various domains from general things [GGM+02] and biology [Con04] to sensor networks [CBB+12, JHC+19a]. For example, the Semantic Web utilizes ontologies to uniformly represent information in the Web [BLHL+01]. In the Semantic Web, users rely upon Resource Description Framework (RDF) [RDFa] and Web Ontology Language (OWL) [OWL]. They together standardize concepts for generalization and specialization and essential properties entities can have.

Based on the Semantic Web framework, Semantic Sensor Network Ontology (SSN) [CBB+12] comprehensively models sensor network entities and their properties. SSN models any sensors' behaviors as the (phenomenal) stimulus, sensors, and observation pattern, and it has become the base of all the sensor-based models such as smart cities [AGM15], smart buildings [BBF+16a], and the Internet of Things [AFE+16]. Extending SSN, Sensors, Observations, Samples, and Actuators Ontology (SOSA) introduce the concept of sampled data and actuation [JHC+19b]. SSN and SOSA provide a basic framework to model interaction between sensing/actuating devices and the physical world.

IoT applications by nature demand incorporating heterogeneous systems possibly equipped by different vendors and users. For IoT devices, various standards, such as Open Connectivity Foundation (OCF) [ocf], the Web of Things (WoT) [Web19], and `iot.schema.org` [BAD+16b], attempt to provide frameworks for IoT modeling. OCF's information model mainly represents entities with "resource types" and their "interfaces". The resource types are drawn from various systems such as health care (e.g., glucose, heart rate), buildings (e.g., humidity, lock) and energy (e.g., PV system connection terminal, inverter). However, OCF's model is not systematically extensible because it is merely a set of enumeration values. It lacks in maintainability, that is, new concepts cannot be interpreted and relationships between concepts are unknown. Furthermore, OCF specification is defined in bare texts such as PDF files and OpenAPI specification[1] not suitable for programmatic queries and knowledge management. WoT is a collection of protocols to exchange information based on a formal description of Things.

WoT serves as a framework to model Things with their capabilities such as types of properties and actions. `iot.schema.org` provides vocabularies to represent such properties (e.g., humidity, illuminance) and actions (e.g., turnOff, countDown). Both OCF and WoT lack in organizing knowledge at scale. They represent what Things could do but do not maintain the relationships among the concepts, and thus force the users to rely on either specific keywords in the standards or custom metadata outside of the standards.

For buildings, Industrial Foundation Classes (IFC) and Project Haystack (Haystack) are widely-adopted metadata schemata, though they are not officially aligned with the Semantic Web framework. IFC was firstly introduced in 1996 and has become an ISO standard 16739-1:2018 [2].

[1] `https://swagger.io/docs/specification/about/`

[2] `https://www.iso.org/standard/70303.html`

It can describe details of construction information such as the widths of walls, the proximity between spaces, architectural materials, etc. However, many of the applications are agnostic to such details but rather need a description of dynamic data such as sensors/actuators and their associated equipment. Even though IFC has attempted to model such concepts, the vocabulary set is rudimentary. Furthermore, its file format is highly customized and not interoperable with existing data exchange protocols, and thus lacks in programming interface. We will discuss IFC in-depth in Section 2.3.1.

Haystack is an open-source standard for modeling building systems with tags. It started in 2014, with a goal to simplify and standardize building data models. Haystack defines tags, their meanings, and a data format for associating entities with the tags. While it is easy to understand Haystack's meta-model, it is fragile as users are free to associate the tags with any concepts, which can vary across different users. This unverifiable flexibility eventually deteriorates the interoperability across models. We discuss this trade-off in-depth at Section 2.3.2.

There are other notable building metadata schemata. Haystack Tagging Ontology (HTO) [CKAK15a] attempts to provide a semantic structure to Haystack tags in parallel. Their "Domain Model" governs the possible relationships Haystack tags can have. Despite this formality, HTO lacks in structurally specializing and extending concepts outside their domain model, thus, limiting expressivity. Building Topology Ontology (BTO) [bot] is specialized in modeling topological concepts in buildings, with which it could complement other models such as Haystack. We present our unified metadata schema, Brick, in Chapter 3.

While metadata schemata could provide a uniform interface to buildings, it takes a lot of effort of instantiating standardized models from actual buildings. Existing metadata encode information without a standard such as "RM-ZNT" for zone temperature in a room, and, even worse, they are idiosyncratic across different buildings and even in a building. many algorithms have been proposed to ease the instantiation process by exploiting different types of information sources. They use raw metadata [BHC+15a, HWW15, BVNA15a, LLHW19], timeseries data [GPB15a, HWOW15a], and active system perturbation [HOWC13, KBA+16, PBCM15b] in different frameworks such as active learning, transfer learning, and supervised learning. We discuss existing algorithms in detail at Section 4.9. Similar problems have been studied in data mining and database research communities as well [Chr12a], but metadata in buildings are differed with its higher heterogeneity. Each building has its own convention and unique system configuration. Furthermore, required label types also vary across different apps. Thus, it is difficult to reuse the knowledge learned from a building system for other buildings. We will present our approaches for organizing metadata with minimal human effort in Chapter 4 and a general framework in Chapter 5.

1.2 Contributions

We envision that buildings can be a programmable platform where users and operators can develop applications deployable at scale, and users can easily install the right apps matching their demands, similar to installing smartphone apps on a smartphone. To realize this vision, first, we present a metadata schema, **Brick+**, for programmable buildings. Second, To easily map existing buildings into Brick+, we present a set of novel algorithms to normalize existing metadata into Brick+ at scale. Third, we standardize the programming pattern of the algorithms, to integrate and utilize them easily in different contexts, along with a practice user interface of the algorithms. Finally, we present a dynamic authorization workflow with a comprehensive access control model to guarantee the least privilege to apps written with Brick+, which we exemplify with actual apps.

We present Brick [BBF+16a, BBF+18a] and its successor, Brick+ [FKA+19], as a base model for semantic programmability over heterogeneous entities in buildings. Brick represents entities and their relationships in buildings. Its query mechanism is flexible to cover a vast majority of the possible patterns building applications may need. Unlike existing schemata, Brick is consistently extensible and usable at the same time. As soon as we presented Brick initially in

2016, it drew in a community of different types of users, including building system vendors, IoT system integrators, and building managers. As a result of discussions in the community, we have updated Brick for consistent extensibility and formal description of the semantics. Brick is now beginning to be adopted by companies and research institutions and has formed working groups in different standard bodies.

While Brick's metadata schema enable semantic programmability over buildings, instantiating a whole system takes much effort because of the heterogeneity in different buildings, analogous to porting an Android OS to actual devices. Domain experts need to interpret existing metadata in buildings and manually convert them into Brick. We discuss the difficulty of the problem in Section 2.2.

We propose a set of metadata methods with machine learning algorithms to easily map existing buildings into Brick. The methods help users to adopt Brick in practice by converting existing buildings into Brick with a minimal effort. There are different aspects of metadata, of which each is represented in different data sources in buildings.

Scrabble [KBS+18] is a novel algorithm to parse raw metadata into Brick labels. It utilizes an intermediate representation of actual Brick tags to improve generalizability of a learned model across different buildings, while buildings have different styles and configurations. Scrabble shows the best performance in the category of the algorithms. However, Scrabble can extract only the information presented in raw metadata.

Often, raw metadata solely represents entities' functions but not their relationships across each other. **Quiver** [KBA+16] utilizes system perturbation to detect the co-location of different sensors in real buildings, which may not be encoded in the raw metadata. With the minimal perturbation of critical control points only, Quiver can co-locate different sensing and control points of Variable Air Volume Units (VAVs) 100% precision and 63% recall.

While studying various algorithms, we have identified a common programming pattern in metadata normalization. A common programming pattern can help 1) development of new algorithms, 2) comparison between similar algorithms, and 3) integration of different algorithms. It is similar to the role of `scikit-learn` in Python. `scikit-learn` provides a standard programming interface to machine learning algorithms in Python, helping their usability. We present **Plaster**, a standard meta-framework for metadata normalization where several algorithms, including Scrabble and Quiver, are implemented and compared. Then, we further present Plaster User Interface (UI) that provides an usable interface for non-programmer users. All the methods are essential for adopting Brick in the real world.

Apps need different types of resources in diverse contexts and representing their access requirements is a must for practical deployment of building apps. We propose an access control model using both Brick and other data sources based on our analysis of 125 building apps' requirements [KHN+19]. We have also observed that it is required to evaluate apps' access patterns at runtime because of the apps' dynamic nature and multi-tenancy. For online evaluation workflow, we propose Dynamic Dual Authorization (DDA) workflow. We then demonstrate the entire workflow via two apps, a Web thermostat and an energy dashboard, showing the feasibility of our metadata models and methods for building app development and deployment.

All the schema and the source code are open-source, and we are devoted to developing our work further through community contributions.

- Brick: `https://brickshema.org`
- Brick Server: `https://github.com/brickschema/brick-server`
- Plaster Service: `https://plaster.ucsd.edu`
- Plaster Python Package: `https://github.com/plastering/plsatering`
- Plaster UI Source Code: `https://github.com/plsatering/plastering-ui`

With the various set of frameworks and algorithms, we can achieve a seamless connection from entities, i.e., data sources, to apps or programs that can benefit people around the world.

We believe it is an important mission to improve our environment all around and people's living conditions, where metadata models and methods play a crucial role.

1.3 Organization

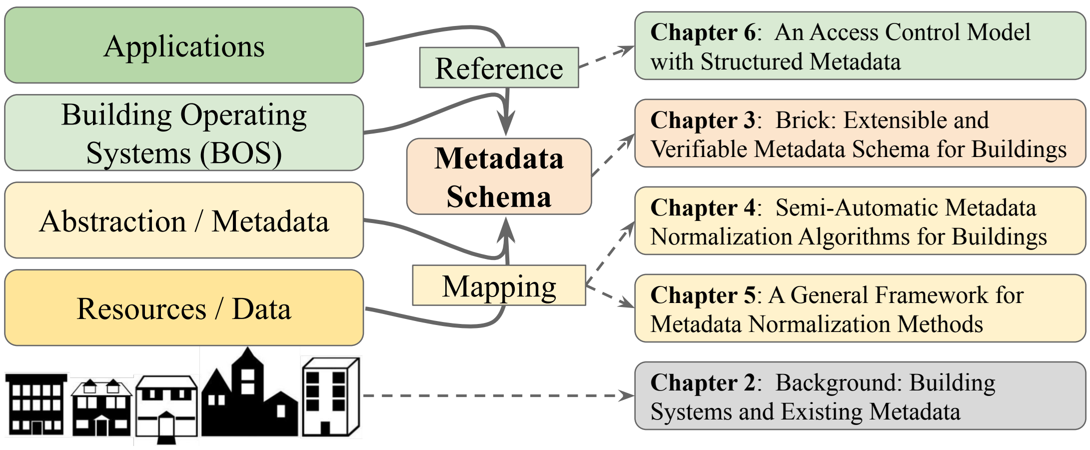


Figure 1.2: An Overview of the Thesis Organization

This dissertation is organized as follows. In Chapter 2, we detail the necessary background knowledge for understanding building systems. We explain the workflow for apps to access required resources through BOSes as well as the role of metadata in the workflow. We present the challenges in metadata for buildings via exemplifying actual metadata from existing buildings. Then, we show how two major existing metadata schemata, IFC and Haystack, fail at comprehensively representing entities in buildings.

In Chapter 3, we present Brick, a unified metadata schema for buildings, and its evolution, through various feedback, to Brick+. We initiated Brick with the other six institutions sharing the same motivation; we need to be able to represent buildings for apps. Brick has a class hierarchy where each class precisely represents a category of entities. Thus, users can consistently instantiate classes to model entities in their buildings. Their associated tags collectively present the meanings of classes. We show Brick's expressivity and completeness over six different buildings and eight apu categories. Following the Semantic Web framework, Brick's ecosystem reuses existing technologies for querying, data management, and data integration.

In Chapter 4, we present two metadata normalization algorithms, Scrabble and Quiver, for different use cases. Scrabble is an active learning framework which can reuse the knowledge across similar buildings. It first maps raw metadata to an intermediate representation (IR), Brick Tags, and then map them into actual labels, Brick TagSets. With IR, Scrabble can reuse existing knowledge from other buildings to normalize a new building. At the same time, its active learning algorithm can gradually improve the accuracy at the new building with the help of a domain expert. Quiver is an active perturbation framework to co-locate different points and find causality between them. Because different rooms work in a very similar schedule, their properties are indistinguishable in historical data. Quiver provides a framework to safely perturb existing systems to improve the observability of relationships in the data. We evaluate the algorithms over different buildings and different rooms, and they show the best-in-class performances.

In Chapter 5, we present Plaster, a meta-framework for metadata normalization. Throughout our experiences on metadata normalization, we identify the commonality across different frameworks. We define the common programming model in Plaster so that we can compare or integrate different algorithms. We detail our design, evaluate different algorithms with the same data set, and show the possible benefits from integrating different algorithms. Based on the Plaster programming interface, we also present a Web-based user interface and its architecture.

In Chapter 6, we present our analysis of access control models over 125 app papers and propose a manageable access control model based on the analysis. Apps are heterogeneous and have different requirements, and thus a single access control model or naïve access control list cannot simply express the requirements with which human managers can deal. Our analysis shows three information dimensions as Who, What, and When are essential for expressing access requirements. To represent all the information dimensions, our access control patterns (ACPs) accommodate diverse data sources through federating databases. The identified access control model requires online evaluation of access policies instead of static delegation. Our proposed Dynamic Dual Authorization (DDA) workflow provides a tight bound even for multi-tenancy apps with ACPs. We exemplify this workflow with two actual applications as Genie, software web thermostat, and VEnergy, and energy dashboard.

We discuss future work in Chapter 7 and conclude in Chapter 8. Figure 1.2 visualizes the overview of our dissertation.

Chapter 2

Background: Building Systems and Existing Metadata

2.1 Building Systems and Applications

We first introduce a general workflow of apps in buildings as background information of this dissertation. The goal of any building apps is to understand the environment, provide insights, and possibly control buildings better for their optimal use.

For its occupants, buildings need to serve various functions with several subsystems such as Heating, Ventilation, and Air Conditioning (HVAC) systems, lighting systems, elevators, security systems, computer rooms, and energy storage. The example configuration in Figure 2.1 has two subsystems as HVAC and lighting systems. There are various types of components and relationships to completely represent the systems. First, there are two rooms and each of the rooms is differently served by devices or equipment. The Variable Air Volume Box (VAV) serves both of them whereas the luminaire illuminates only one of them. Second, the relationships between devices are also complicated; the VAV is connected to an Air Handler Unit (AHU) that is often in charge of many VAVs, and each device has subdevices such as fans and dampers. Third,

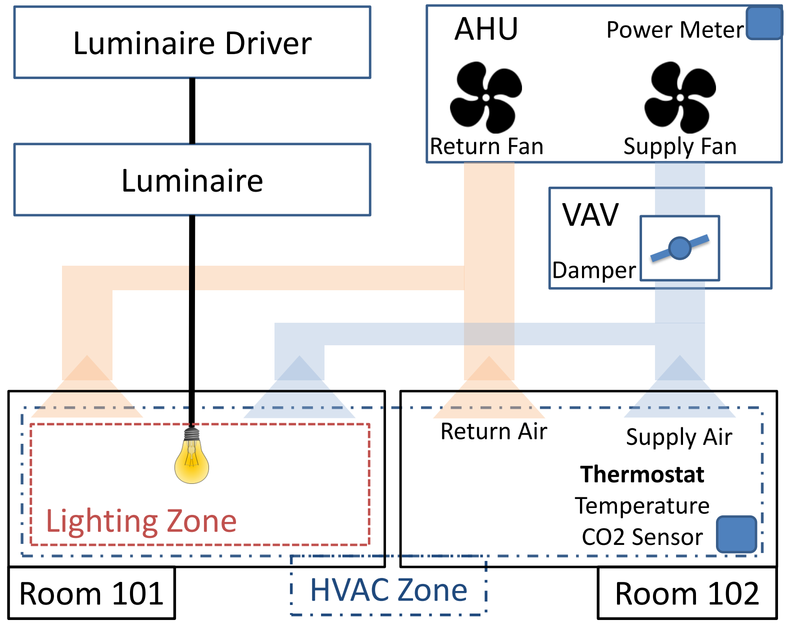


Figure 2.1: An Example Building Configuration

such devices refer to various data sources such as temperature sensors and temperature setpoints to accurately control the temperature. Notably, we call these data sources "points". Points are either a physical or virtual entity that continuously generates a value stream whose context is consistent across time. Each of the points generates data continuously over time, captured as timeseries.

Existing subsystems in buildings are operated through manual control, such as thermostats and switches, or with predefined logic often hard-coded in a local controller. In a typical HVAC controller, if the temperature of a room rises higher than a certain setpoint, an HVAC unit (e.g., VAV) would kick in. Better control logic might predict the status of the room and preemptively cool down the room for both occupants' thermal comfort and energy efficiency. The HVAC's schedule is also stationary as from 7 AM to 7 PM on weekdays regularly regardless of actual occupants' behaviors. Moreover, on weekends, it operates only on occupants' explicit activation through thermostats, which may not be accessible from a tenant's office if the shared thermostat is located inside the next office, as in Figure 2.1.

There are various ways to improve building operations, which we call smart building applications (apps). In general, apps pursue improving the energy efficiency of building operations and living environments for occupants. Table 6.1 shows the categories of building apps found in our most recent study [KHN+19] For instance, web thermostats, such as Genie [BKWA16], can provide more accessibility that could result in energy-saving and thermal comforts. Demand Response apps [Sia14] could save energy during peak energy demand through electricity price control for efficient energy generation and grid stability. Automatic customization of the environment [ZARP15] can provide better lighting conditions while meeting the various requirements from diverse tenants even in the same environment.

However, to deploy such apps on existing buildings, existing systems lack in an application programming interface (API). Control logic of building systems is commonly implemented inside a local controller that is often unconnected to any network or connected to a closed network such as BACnet to limit external access. Such closed network is commonly managed through Building Management Systems (BMSes), such as MetaSys from Johnson Controls, whose main purpose is for building operators to review and control when it is needed. They support visualization of equipment statuses, basic search functions for human managers to look up resources in their interests, and interfaces to change equipment configuration manually. However, because their primary end-users are human, BMSes do not have a programming interface through which apps can interact with actual equipment.

In contrast, emerging Building Operating Systems (BOSs) are designed for providing API as well as related administrative functions including data storage, access control, resource scheduling, and resource discovery. The end-users of a BOS are apps or programs that would analyze historical data from the underlying systems and control the systems based on certain logic, for which the core functionality is to interface with points, such as sensors and actuators, governing the underlying systems. For example, an app may read an occupancy sensor's log to infer an occupant's schedule or turn off an HVAC terminal unit when it notices the corresponding room is empty.

Here, the metadata plays a crucial role in identifying the right points for apps. Apps would rely on the resource discovery process a BOS provides. If the metadata is heterogeneous across different buildings, users or integrators would need to laboriously adapt the apps per building, which causes a high cost. This role of resource identification can be expanded to other administrative functions as we will discuss more in Section 6.3. In the next section, we will explore if the existing metadata in buildings suffice the requirements for such a standardized resource discovery mechanism.

2.2 Existing Building Metadata

Metadata is often called data about data and can embed any information that helps understand the meaning of the data. In buildings, there are various types of data such as timeseries data of a temperature sensor or the status of a light switch and structural information of spaces. The raw data are merely a set of certain values, such as, for sensors, numbers (72°F) and symbols (`on` or `off`) with timestamps, which do not convey what the values actually mean. Metadata could provide such information such as measurement types, location of the points, points' relationships with each other, etc.

However, metadata in existing BMSes, whose main users are building operators, is designed mainly for humans to interpret with domain knowledge about the target system. Table 2.1 presents anonymized real-world examples retrieved from existing buildings systems. There are several types of metadata depending on their sources. Vendor names are displayed and typically downloadable at vendor-provided user interfaces such as MetaSys and Desigo. Building managers read vendor names to understand their measurement types, locations, and related equipment to resolve system faults or tenant requests. Some communication protocols also provide metadata about points. For example, BACnet defines several fields to annotate points

Table 2.1: Metadata examples in buildings on different campuses.

Each row in the tables is raw metadata for a data point in the first two tables and corresponding labels in the last table. ID stands for point identifiers. Campus and building names are anonymized. Vendor Names are retrieved from vendor-given tools such as MetaSys while metadata related BACnet is only accessible through direct communication with BACnet devices.

ID	Campus-Building	Vendor Name
P1	A-1	ENG.CRAC-1.TEMPSETF
P2	A-2	SC-CRAC-1-MIG-008.TMP
P3	A-2	SC.3FLW-HALL.ZN-T
P4	B-1	RM123A Zone Temp 3

ID	BACnet Name	BACnet Description	BACnet Type
P1	NAE 05 N2 2 VND 162 TEMPSETF	Temp Setpoint	Analog Output
P2	NAE 14 N2 Trunk 1 MIG 008 Temp	Temperature	Analog Input
P3	NAE 13 N2 Trunk 2 VAV327 ZN T	Zone Tempreature	Analog Input
P4	N/A	N/A	N/A

ID	Point Label	Equipment Label	Location Label	Network Interface
P1	Temperature Setpoint	CRAC-1	N/A	VND-162, N-2-2, NAE-05
P2	Temperature Sensor	CRAC-1	N/A	MIG-008, NAE-14, Trunk-1
P3	Zone Temperature Sensor	VAV-327	Floor-3, W-Hall	Trunk-2, NAE-13, N-2
P4	Zone Temperature Sensor	N/A	Room-123A	N/A

such as Name, Description, and Object Type [ASH16]. Name and Description are still designed for human readability for managers to understand the points' context, and, because BACnet is a communication protocol, there are other information types than those in vendor names such as network interfaces (e.g., VND, N2, NAE, which are specific names of underlying network devices or sub-protocols.) There are symbolic fields too, such as Object Type. Object Type in BACnet has six different values as Binary Input, Binary Output, Analog Input, Analog Output, Multi-value Input, and Multi-value Output. BACnet devices use this information to check whether a certain signal from points is compliant to the designated port in the hardware. Likewise, most of the information in BACnet is designed for either human readability or protocol-specific compatibility check, but not for general apps information exchange.

Apps would need to access many resources upon their own application logic and the existing metadata is not the right tool for expressing apps' requirements. The existing metadata is inconsistent across different buildings and often even in the same building. Thus, to deploy an app across different buildings, any discovery mechanism should be able to cover all the different patterns of metadata. For example, a building environment analysis app needs to gather all the ambient temperature sensors residing in rooms per floor. The first step would be to identify the temperature-related points. With the given metadata as in Table 2.1, the app would need to look for anything containing “T”, “Temp”, “TEMP”, and “TMP”. Even such simple string comparison will fail if any of the keywords are a part of other keywords such as “Temporary”. Furthermore, there is no guarantee that all the information is there for representing apps requirements. After the above example app sorts out only temperature-related points, it still needs to verify location information whether the points are located in rooms of the first floor. While the inconsistent naming is still a problem, some points do not have location information at all. In that case, the app logic will be broken.

Traditionally, for deploying an app, an app integrator manually normalize the existing metadata into certain labels that the app can understand. Table 2.1 also shows normalized labels as point types (i.e., measurement type), associated equipment, locations, and network interfaces. For example, even though all the points in the table are related to temperature, they are encoded in various forms (e.g., TEMP, TMP, T, and Temp) as a different part of the strings (e.g., TEMPSETF, ZN-T, ZN T, and Zone Temp). Location is encoded in various ways while it is missing in some points. P1 and P2 do not have location information, P3 contains floor and room labels, and P4 only has room info. The names of equipment and network interface are not standardized as well and the meanings of names are specific to the vendors and the authors of the metadata. Metadata should have a standardized way of representing such labels that apps and BOSs can exploit in their operations.

The inconsistency in metadata stems from various factors. Existing metadata are mainly authored by humans without any rules or tools. First, there are no or few standard rules that authors can stick to. While a vendor’s internal convention defines several keywords or the types of information to put in metadata, those are not complete and the authors need to be flexible across different buildings' configuration and requirements. Of course, these conventions are only internal but not agreed upon by all the vendors. Second, the authors are human and error-prone. Even with some internal conventions, humans might make typos or omit necessary information as well as misunderstand some given rules. Lastly, buildings evolve across time. While there could be some fixes, renovation, or new device installation, not all of them are done by the same metadata author, and thus the styles of metadata change.

The fundamental reason behind all of these is the lack of motivation from system integrators or vendors in the past. Traditionally, there was a limited demand for providing a programmable interface to these subsystems. Lately, however, many interest parties, including system vendors, energy retrofitting companies, standard bodies, and government agencies, shave recognized the value of integrating different systems and providing a standardized programmable interface for general apps.

2.3 Existing Building Metadata Schemata

People have studied and developed several metadata schemata for buildings. Bhattacharya et al. proposed a quantitative analysis framework for IFC and Haystack whose baseline is to completely model three different buildings and seven application categories [BPC15]. This work has become the foundation of Brick (Section 3.1.) Fierro et al. [FKA+19] showed a qualitative analysis of Haystack to show its tagging scheme is not extensible and leads to inconsistent usage in practice. Section 3.7 describes our update of Brick to Brick+. In this section, we summarize such studies [BPC15, FKA+19] as well as review other existing metadata schemata for buildings [Koh16].

2.3.1 Limitation of Industrial Foundation Classes

IFC was initiated in 1994 and became an ISO standard in 2018 to standardize Building Information Models (BIM) mainly for building architect and construction [ISO14]. Thus, its vocabulary lacks the necessary concepts for the building operation and apps. While IFC has updated its model since then, its vocabulary coverage was only 29% of unique tags occurring in the three base buildings' systems as of 2015 [BPC15]. More importantly, its relationship model is constrained to 3D modeling and cannot represent simple queries such as logical relationships between objects (e.g., whether two different sensors are located in the same room.) In addition, IFC's data model and serialization format are not designed for general programming language and thus IFC lacks in programming interfaces such as a programmable abstraction (e.g., object mappers) and query logic for general programming languages with which apps are developed.

2.3.2 Limitation of Project Haystack

Haystack is an open-source initiative formed in 2014 to simplify and standardize building modeling with tags. It defines a set of tags and data formats to annotate entities in buildings. In Haystack, there are three types of tags:

- *Value tags* define the roles of certain values associated with the corresponding entity. For example, `unit` is a value tag whose value represents a unit of the associated point.
- *Marker tags* specialize the type of the corresponding entity. For example, `temp` tag means that the associated entity is related to temperature.
- *Ref tags* are a type of value tags where the values are other entities. For example, `equipRef` means that the associated entity is related to the specific equipment.

Figure 2.2a shows an example entity in Haystack. The first entity is an instance of setpoint (`sp`) that configures the amount of airflow (`air`, `flow`) from a VAV, `VAV-101` (`equipRef`). Such

```json
{
    "id": "safsp-101",
    "air": "m",
    "flow": "m",
    "sp": "m",
    "point": "m",
    "equipRef": "VAV-101"
}
```

(a) A Haystack Representation of Supply Air Flow Setpoint

```json
{
    "id": "maxsafsp-101",
    "max": "m",
    "air": "m",
    "flow": "m",
    "sp": "m",
    "point": "m",
    "equipRef": "VAV-101"
}
```

(b) A Haystack Representation of Max Supply Air Flow Setpoint

Figure 2.2: Project Haystack Examples

simple modeling is flexible and usable with little knowledge about the building and the schemata itself and thus has attracted a significant user base.

However, such flexibility in Haystack trades the formality of models for the discoverability and consistency. Discoverability is whether a user can find the right entities from the model intuitively. While each tag represents partial information about an entity, their roles are different for different kinds of entities, and Haystack ignores such differences but just represents the undeterministic association of tags with entities. This results in the disparity between an actual model and users' intuitive understanding of the model because the partial information does not represent the entire entity whereas users would rely on the partial information to discover the necessary entities from the model. As an example, compare the first entity with the second one in Figure 2.2b with an additional tag, `max`. The latter one configures the maximum possible value that the former one could have, other than how the airflow rate should be actually set. It contains all the marker tags that the setpoint has because it is associated with all the concepts within the former one. With such overlapping concepts, to change the current airflow rate of a certain zone, a user would look up entities with all the tags, `air`, `flow`, and `sp`, which happen to be a subset of `max air flow sp` as well. Thus, the tag matching query would select both kinds of points even though one of them is a configuration parameter for the other one. To precisely select the right entity among the query results, the user should additionally exclude entities with `max` tag.

Still, `max` is not the only tag to filter out and it might fail with other similar concepts such as `min`. While tags are intuitive for annotating partial information of an entity, it should be coherently used so that it would match users' intuitive understanding of models.

Due to similar reasons, users' modeling with Haystack is inconsistent across different users and buildings. Consistency is crucial for building interoperability; buildings should be modeled with the same understanding of vocabulary in a metadata schema for all of them to be operated in the same manner. However, Haystack's tagging scheme causes inconsistent modeling in several ways. First, how to use tags is encoded in documentation instead of a verifiable machine-readable model. For example, `vfd` can be a point or equipment in Haystack. While VFDs are equipment, they are often represented by a singled point in BMSes and Haystack allows users to decide how it is used in their models without guaranteeing any interoperability in the future. Second, the meaning of a tag varies within different compositions of tags and the documentation is the single source. "chilled" has different meanings when used standalone and in `chilledWaterCool` as the former one for points and the latter one for equipment, and there is no formal way to verify which is correct is actual usage. Third, there is no way to prevent human errors with incomplete or conflict modeling. Partial mapping occurs in Haystack instances in practice as people naïvely map existing BMS tags into Haystack tags while BMS tags are often implicitly meaning diverse information. For instance, `sensor` is often omitted in BMSes to represent a temperature sensor just with `temp`. While a set of hard-coded rules could prevent it (e.g., if `temp` occurs, one of `sensor`, `sp`, or `cmd` should occur too,) there is no standard way of representing such rules and evaluating them in Haystack.

These problems can be resolved by a formal representation of the knowledge such as fine-grained tagging scheme or a class-based scheme. For example, instead of nondeterministically associating `sp` to both of the example entities in Figure 2.2, we could associate the tag with a predicate such as `instantiates sp` and `regulates sp` where the former one indicates the entity is an instance of setpoint and the latter one means it is an instance that regulates a different setpoint. However, fine-grained tagging complicates the modeling process and usage because all the tags should be associated with different predicates or contexts. On the other hand, the class scheme can consistently model concepts without too much complication of associating different predicates for tags. In a class scheme, users strictly instantiate a class to model an entity, which might be related to other entities. A class can implicate the actual meaning of tags within a hierarchy. For example, `air flow sp` can be a specialization or a subclass of `sp` whereas `max air flow sp` can be a totally different class. In this case, `max air flow sp` class implicitly defines that it is not a type of `sp` but a class whose instances would regulate other setpoints. While users can consistently instantiate classes with a common understanding of the knowledge instead of arbitrarily associating tags with entities, some information is implicit in the class definitions. Our proposed metadata schema, Brick, combines the best of both worlds by providing a usable class hierarchy (Section 3.1.4) and structuring the hierarchy with fine-grained semantic tagging scheme (Section 3.7).

2.4 Definitions

We will reuse the following terminologies throughout the dissertation.

- **Entity**: An entity is a conceptualized existence, which can be either physical or virtual. It is also called resources in this dissertation. Applications refer to entities' metadata and data for their operations.
- **Class**: A class is a category of entities sharing the same properties.
- **Point**: A point is either a physical or virtual entity that continuously generates a value stream whose context is consistent across time.
- **Building model**: A building model is a machine consumable representation of the building, often as a collection of metadata for entities existing in the building.

- **Application/App**: An application utilizes information about the building and various entities inside to improve building operation. Apps refer to a building model to its operation.
- **Building Operating System (BOS)**: Building Operating Systems is a collection of software components to provide API for apps and users to access entities in buildings.
- **Building Occupant**: Occupants are the end users of buildings, who live, work or spend time inside buildings.
- **Building Operators/Managers**: They are responsible for operating buildings' equipment to meet the buildings' requirements such as energy usage and environment control.

Chapter 3

Extensible and Verifiable Metadata Schema for Buildings

In this chapter, we present our standard building metadata schema, *Brick*. As described in the previous chapter, existing work has observed that existing metadata schemata are not suitable for representing all the information that applications (apps) would need for interacting with building systems. From the beginning, Brick is designed for the large-scale app deployment, analyzed over six different buildings and apps from 85 papers. It is the first metadata schema quantitatively analyzed over actual buildings. Brick is extensible, by its design, to new terminologies while preserving the semantics even with new concepts, and also can be integrated with other models using standard semantic web technologies. As Brick is a formal model for representing resources in buildings other than just a set of predefined vocabulary, Brick can better maintain consistent usage across different buildings.

We first introduced Brick in 2016, and Bhattacharya among the authors also presented Brick in his dissertation [Bha16]. In this chapter, we present the original Brick as well as how it has evolved to Brick+ for bugs fixes, new concepts, integration with building systems, and consolidation of its design, which show Brick's usability in the real world. We have deployed

Brick with a building at the University of California, San Diego (UCSD), where actual apps are running. We will discuss app deployment workflows in the real world more at Chapter 6.

Brick is now being adopted into the industry (i.e., companies and other standard bodies) and publicly maintained via its community at `https://brickschema.org`.

3.1 Schema Design

3.1.1 Design Principles

Brick's design focuses on data points, their metadata found in real building deployments and requirements defined by end-use apps for operations and management. 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 introduce 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 3.4). We pick eight representative application categories from the list of smart building applications compiled by Bhattacharya et al. [BPC15], 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 3.4 contains our findings for the six buildings.We use existing standards in ontology development as Turtle [tur] for data formatting and SPARQL [spa] 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 98% of the vocabulary found in six buildings in different countries. (Section 3.4)
- **Vocabulary Extensibility**: The structure of Tags/TagSets allows easy extensions of TagSets

for newly discovered domains and devices while allowing inferences of the unknown TagSets with Tags. (Section 3.1.3)

- **Usability**: Brick represents an entity as a whole instead of fragmentarily annotating it. It promotes consistent usages by different actors. Furthermore, its hierarchical TagSets structure allows user queries more generally applicable across different systems. (Section 3.1.3 and 3.1.4)
- **Expressiveness**: Brick standardizes canonical and usable relationships between entities, which can be easily extended with further specifications. SPARQL facilitates all the possible combinations of the relationships required by queries of the eight application categories in the literature. (Section 3.1.5 and 3.3)
- **Schema Interoperability**: Using RDF enables straightforward integration of Brick with other ontologies for different domains or aspects. (Section 3.5)
- **Formality**: The formal rules of Brick TagSet formation promote the integrity of the schema design and enforce consistent usage of Brick over different users. (Section 3.7)

3.1.2 Development of Brick

An ontology is a structured collection of knowledge in a specific domain. Because an ontology is meant to be commonly used in the target domain (and sometimes together with other related ontologies for different domains), it should be able to represent all the necessary concepts and meet all the users' requirements in the target domain. Thus, the development of an ontology could be a highly iterative procedure where many experts and institutions altogether would gather the information to model in an ontology.

While there are several ontology engineering methodologies [IMM+13], we generally follow the Unified Process for ONtology building Lite (UPON Lite) methodology [DNM16] because of its agility. With UPON Lite's workflow, developers without the knowledge about ontology engineering can systematically contribute to the ontology design. UPON Lite consists of six steps;

1. *Collect lexicon* for the target domain.

2. *Embed meanings* into lexicon such as defining textual descriptions and finding synonyms.

3. *Organize a taxonomy* from the lexicon with generalization/specialization hierarchy.

4. *Define relationships* across different terms.

5. *Identify meronyms*[1].

6. *Formally encode* the collected knowledge into an ontology.

Throughout the entire procedure, ontology engineers can gradually refine knowledge. A preliminary step implicit in UPON Lite is to scope the target domain. However, it is hard to determine the scope of buildings in general because different buildings would have different resources and apps' requirements vary as well. We set our initial goal as to completely represent the data points discovered in our six testbed buildings and the apps that we identified from 85 papers.

We follow similar steps for establishing a common ontology for our target domain. The researchers from each campus are responsible for extracting and organizing all the information from their campus. To collectively gather the information through the ontology engineering procedure, we design a Comma-Separated Values (CSV) template[2]. CSV formats are a table-like structure where each column has a specific category of information and each row represents a set of information about a thing. In our case, each row represents a single class of entities defined by a developer and has the information specified in the following columns:

- *Lexicon*: Classes or types of entities observed in buildings. We reuse the types provided by BMSs of the buildings. The collection of this corresponds to the first step of UPON Lite.

[1] In our case, meronyms are identified in the process of defining relationships.

[2] V1.0.3 CSV file: `https://github.com/BrickSchema/Brick/raw/v1.0.3-archive/src/TagSets.csv`, last accessed: 2019-10-26

- *Definition*: Definitions of terminologies. While we put our best effort into defining rare concepts, we partially adopt definitions from ASHRAE Terminology[3] as ASHRAE is one of the most authoritative institutions for standardizing building systems.
- *Synonym*: Terminologies with the same meaning.
- *Abbreviation*: Terminologies with the same meaning. BMSs tend to use abbreviations to represent commonly used concepts, such as Chilled Water Pump (CWP) and Variable Air Volume unit (VAV), within a short description. This is a type of synonym.
- *Parents*: All the concepts that generalize the row. This is similar to parent classes in the object-oriented programming model.
- *Author*: The author of the row for provenance.

The methodology of UPON Lite is supposedly a linear process where each step contains the sufficient information required in the next step. In practice, because people might have different understandings of the same concepts, each step is internally iterative to agree on all the specifications. Furthermore, human errors could occur or the extracted information from each stage might not be complete. Thus, we evaluate the coverage of our model after all the steps and then augment missing or incorrect concepts in each step until our model completely represents everything in our target buildings.

Note that, in the CSV template, there is no process of defining relationships between concepts. There are many relationships in building systems such as physical proximity, network connections, and hardware configurations. Modeling all of such relationships is an exhaustive process and may not be necessary. We, instead, focus on our goal, enabling a programmable interface for *apps*. We analyze 85 app papers to identify the necessary relationships by apps (Section 3.1.5). We identify the necessary relationships required by apps based on our analysis of

[3]ASHRAE Terminology, `https://xp20.ashrae.org/terminology/`, last visited: 2019-10-25.

85 related papers as discussed in-depth in Section 3.1.5. After the curation of knowledge, we use a Python script[4] to encode the definition, taxonomy, constraints, and relationships into RDF/OWL format as detailed in Section 3.2.

3.1.3 Tags and Tagsets

We borrow the concept of *tags* from Project Haystack [hay] (Section 2.3.2) 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 combination of tags 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 [CKAK15b], an entity such as `Zone_Temperature_Sensor` from Figure 2.1 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 a 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

[4]Brick generation script for 1.0.3: `https://github.com/BrickSchema/Brick/blob/v1.0.3-archive/src/BuildBrick.py` last accessed: 12/08/2019.

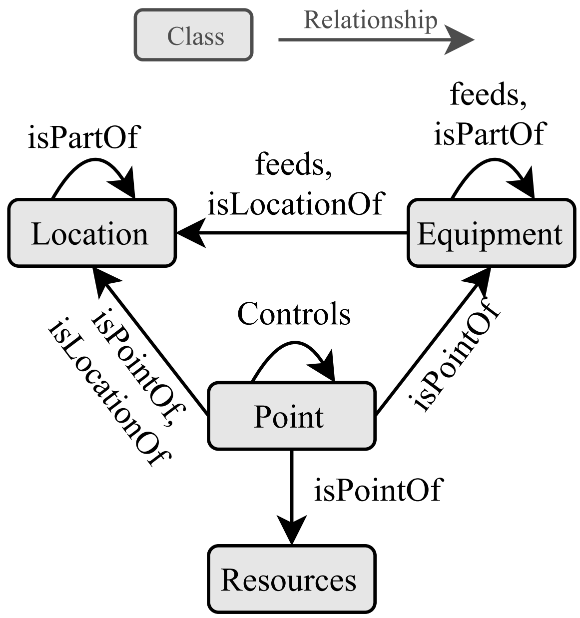


Figure 3.1: Information concepts in Brick and their relationship to a data point.

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 the use of complex tags such as the `chilledWaterCool` and `hotWaterReheat` tags in Haystack. The vocabulary of Brick is defined by its list of tagsets.

3.1.4 Class Hierarchy

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. [BPC15] 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* (Figure 3.1). We define each category as follows:

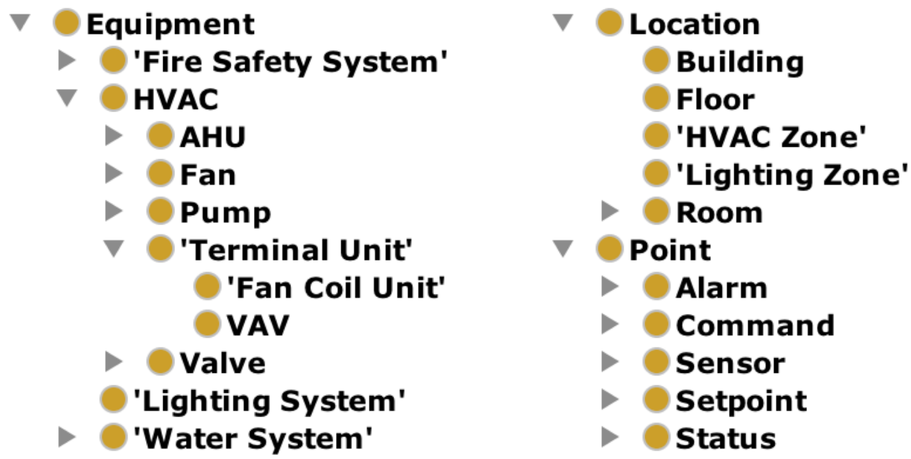


Figure 3.2: A subset of the Brick class hierarchy

- 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 that 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 resources 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.

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. Figure 3.2 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 inheritances when appropriate. For example, a desk lamp can be a subclass of both the lighting system and office appliance classes.

3.1.5 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 (Figure 2.1) 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 [BPC15].

We construct essential relationships by pulling a representative example from each of the eight common application dimensions identified by Bhattacharya et al. [BPC15] as summarized in Table 3.2. The categories of quintessential relationships we extract from the applications are:

- **Taxonomy:** what class or classes of things define an entity
- **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
- **Composition:** what entity an entity is a part of, or what entity is a part of it
- **Point Connections:** what points affect the behavior of other points

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

Table 3.1: 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.

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 3.1. The specific entities and relationships each application category requires are listed in Table 3.2. 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

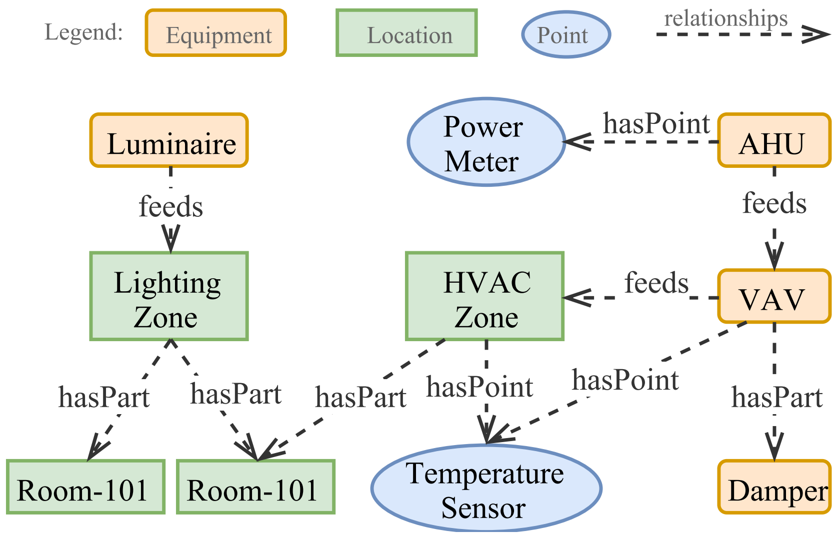


Figure 3.3: Brick classes and relationships for a subset of the example building in Figure 2.1.

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. Figure 3.3 shows the relationships for a subset of the example building in Figure 2.1.

Brick uses the possible subjects/objects defined in Endpoints column of Table 3.1 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.

3.2 RDF and SPARQL

3.2.1 Representing Knowledge in RDF

Brick adheres to the RDF (Resource Description Framework) data model [RDFa], 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 [RDFa], RDFS [RDFb], and OWL [OWL] 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 Figure 3.4 represent the connection of the VAV to the temperature sensor using the `hasPoint` relationship from the example building in Figure 3.3. 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

```
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
```

Figure 3.4: RDF triples instantiating a VAV and a Temperature Sensor and declaring that the VAV measures temperature via that sensor.

```
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 }
```

Figure 3.5: A simple SPARQL query for retrieving all rooms connected to a given Air Handling Unit (AHU).

associated with the given temperature sensor.

3.2.2 Querying Knowledge with SPARQL

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

Figure 3.5, 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 3.1).

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 Figure 3.5 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.

3.3 Applications

Table 3.2: App Requirements for entities and relationships.

	Entities	Occupancy Modeling [JKK+13]	Energy Apportionment [JSSJ11]	Web Displays [BKWA16]	Model Predictive Control [SGMS12]	Participatory Feedback [KDHL+11]	Fault Detection and Diagnosis [SBCH06]	NILM [MH09]	Demand Response [WBD+11]
Points	Temp Sensor	X					X		
	CO2 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 `hasLoc` Loc.	X	X			X		X	
	Equip `hasLoc` 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			

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 [EKBD17]. 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. [BPC15]. Research has shown that each of these applications can have a significant impact on improving building energy efficiency [MH09, JKK+13, JSSJ11, BKWA16, SGMS12, KDHL+11, SBCH06, WBD+11]. There have been hundreds of papers published that discuss how to design each of these applications to maximize their energy savings, and we have seen several industry startups that have started to deploy them in real buildings [com, bid, sky, ene, kgs]. 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.

3.3.1 Application Coverage

We implemented these applications as a set of SPARQL queries identifying the relationships in Table 3.2. 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 3.3 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 with each other. The Participatory Feedback application is designed for lighting controls. It shows relatively low coverage of buildings because many of the BMSes in our buildings do not expose points related to lighting systems. However, the relationships used in the application are 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 using UNION operations that allow the temperature sensor to be associated with either a zone or a VAV.

3.3.2 Example Application: Genie

We show an example application from the perspective of Brick. The Genie [BKWA16] 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

Table 3.3: 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 [JKK+13]	261	245	366	821	265	232
Energy Apportionment [JSSJ11]	-	302	-	397	4	-
Web Displays [BKWA16]	699	81	65	835	106	605
MPC [SGMS12]	482	69	428	324	110	482
Participatory Feedback [KDHL+11]	-	253	-	386	-	-
FDD [SBCH06]	229	29	229	728	-	136
NILM [MH09]	6	82	-	1348	-	-
Demand Response [WBD+11]	2300	24	2490	608	4	152

```
SELECT ?airflow_sensor ?room ?vav
WHERE {
  ?airflow_sensor rdf:type/rdfs:subClassOf*
    brick:Supply_Air_Flow_Sensor .
  ?vav rdf:type brick:VAV .
  ?room rdf:type brick:Room .
  ?zone rdf:type brick:HVAC_Zone .
  ?vav brick:feeds+ ?zone .
  ?room brick:isPartOf ?zone .
  ?airflow_sensor brick:isPointOf ?vav .
}
```

Figure 3.6: The Genie query for airflow sensors and rooms for VAVs. The query returns all relevant triples for Genie to bootstrap itself to a new 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 Figure 3.6. 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 and `Supply_Air_Flow_Sensor`s. The "Web Displays" row of Table 3.3 contains the results of running Genie over the six buildings.

3.4 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 Chapter 4 and Chapter 5 at scale.

Table 3.4: Case Study Buildings Information

Building Name	Location	Year	Size (ft^2)	# Points Points	% Tagsets Mapped	# Relationships Mapped
Gates Hillman Center (GHC)	Carnegie Mellon Univ., Pittsburgh, PA	2009	217,000	8,292	99%	35,693
Rice Hall	Univ. of Virginia, Charlottesville, VA	2011	100,000	1,300	98.5%	2,158
Engineering Building Unit 3B (EBU3B)	UC San Diego, San Diego, CA	2004	150,000	4,594	96%	8,383
Green Tech House (GTH)	Vejle, Denmark	2014	38,000	956	98.8%	19,086
IBM Research Living Lab	Dublin, Ireland	2011	15,000	2,154	99%	14,074
Soda Hall	UC Berkeley, Berkeley, CA	1994	110,565	1,586	98.7%	1,939

3.4.1 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 8,000 BMS data points for

HVAC. CMU contracts with Automated Logic[5] 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 equipmentnot 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.

3.4.2 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[6] 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

[5] Automated Logic, `http://www.automatedlogic.com/`

[6] Trane, `https://www.trane.com/`

features several different 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 of some uncommon equipment such as a heat recovery system 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 with 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.

3.4.3 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[7], 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 the 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 hot and cold water for domestic medium temperature water systems and controlling air temperature in the HVAC. The corresponding sensors that measure the hot and

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

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 describes 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.

3.4.4 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 mechanisms. 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 airflow 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 airflow 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.

3.4.5 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[8], but to protect basic building functionality only a subset of the BMS points is 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 screenshots, 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 either on walls or in a 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.

[8]Tridium, `https://www.tridium.com/`

3.4.6 IBM Research Living Lab

The IBM Research building in Dublin was retrofitted as a modern 15,000 m^2 office in 2011 from an old factory. The building serves as a 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 2,154 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 1,000 points for 161 FCUsas 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 the importance of the relationship modeling and the capability of Brick.

3.5 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 on how to systematically integrate different models in buildings. In 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

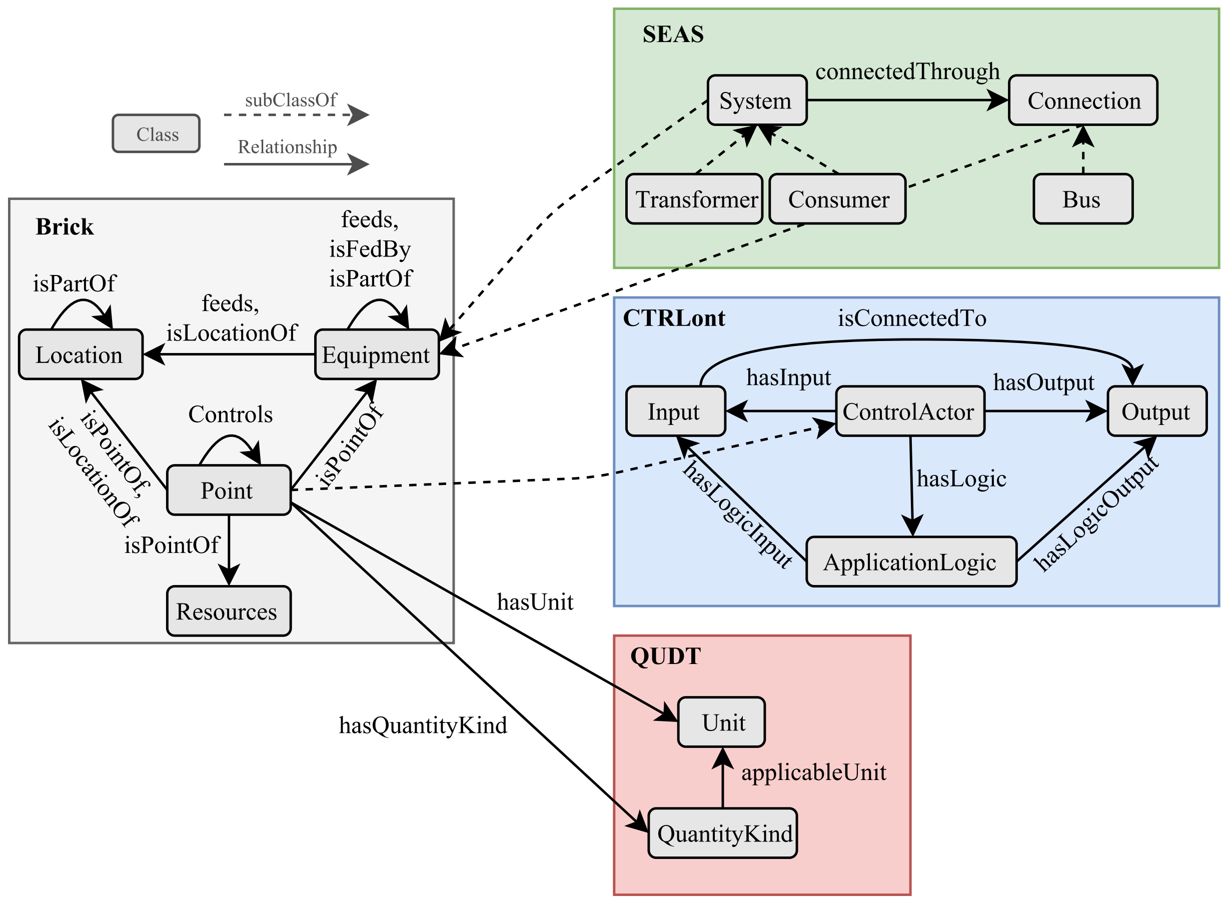


Figure 3.7: Integration of Brick with other ontologies.

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.

different domains in Figure 3.7, showcasing Brick's flexibility and extensibility even for the scope outside Brick's original design.

3.5.1 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 [qud]. We link the vocabularies under `QuantityKind` and `Unit` in the QUDT ontology using the relationships `hasQuantityKind` and `hasUnit` respectively as shown in Figure 3.7. `QuantityKind` represents ”any observable property that can be measured and quantified numerically” such as temperature and energy [qud]. 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” such as Celsius and Joule [qud]. Building domain’s unit vocabularies 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 while handling units [SSK16]. We present two of the use cases in Figure 3.8 for building applications. The first one (Figure 3.8a) 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, Celsius in the example. The second one (Figure 3.8b) is to validate if the given unit for a point is correct. The validation query matches the discovered unit to units applicable to the corresponding quantity kind. Thus, the QUDT integration enables the automated functionalities with unit composition, conversion, and validation.

3.5.2 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 [SPS17] is an ontology modeling control logic that can fully describe control actors and logic, and modularize the logic to ensure reusability and easy extension. Especially, it is an ontology that can be simply integrated with Brick. Its core concept is sense-process-actuate to model control processes where `ControlActor` processes `Inputs` based on `ApplicationLogic` and produces `Outputs` that may actuate devices. Points in Brick receives inputs based on `controls` relationships from other points to produce its own output, which is an abstraction of `ControlActor` in CTRLont. `point` being a subclass of `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` without a conflict, the integration can inherit functionalities proposed by CTRLont such as the automated rule-based verification of control logic in BMS [SPS17].

3.5.3 Electrical Power System (SEAS)

Smart Energy Aware Systems (SEAS) Knowledge Model [Lef17] is an ontology aligning energy systems to existing ontologies such as SOSA (Sensor, Observation, Sample, and Actuator) ontology [JHC+19b] and SAREF (Smart Appliances REFerence) ontology [DdHR15] and has several subdomains including electric power systems. Brick has already extended vocabularies to electrical power systems, but further extensions of the vocabulary set and sustainable cooperation with other models would be practically required for system interoperability and portability. In SEAS, `Systems` are connected with each other through `Connections` 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.

However, the connection in SEAS ontology is undirected while `feeds` is directional. In SEAS, the connection through a bus represents physical connection as a mere wire through which electricity can flow in any direction. The `connectedThrough` relationship tells that a system and a connection are connected but not the direction of electrical current. On the other hand, in Brick, the connection is directional as the electricity flows through the wire based on the role of the system such as from a transformer to a consumer through a bus. Thus, merging the relationships raises a logical conflict in RDF though a user may keep both of the relationships for the system. In the integration, two different models can share the common vocabularies for equipment, providing both of the functionalities.

3.6 Limitations of Brick

3.6.1 Inference with Tags

As we detailed in Section 3.1.3, the original Brick (Brick 1.0 hereafter) has predefined TagSets consisting of several Tags, of which each has an indivisible meaning. The TagSets form a hierarchy to represent concepts in a lattice to represent specialization and generalization relationships across each other.

With these concepts, we can logically infer[9] several types of information from existing building models. First, we can expand an is-a relationship of an instance with a TagSet to its superclasses. For example, an instance of `Temperature_Sensor` should be an instance of `Sensor` which is a superclass of `Temperature_Sensor`. Even though SPARQL's query engine can represent transitive relationships through inference with `rdfs:subClasOf*` relation (Section 3.2), expanding is-a relationships throughout the hierarchy can make app queries simpler without inferring the transitive relationships and even help to adapt Brick to other query models that do not support graph traversal.

Moreover, we can interpret TagSets as Tags in instance models. The definition of a TagSet is a predefined collection of Tags and the TagSet's name is the concatenation of the corresponding Tags in order. Thus, one can infer that an instance of a TagSet, such as `Temperature_Sensor`, is associated with its decomposition into Tags, such as `Temperature` and `Sensor`. Given mapping

[9] Refer to Fierro et al. [FKA+19] for the implementation of inference.

TagSets into Tags, modeling with TagSets is a superset of tagging schemes, such as Project Haystack (Haystack hereafter) [hay], and Brick can be compatible with any existing systems with a tagging scheme.

However, inferring TagSets from models with Tags remains unformalized in Brick's design. While Brick's TagSets guide users to consistently annotate an entity through instantiating a predefined TagSet instead of associating the entity with arbitrary Tags, we have observed that some users feel more comfortable with using Tags instead of TagSets. It is because existing users are already familiar with a tagging scheme; more importantly, the vocabulary size of TagSets is larger than that of Tags. As of Brick 1.0.3, there are 2027 TagSets and 313 Tags. TagSets are combinations of Tags, and, though we predefine canonical TagSets only, the number of TagSets is potentially exponential to the number of Tags. Even though users can utilize a schema search engine[10] without memorizing everything, some users might be overwhelmed by the vocabulary size of TagSets. Thus, it is a desired feature to infer TagSets from instances associated with Tags and formalize their relationships.

When we designed the first version of Brick, we overlooked the possibility of primarily using Tags, and, thus, we did not formally verified the relationships between Tags and TagSets. The simple rule to infer a TagSet from the associated Tags is as follows: if an instance is associated with all the Tags that a TagSet has, it is an instance of the TagSet as well. Such inference is the reverse direction of inferring Tags from a TagSet. For example, if an instance is associated with `Temperature` and `Sensor`, it is an instance of `Temperature_Sensor`.

[10]Brick schema search tool: `https://brickschema.org/ontology/1.0.3`

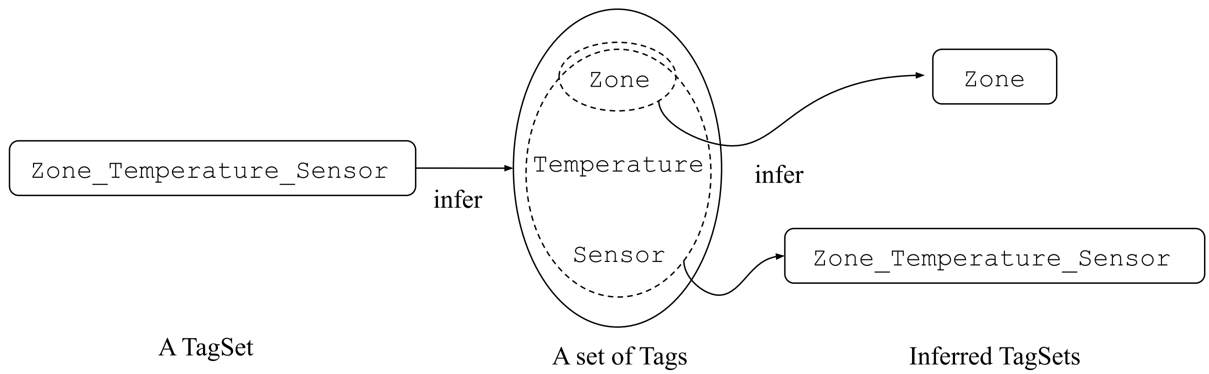


Figure 3.9: Infer Tags from TagSets and Vice Versa in Brick 1.0

However, Tags in a TagSet may imply more information than just their association with the exact TagSet, because Tags' roles vary across different TagSets. For example, `Zone` in `Zone_Temperature_Sensor` means that its instances measure the temperature of a zone more than just it is somehow related to `Zone`. As visualized in Figure 3.9, this confuses the TagSet inference logic because there is no difference between `Zone`'s meanings for `Zone_Temperature_Sensor` and `Zone` TagSets whereas `Zone` is a measurement target in the former one and a class itself in the latter one. Furthermore, Tags might also imply its TagSets' superclasses in the hierarchy. For instance, `Sensor` in `Zone_Temperature_Sensor` means that it is a type of `Sensor` as well as a type of `Point` as defined in the hierarchy. It is often misleading to deducing such information directly from Tags.

3.6.2 Coherent Extensibility

Brick 1.0 provides a hierarchy across all the TagSets represented by subclass (i.e., specialization and generalization) relationships (Section 3.1.4). A modeler can extend the schema by adding a new TagSet that subclasses another TagSet in the hierarchy. The newly added TagSet will preserve all the semantics of its superclass while introducing new semantics. For instance, `Temperature_Sensor` is a subclass of `Sensor` and the users of the model would expect it to preserve the semantics of `Sensor`. Even though the hierarchy can be as much deep as needed by modelers, apps can query resources with the right level of abstractions with the hierarchy. For example, a query looking for any instances of `Sensor` can identify `Temperature_Sensor` as well.

However, such hierarchy is designed solely based on human knowledge of the original Brick developers and there were no formal rules for a TagSet being a subclass of another TagSet. For example, `Zone` is a subclass of `Location`, and there is no semantic link between them other than the subclass relationship defined by a domain expert. Because it is done by a human, the design entails human errors. The errors include inversed subclasses, incorrect subclasses, and incorrect class definitions. While a hierarchy allows extensibility through subclasses, the hierarchy is not fully verified and its extension relies on manual engineering. There should be a more formal way to coherently extend the hierarchy by explicitly encoding what kind of information has been added between a class and its parent class.

3.7 Design of Brick+

We introduce Brick+ to solve the two major problems in Brick 1.0: ambiguity in mapping from Tags to TagSets and systematic extensibility of the schema. They all can be resolved by formal relationships between Tags and TagSets, which enables tag-based inference and schema verification. Our schema verification method has found several bugs in Brick 1.0 including incorrect class designs and subclass relationships.

3.7.1 Formal Structure of Brick+

First, we separate Tags from class names. As we discussed in the previous section, Tags in a TagSet often does not thoroughly represent what the TagSet means. Figure 3.10a shows that each TagSet is associated with Tags within its name. In Brick+, we instead relate a class with all the Tags necessary for the class, independent of the class name. Thus, TagSet, our previous term for classes, becomes obsolete because a name is not just a lexical concatenation of the corresponding Tags. We instead call classes “Class”. In this way, Brick+ becomes more flexible in associating necessary Tags to a Class without complicating names too much. Figure 3.10b shows an example class and its associated Tags. Unlike in Brick 1.0, we can associate a Class named `Zone` with both `Zone` and `Location` to completely describe what the class is about even though `Location` is not a part of the name. To exactly represent `Zone` class within its name, it should be `Zone_Location`, combining all the related Tags, to disambiguate itself from `Zone` in other zonal Classes such as `Zone_Air_Temperature_Sensor`. We can now associate an arbitrary number of necessary Tags outside a Class's name to completely describe the semantics.

It also allows representing Classes with Tags with sufficient and necessary conditions, which enables complete inference between Tags and Classes without ambiguity. Previously, it is nondeterministic to infer `Zone` Class from `Zone` Tag because it could be used for representing other zonal Classes (Figure 3.9). In Brick+, we make unambiguous all the sets of Tags associated with Classes. Thus, `Zone` Class should be associated with both `Zone` *and* `Location`, as in Figure 3.10b, which is differentiated from other zonal Classes such as `Zone_Air_Temperature_Sensor`.

First, we separate Tags from class names. As we discussed in the previous section, Tags in a TagSet often does not thoroughly represent what the TagSet means. Figure 3.10a shows that each TagSet is associated with Tags within its name. In Brick+, we instead relate a class with all the Tags necessary for the class, independent of the class name. Thus, TagSet, our previous term for classes, becomes obsolete because a name is not just a lexical concatenation of the corresponding Tags. We instead call classes “Class”. In this way, Brick+ becomes more flexible in associating semantically necessary Tags to a Class without complicating names too much. Figure 3.10b shows an example class and its associated Tags. Unlike in Brick 1.0, we can associate a Class, `Zone`, with both `Zone` and `Location` to completely describe the semantics of the class even though `Location` is not a part of the name. To exactly represent `Zone` class within

[11] Though it is `Zone_Temperature_Sensor` in Brick 1.0, we instead present `Zone_Air_Temperature_Sensor` for the fair comparison with Brick+.

its name, it should be `Zone_Location`, combining all the related Tags, to disambiguate itself from `Zone` in other zonal Classes such as `Zone_Air_Temperature_Sensor`. We can now associate an arbitrary number of necessary Tags outside a Class's name to completely describe the semantics.

It also allows representing Classes with Tags with sufficient and necessary conditions, which enables unequivocal inference between Tags and Classes. Previously, it is nondeterministic to infer `Zone` Class from `Zone` Tag because it could be used for representing other zonal Classes (Figure 3.9). In Brick+, we map all the sets of Tags associated with Classes one-to-one. Thus, `Zone` Class should be associated with both `Zone` *and* `Location`, as in Figure 3.10b, which is differentiated from other zonal Classes such as `Zone_Air_Temperature_Sensor`.

As a first step toward semantic tagging, we explicitly specialize `Point` Classes with `Substance`[12] and `Quantity` Classes for describing what a `Point` Class `measures`. This information dimension augments the class hierarchy through grouping `Point` Classes with the substance that points are associated with (e.g., `Air` and `Water`) as well as quantities of the substance (e.g., `Temperature` and `Air_Flow`). Each of the substance types can be specialized further within the category as exemplified above (e.g., `Return_Air`), forming a hierarchy, and we can reuse different substance types in different Classes. `Quantity` has its own hierarchy too (e.g., `Reactive_Power` is a subclass of `Power`.)

We have formalized the relationships between Tags and Classes through disassociating Tags from Class names and strictly disambiguating associated Tags per Class. Furthermore, we semantically augment tagging especially for `Point` through `measures` relationships with `Substance` and `Quantity`. Semantic tagging helps to consistently extend the schema with preserving the interpretability of new Classes. Brick+ is now released as Brick 1.1.

[12]Formerly `Resource` in Brick 1.0.

Algorithm 1 A Schema Verification Algorithm with Subsumption Rules
C: All Classes in Brick+, c_i: ith Class, T_i: Tags in c_i

procedure CHECKSUBSUMPTION(C)
    for $\forall c_i \in C$ **do**
        $Super_{i-hierarchy} = \{c_j | c_i \text{ subClassOf } c_j\}$
        $Super_{i-inferred} = \{c_j | T_j \subset T_i\}$
        assert $Super_{i-hierarchy} \equiv Super_{i-inferred}$
    end for
end procedure

Schema Verification

While there are various aspects to be verified in a schema, with the newly introduced formal structure, we can verify whether the semantic of each class complies with the class hierarchy. The semantic meaning of a Class is represented by its associated Tags, and the Class hierarchy should follow the subsumption rules of Tags. In other words, a Class's Tags should be a subset of its subclass's Tags. Algorithm 1 describes the procedure. For each of the Classes in Brick, we infer its superclasses in two ways: the first one is based on subclass hierarchy defined with `rdfs:subClassOf` and the other one is based on the subsumption rules with Tags, and those two sets of superclasses should be identical. The algorithm has identified that 177 Classes were not correctly defined during the Brick+ development process[13] and has led to the modification of some major Classes.

3.7.2 Changes in Major Classes

Through formally associating Tags with Classes, we have fixed several semantic errors in Brick 1.0.

[13]The related discussion: `https://github.com/BrickSchema/Brick/issues/66`

Introduction of Parameter

Control parameters in a system govern the behavior of associated setpoints. For example, Max_Air_Flow_Setpoint in Brick 1.0 regulates the maximum possible value that its associated setpoint can have. While we have put it in the Setpoint subtree in Brick 1.0, the Tag Setpoint in the TagSet does not represent its type but rather the associated point that it regulates. Such a pattern is widely observed in configuration variables in building systems, including PID parameters and minimum/maximum limits. We thus introduce Parameter as a new subtype of Point and migrate all the related classes under the subtree.

Removing Equipment-based Points

Point TagSets account for the largest number of TagSets in Brick 1.0 because we originally designed TagSets to be as standalone as possible to represent point names with a single TagSet in building systems. However, we have found that such practice inflated the number of TagSets, of which the majority is not actively used. One of the redundant information type in TagSets is equipment-related Tags because equipment is commonly instantiated and explicitly related to points. For example, VAV_Zone_Temperature_Sensor represents Zone_Temperature_Sensor instances working for a VAV while they can be associated with actual instances of VAV. Thus, in Brick+, we remove all the equipment-related Tags in Point Classes and instead encourage users to explicitly relate Equipment instances with Point instances. This change is an example of the tradeoff between instance-level and class-level modeling methods, which we will discuss more in Section 3.8.

Augmenting Location-associated Points

In Brick+, we explicitly associate Substance and Quantity for what Point classes can measure. However, conventionally Air is implicit in Location Tags when used for Point Classes such as Zone_Temperature_Sensor. To consistently model Point Classes, we explicitly add Air Tag to the related Classes, and thus Zone_Temperature_Sensor has become Zone_-Air_Temperature_Sensor. Though this might be against the conventional usage of Tags by field engineers, the role of Brick+ is to give a semantically robust modeling framework. Still, for usability, we could implement an inference engine that can infer Classes from conventional sets of Tags which may not directly comply with Brick+'s tagging rules.

3.8 Discussion: Instance- vs Class-level Modeling

One of the biggest challenges in maintaining Brick is to curate the vocabulary. Brick 1.1 (Brick+ in the previous section) has 888 Classes and 299 Tags. Having a too large vocabulary set would harm Brick's usability and possibly consistency as well. In contract, a small vocabulary set might be insufficient to express all the essential concepts or too generic to deliver users' intention precisely.

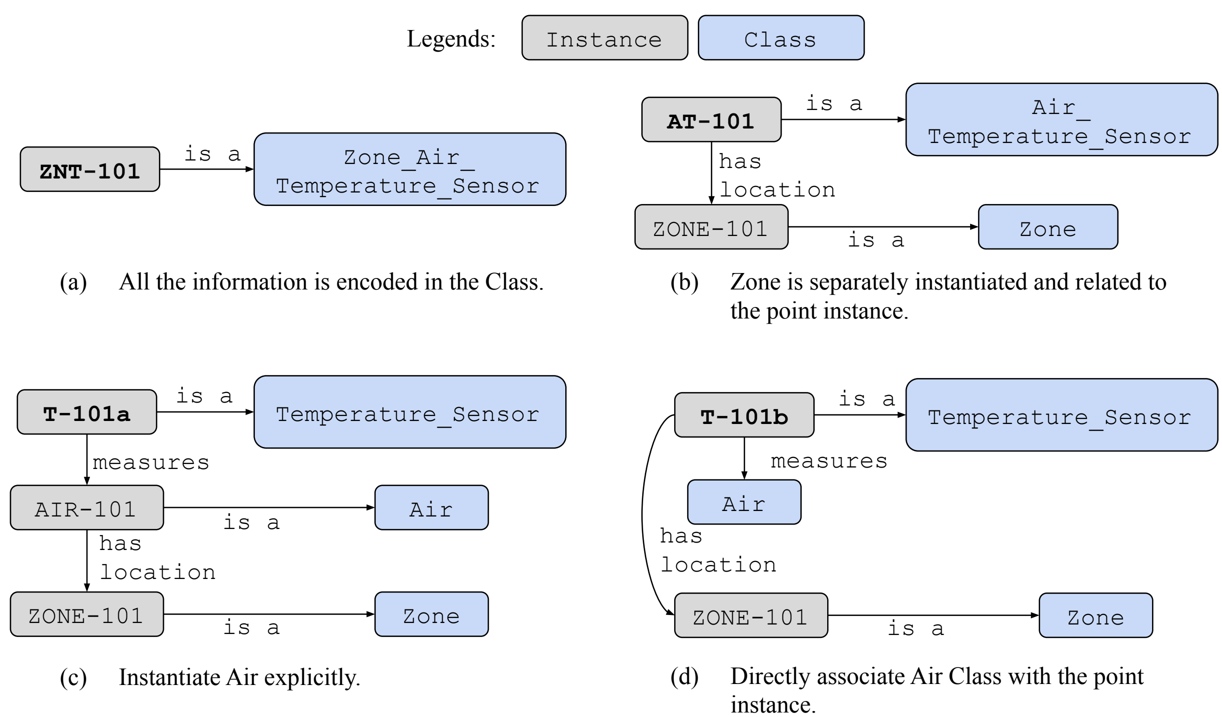


Figure 3.11: Information concepts in Brick and their relationship to a data point.

However, choosing the right amount of vocabulary is an open problem in any schema design because there are various ways to convey the same information. Consider the following examples that represent points with the same properties in different Classes.

(a) An instance of `Zone_Air_Temperature_Sensor`.

(b) An instance of `Air_Temperature_Sensor` located in a zone.

(c) An instance of `Temperature_Sensor` measuring air of a zone.

While different Classes represent different levels of abstraction, their instances could be related to other entities, and the relationships could complement the missing information in different Classes.

Compare the first two examples visualized in Figure 3.11. `ZNT-101` is an instance of `Zone_Air_Temperature_Sensor` while `AT-101` is an instance of `Air_Temperature_Sensor` located in `ZONE-101` which is an instance of `Zone`. Those two instances share the same properties as measuring the air temperature of a particular zone. While these two models are comparable, they necessitate different efforts at the modeling stage. On the one hand, the former one does not require an actual instance of `Zone` whereas the latter one needs it. On the other hand, the former one requires to create a more specialized Class, `Zone_Air_Temperature_Sensor`, whereas `Air_Temperature_Sensor` can be reused for other types of air such as discharge air or return air.

It is the trade-off between instance-level (ABox, assertions on individuals) modeling and class-level (TBox, assertions on concepts) modeling [DGL96]. If we model all the information as instances and their relationships, the metadata schema could be more straightforward, which in turn would make actual usage more consistent (the principles of Occam's razor.) However, it also means that a modeler should instantiate all the detailed information, which requires more effort and knowledge about the system (Figure 3.11(b)). Instead, a metadata schema can embed more properties into Classes (Figure 3.11(a)) even though it would increase the number of Classes in the schema.

The trade-off exists within instances as well because some information is inherently tricky to instantiate. Compare the last two examples in Figure 3.11. Figure 3.11(c) explicitly instantiates `Air` to be associated with the point instance. However, most of the `Substance` Classes are intangible, and, thus, it is hard for users to comprehend them as instances, as well as instantiating all the substance could explode the number of instances in a model. In Brick, we allow associating Classes with instances through punning[14] as `T-101b` measures `Air` Class directly but not its instance in Figure 3.11(d). It allows the flexibility for modelers to relate entities explicitly with intangible Classes without instantiating them.

Brick's primary goal is to provide interoperability across any systems instantiated by different users. In a schema design, we should consider how to guide users to properly model buildings with Brick in a correct manner and have to balance the trade-off between the instance-level and class-level modeling. Having too many Classes would decrease the consistency and confuse the users, whereas it is cumbersome for modelers to instantiate every single concept in the model. That is why we remove all `Point` Classes with equipment-related Tags in Brick+ (Section 3.7.2). As we commonly instantiate `Equipment`, equipment-related Tags in `Point` Classes become redundant. On the other hand, it is rare to refer to instances of `Substance` Classes in practical query patterns as the substance is intangible and abstract. Thus, we define `Point` Classes specialized for `Substance` Tags, with which users are free from organizing too many intangible instances.

In principle, both instance-level and class-level modelings are compatible through an inference logic because instances can be abstracted to Tags which can be associated with a specific Class. For example, we can infer an instance of `Air_Temperature_Sensor` measuring a zone is also an instance of `Zone_Air_Temperature_Sensor`. Thus, we should have an inference engine

[14] Refer to Fierro et al. [FKA+19] for more details.

that can normalize all the information in a model so that all the models are compatible with each other. Such tools would give flexibility to users for choosing a preferred modeling principle with preserving interoperability.

Still, we need to verify if every instance's relationship can be uniquely encoded into a Tag for the correct inference, and, furthermore, an overwhelmingly large schema would degrade usability for practitioners. While Brick's framework provides a complete vocabulary and expressivity, its usability should be periodically reviewed by field engineers so that we can maintain the Brick's design to be harmonized with the practical usage.

3.9 Future Work

While Brick is the canonical metadata model for buildings, we would need to resolve several issues for building apps and systems to be truly interoperable across different buildings. The first is to align the human understanding of the models to the actual standard. Even though Brick is canonical, users of Brick might not have a complete understanding of how they should use Brick. There are several cases where an actual understanding of Brick might diverge across different users, such as different levels of abstractions (Section 3.8) and partial relationships between instances. We should augment Brick by providing a possible structure between Classes and define inference rules for guiding users to properly use Brick without dictating them with a document or inferring canonical models from imperfect models in practice.

The second is the value-level heterogeneity. While Brick provides the framework to model entities and their relationships, points' actual timeseries values have not been standardized. For example, `Occupancy_Command` in Johnson Controls' Systems usually have three values as 1, 2, and 3, which represent "unoccupied", "standby", and "occupied" respectively. The semantics of values might be varying across different systems, and we should be able to provide a reference of how users should interpret the meaning of points' values with Brick. We could provide enumerations for common points and map the enumerations into actual values in the target system's model.

The third is the functional heterogeneity. Different buildings have functionally different system configurations, and we need to be able to interface over those different systems if possible. For example, different types of terminal units, such as VAVs and FCUs, might have different on/off model while a remote controller app would like to provide the same interface over them. Another example is occupancy information. We can infer occupancy from various data sources such as temperature changes or PIR sensors. Even though different buildings might have different raw data representing occupancy, we would like to provide the same binary value for the occupancy. Our future work here should consider inferring semantics of values aligned to apps' requirements.

3.10 Summary

With Brick, we have shown that a structured metadata can be the foundation of large-scale building app deployment. Brick can represent resources in buildings with their types and relationships with each other, and apps can refer to Brick for actually finding their necessary resources. We have developed Brick through a collaborative procedure with a target goal of representing six different buildings from different campuses for seven app categories. Our reference buildings validate the completeness of Brick and application queries show its expressivity for app requirements.

Brick's design principle is based on consistent usability and coherent extensibility. We have seen that modeling with classes is more consistently usable than tagging scheme, while tag-based models can be easily mapped to our classes-based model. RDF, as a modeling framework in Brick, has enabled the integration of Brick with other domain-specific ontologies with RDF's flexible and polymorphic query model (SPARQL). Furthermore, there are many existing modeling frameworks for different concepts in RDF (e.g., RDF, RDF, OWL, SSN, etc.). Brick can utilize them to verify whether our modeling practice is aligned with other people's understanding of general concepts. In general, Brick is *complete* as capable of representing all the information buildings and apps need, and *maintainable* as consistently usable and coherently extensible.

Since Brick was introduced in 2016, it has much attracted both the industry and academia. Companies, including HVAC vendors and software companies, have adopted Brick into their systems. Many papers have followed up Brick to extend it for different aspects (real estate management [HWKH19], services [HK18]), provide a testing framework [FPA+18, MJLM19], and use it for reasoning [KH18]. We thank all the collaborators and researchers who made the community around Brick. While we set the foundation, the Brick community will drive Brick's future, address problems for the society but not just several institutions, and discover solutions established by a group of users but not just a single expert.

3.11 Acknowledgment

Chapter 3, in part, is a reprint of the material as it appears in Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments, 2016 by authors Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjrgaard, Mani Srivastava, and Kamin Whitehouse. The dissertation author is one of the primary investigators of this paper.

Chapter 3, in part, is a reprint of the material as it appears in Applied Energy, 2018 by authors Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjrgaard, Mani Srivastava, and Kamin Whitehouse. The dissertation author is one of the primary investigators and the corresponding author of this paper. This is a journal extension of the above paper.

While this is a collaborative work that, in nature, involves active discussions among all of the authors, among Jason Koh's distinguished contributions are proposing and establishing a meta-framework with the Semantic Web technology, designing the knowledge collection workflow, organizing the collaboration both technically and non-technically, curating the collected knowledge into the canonical ontology, and taking in charge of the journal paper submission as the corresponding author.

Chapter 3, in part, also contains material as it appears in Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation, 2019 by authors Gabe Fierro, Jason Koh, Yuvraj Agarwal, Rajesh K. Gupta, David E. Culler. The dissertation author was the second author of this paper.

```
# 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) 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.

```
# 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) 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.

Figure 3.8: Example usages of QUDT with Brick.

```
@prefix brick: <https://brickschema.org/schema/1.0.3/Brick#> .
@prefix tag: <https://brickschema.org/schema/1.0.3/BrickTag#> .
@prefix bf: <https://brickschema.org/schema/1.0.3/BrickFrame#> .

brick:Zone_Air_Temperature_Sensor
    bf:usesTag tag:Zone;
    bf:usesTag tag:Air;
    bf:usesTag tag:Temperature;
    bf:usesTag tag:Sensor.

brick:Zone
    bf:usesTag tag:Zone.
```

(a) `Zone_Air_Temperature_Sensor`[11]and `Zone` in Brick 1.0. TagSets are simply associated with the Tags in their names. `bf:usesTag` is an annotation property that explains how the TagSet is formed.

```
@prefix brick: <https://brickschema.org/schema/1.1.0/Brick#> .
@prefix tag: <https://brickschema.org/schema/1.1.0/BrickTag#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

brick:Zone_Air_Temperature_Sensor
    owl:equivalentClass [
        owl:intersectionOf (
            [ owl:hasValue tag:Zone; owl:onProperty brick:hasTag; a owl:Restriction ]
            [ owl:hasValue tag:Air; owl:onProperty brick:hasTag; a owl:Restriction ]
            [ owl:hasValue tag:Temperature; owl:onProperty brick:hasTag; a owl:Restriction ]
            [ owl:hasValue tag:Sensor; owl:onProperty brick:hasTag; a owl:Restriction ]
            [ owl:hasValue tag:Point; owl:onProperty brick:hasTag; a owl:Restriction ]
        )
    ] .

brick:Zone
    owl:equivalentClass [
        owl:intersectionOf (
            [ owl:hasValue tag:Zone; owl:onProperty brick:hasTag; a owl:Restriction ]
            [ owl:hasValue tag:Location; owl:onProperty brick:hasTag; a owl:Restriction ]
        )
    ] .
```

(b) `Zone_Air_Temperature_Sensor` and `Zone` in Brick+. Classes are associated with Tags more than what the names have. `owl:hasValue` is a formal way of defining a rule to associate a Tag with a Class.

Figure 3.10: A comparison of relationships between Tags and Classes in Brick 1.0 and Brick+.

Chapter 4

Semi-Automatic Metadata Normalization Algorithms for Buildings

While standard metadata schemata, such as Brick, provide a basis for standardized semantic representation of resources for building applications (apps), most of the existing buildings do not adopt any standard metadata schema. Therefore, metadata normalization — converting existing metadata into a standard schema — is the first step toward providing general programmable interface for heterogeneous buildings. However, metadata normalization currently requires tremendous manual effort as well as significant domain expertise.

A building might have tens of thousands of points monitoring and controlling the hundreds of devices in tens of subsystems that are often from various vendors. As detailed in Section 2.2, building metadata is often in disparate formats across different buildings and could be inconsistent even within the same building. As a result, only experts on target buildings (e.g., building managers or contracted maintainers) can understand the actual meaning of the existing metadata. To annotate these points, an needs to extract the meanings of all the metadata in the building and maps them into Brick. Furthermore, experts who normalize existing metadata into Brick need to have a good understanding of Brick, from the syntax to the right vocabulary for the entities in their target buildings, to the necessary parts of Brick for their projects. Currently, mapping an existing building to Brick is a manual process, and doing so for each point is not only resource-demanding but also error-prone. In this chapter, we present two machine learning algorithms to facilitate instantiating metadata into the Brick format in actual buildings.

However, devising ML algorithms for metadata normalization face technical challenges in three ways. First, while typical ML algorithms presume that the patterns are identical or similar in training and target data, the patterns are significantly diverse across buildings, as we detailed in Section 2.2. As accumulating information is less feasible over different buildings with typical ML algorithms, a domain expert should laboriously label many training data across different buildings, and, even with that, it is not guaranteed that the collected training data would be useful for new target buildings. Therefore, any proposed algorithm should be either highly generalizable or have high sample efficiency to reduce the human effort.

Second, there are various types of metadata that apps might need to use. For Brick, there are hundreds of Classes and tens of relationships, and it demands a lot of effort for domain experts to extract all the information completely. At the same time, different apps would need different types of metadata such as a remote thermostat app that would primarily need temperature information per room. Moreover, different data sources, such as raw metadata and timeseries data, contain different types of information with different accuracy. Thus, it is challenging to devise a versatile algorithm that can cover all the aspects of metadata, different data sources, and various use cases.

Lastly, the result of metadata normalization is used in a mission-critical system, buildings. Though data analytics apps are mainly used by humans, some apps could be consumed by the system, such as automatic control and HVAC user interface, which can directly control building systems. Thus, the accuracy of metadata normalization is crucial to avoid disastrous misconfiguration such as deactivating HVAC of server rooms. Notably, high precision (more true positives and less false positives) is more important than high recall (more true positives

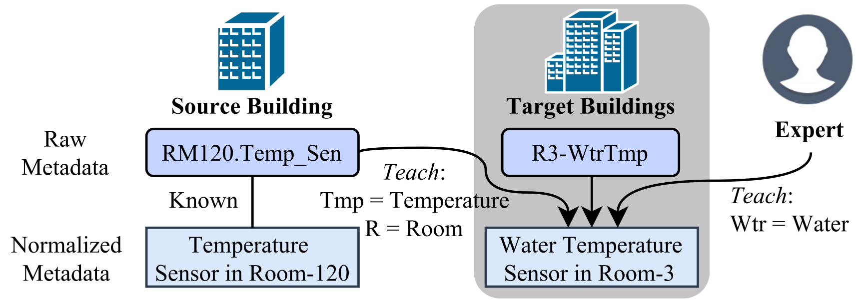


Figure 4.1: Fast metadata normalization of new buildings with a known building's information and an expert's knowledge.

possibly with large false positives) because false positives may lead to unnecessary control action on critical points [GG05].

In summary, a good metadata normalization algorithm should have the following features:

- it requires only a minimal amount of human effort (i.e., training data),
- handles various information types, and
- has high precision.

With those requirements, we present two algorithms; Scrabble and Quiver. Scrabble is the best-in-class for extracting all kinds of information present in existing metadata. It utilizes a sequence learning method, Conditional Random Fields, with intermediate representation to maximize knowledge transfer. Quiver is a control perturbation framework to efficiently learn points' relationships across each other, which are unknown in existing metadata, such as co-location and dependency.

4.1 Scrabble: Semi-Automated Metadata Normalization using Intermediate Representation

We present *Scrabble*, a framework to retrieve semantic metadata from unstructured raw metadata while reusing known information in existing buildings to reduce the amount of effort for domain experts to provide input labels. Fig. 4.1 provides an overview of the goal that known metadata can be mapped automatically (e.g. ‘Tmp’ is Temperature in target buildings) and ask domain experts to provide undiscovered labels (e.g. ‘Wtr’ is Water). Scrabble uses a two-stage, active learning approach exploiting a known taxonomy of labels and mapping information from existing buildings. At the first stage, we learn a Conditional Random Fields (CRF) model [LMP+01] to extract reusable intermediate representations (IR) from character sequences in an existing “source building” that has already been mapped to a known schema. In the second stage, we learn a multilabel classifier to map the IR to actual labels. We use Multi-Layer Perceptron (MLP) for the multilabel classification with its capability of handling the high dimension of the input words. For the IR, labels and taxonomy, we use a semantic ontology, Brick [BBF+16b, BBF+18b, FKA+19], that specifies a list of equipment, data points and relationships between them. We use active learning methods to ask domain experts’ input for the unmapped data points. Our model enables a smoother transfer of mappings from known buildings to a new target building while exploiting different types of information sources systematically.

We have implemented and evaluated Scrabble on metadata from five diverse buildings on 6,551 randomly chosen data points from a total of 21,802 points. We verified the ground truth of those points manually. With the IR and proper classifiers, Scrabble extracts entities from unstructured metadata with an improvement of 59%/162% higher Accuracy/Macro-averaged-F1 in a building than a baseline with 10 initial examples. Furthermore, Scrabble can achieve 99% Accuracy with 100-160 examples for buildings with thousands of points while the baselines cannot.

4.2 Scrabble Algorithm

Given unstructured metadata for data points in a target building B_T and ground truth semantic labels for points in a source building B_S, we normalize the metadata of the target

Algorithm 2 Scrabble algorithm overview
B_S: source building, B_T: target building. s_i: a sample

procedure SCRABBLE(B_S, B_T)
  while $U(s_i) < th \ \forall s_i \in B_T$ **do**
    Use CRF to learn Characters from $B_S \cup D$.
    Use MLP to map Tags $\rightarrow$ TagSets from B_S
    Infer Tags of points in B_t with the CRF model.
    Infer TagSets for the Tags of points in B_T with the MLP.
    Select samples with low confidence or utilization.
    Resolve manually on chosen samples.
  end while
end procedure

building B_T into the structured Brick schema. Our goal is to minimize the number of examples to extract correct and comprehensive metadata present in the given unstructured metadata.

Scrabble uses an active learning framework for normalizing metadata of sensors in multiple buildings by adopting a transferrable intermediate layer. We map a sentence from the target building to a set of Brick Tags using a CRF classifier that trains on samples from source buildings and examples from domain experts. We use Brick Tags as a reusable Intermediate Representation (IR) that is free of building specific metadata. IR is effective in reducing the number of learning samples if it is easier to learn the mapping from inputs to IR than labels directly [PPHM09, FEHF09, RPT15]. A multi-layer perceptron (MLP) then maps the Tags to corresponding TagSets. MLP is capable of handling high dimension data very well by learning important features inside hidden layers. We use confidence-based and Tags-utilization metrics to identify samples likely to be labeled incorrectly and resolve them by input from domain experts. Scrabble iterates through all target building samples until it can identify all building labels with high confidence. Algorithm 2 summarizes the entire process and Figure 4.2 describes the mapping of raw metadata to the semantic metadata with an example.

Our technique is based on two key observations. First, a word's meaning does not change even if its usage varies in different sensors. In Figure 4.2, RM is used to indicate *Room* as a sensor's location whereas it can be a part of RMT to represent *Room Temperature* in another

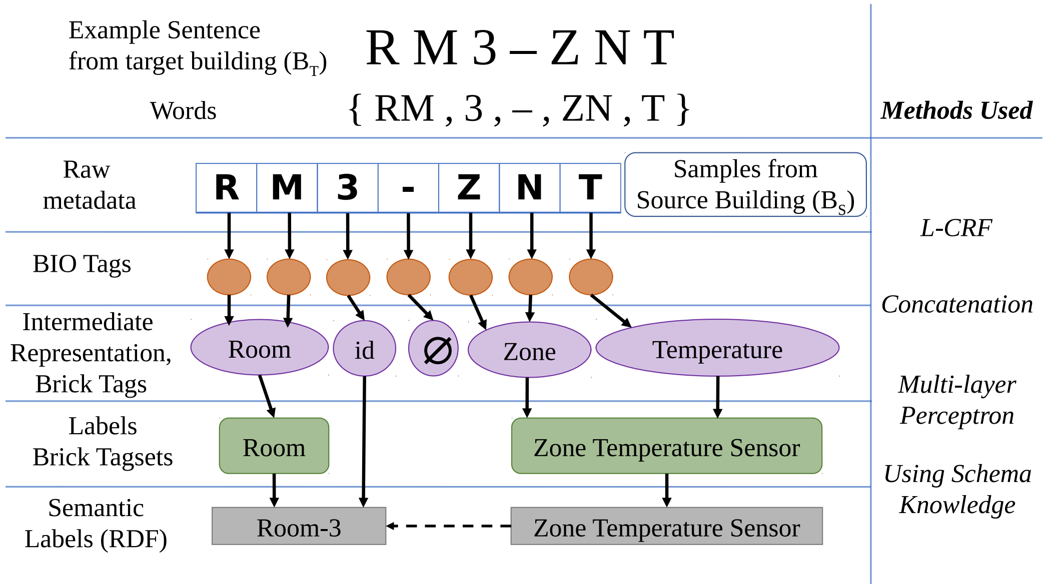


Figure 4.2: Data Mapping from Raw Metadata to Semantic Metadata

CRF maps tokens in raw metadata to the intermediate representation, Brick Tags. Multi-label classifier maps Tags to final labels, TagSets.

sensor. The meaning of RM as a *Room* remains identical in both cases. Thus, mapping from raw strings to Tags can be reused across different buildings though target labels may differ. Second, the relationships between Tags and TagSets can be learned and used across different buildings. For example, across all buildings, the Tags *Temperature* and *Sensor* form the Tagset *Temperature Sensor*. Thus, Tags can be reusable representations of metadata across buildings, which can, in turn, be easily mapped to Brick schema. We adopt this idea from zero-shot learning methods that use semantic codes [PPHM09] and attributes [RPT15]. We reuse the relationships discovered in a source building when available and domain experts can provide samples to learn newly observed relationships. We can rapidly retrieve structured metadata from a new building in this way.

4.2.1 Terminologies

Figure 4.2 gives an example of the terminologies. A building has various *points* that produce a data stream such as sensors. BMSes describe points with various types of *raw* *metadata*.Raw metadata associated with a point represents a set of labels, for which we use Brick *TagSets*. Raw metadata may contain *string metadata* like point names and *code metadata* like units. We call string metadata also a *sentence* composed of multiple *words*. A word may represent a set of Brick *Tags*, such as RM represents `Room`, but it does not have to be delimited by special characters. A word is decomposed to *characters*, of which each is mapped to a *BIO token* [RR09a] associated with its word's Brick Tags (details in Section 4.2.2.) In the learning process, a human *expert*, such as a building manager, provides *examples* for mapping a sentence to 1) Tags and 2) the TagSets that the point's raw metadata represent. Required examples are chosen by Scrabble to minimize the total amount of effort.

4.2.2 Raw Metadata to the Intermediate Representation

We define a word as a set of characters representing a concept. E.g., ZN, T, and RM in Figure 4.2. We map words to Brick Tags as an intermediate representation. Here, words are not necessarily separated by predefined delimiters. For example, ZN and T in ZNT are separable as *Zone* and *Temperature* because they can be reused in other sentences such as RMT for *Room Temperature*. In contrast, mapping ZNT directly to *Zone Temperature* loses the reusability of ZN and T in other contexts.

Every character in a word is labeled with a BIO (Begin, In, and Out) token [RR09a] to represent the location in the word. The word ZN corresponds to the Tag *Zone*, of which Z is located at the Beginning of ZN and N is Inside ZN. The BIO scheme captures this relative position of the character in the word and assigns Z to "B-Zone" and N to "I-Zone". The "O" BIO tag stands for Out, i.e., tokens that do not convey semantic meaning such as punctuations and definitive articles. In Figure 4.2, the punctuations ',' and '-' will be assigned to "O" token.

Scrabble learns a Conditional Random Fields (CRF) model [LMP+01] for mapping raw building metadata to Brick Tags with BIO tokens, e.g it learns that ZNT corresponds to "B-Zone", "I-Zone" and "B-Temperature" with examples from source building. CRF makes a Markov independence assumption, i.e., the tag of a character only depends on the neighboring characters. We use the following as input to our CRF model for character j: the jth character itself, (j-1)th character, (j-2)th character, (j+1)th character, is digit?, and is special character?

We additionally adopt code-based metadata such as BACnet units other than textual metadata. BACnet defines codes for certain entries such as units and object types [Bus97b]. For example, unit code 62 represents Celsius in BACnet, with which we surely know that the point is associated with *Temperature*. Similarly, a point with object type "analog input" in BACnet can be considered as a *Sensor*. These metadata are scattered in different entries other than in a single string so methods only parsing a string cannot integrate them systematically. Scrabble merges Brick Tags from different metadata at the second stage. Though we only use BACnet metadata, which is common in BMSes, this concept can be generalized into any other code-based metadata that can be interpreted as Brick Tags.

4.2.3 Mapping Intermediate Representation to Semantic Labels

We have thus far mapped the raw building metadata to Tags, which are an intermediate representation (IR) of Brick TagSets, our target semantic label. Our second stage maps Tags to TagSets. As the Brick schema can represent a general building, a classifier that labels TagSets from Tags can be shared across different buildings. Mapping from Tags to TagSets is challenging because not all Tags of a TagSet may be present when we perform the raw metadata to IR mapping, e.g., *Sensor* is often omitted in *Zone Temperature Sensor*. A single Tag can be attributed to different TagSets, and we need to identify which of these TagSets is the correct semantic mapping. For example, the Tag *Room* may correspond to the location TagSet *Room* or to a point TagSet like *Room Temperature Sensor*.

We use Bag of Words (BoW) with Term-Frequency Inverse-Document-Frequency (TF-IDF) [SM86] scheme to vectorize IR. BoW counts the occurrence of each Tag learned from a sentence and stores it as a feature vector. The length of the feature vector, i.e. its dimension, is the number of Tags in the Brick schema. TF-IDF skews the Tag counts to reduce the importance of common words such as 'the', 'to'. Learning standard classifiers on BoW such as a decision tree is an intuitive approach to learn the mapping between IR and TagSets. However, there are several difficulties to learn such classifiers. First, a set of Tags generated from a sentence represents multiple TagSets, which is a multi-label classification problem. In our model example, the raw metadata "RM3-ZNT" describes a *Zone Temperature Sensor* and a *Room* simultaneously (Figure 4.2). As the relationships among Tags for a sensor are unknown, what Tags in the set are used for what types of TagSets is also unknown. We need to identify multiple labels from one distribution. From the set of Tags, {Room, id, $\emptyset$, Zone, Temperature}, in Fig. 4.2, we need to identify two TagSets, {Room, Zone Temperature Sensor}. Second, samples are significantly biased across different buildings. There are points widely used such as *Zone Temperature Sensor* while some points specific to control special equipment occur only once. In the four buildings of our data set, 17% of TagSets account for 90% of TagSet occurrences on average and 34% of TagSets occur only once. Moreover, different buildings would have different types of points and metadata. A specific type of equipment may exist only in a particular building. Such new information cannot be pretrained from a source building. To address these challenges, we propose three approaches in Scrabble: sample augmentation, a domain agnostic classifier and iterative sample selections.

Sample Augmentation

Samples in a building are inherently biased toward the building's configuration and its original installer's writing style. We synthetically generate samples from the schema and the given building's samples to mitigate the biased samples in three ways.

(a) Brick Samples: The schema provides samples of mapping from Tags to TagSets as a TagSet is a composition of some Tags (e.g., *Zone*, *Temperature* and *Sensor* for *Zone Temperature Sensor*). We insert sets of Tags for each TagSet and a Tag in each set has a random frequency from 1 to a threshold, th_{tag}, where th_{tag} is the median frequency of Tags inferred in a sensor of a source building. The number of the generated schema samples, th_{ts}, is a hyper-parameter chosen empirically with a validation set. We limit each schema sample to have just one label. For instance, we generate a sample mapping tags of (*Temperature*, *Sensor*) to a TagSet, (*Temperature Sensor*). We avoid adding samples with multi-labels because the number of possible combinations of different TagSets increases exponentially to the number of total TagSets.

(b) Negative Samples: We can also add more possible combinations of TagSets from the given samples using logic similar to Brick Samples. Given a set of labels for a sample, we add its variations without a TagSet by removing Tags related to the TagSet (e.g., remove the Tag *Room*} to remove the TagSet *Room* in the label set of *Room*, *Zone Temperature Sensor*}.) It prevents overfitting to given samples while generating an acceptable number of samples in the order of the number of given samples.

Multi-layer Perceptron for Multi-label Classification

We use Multi-layer perceptron (MLP) to model the mapping from Tags to TagSets. MLP uses fully connected (FC) neural network layers to act as a universal function approximator [Pin99]. Here, we use sigmoid as the non-linearity function and use binary cross-entropy for the loss evaluation in the training phase. We use 2 FC layers and each of them is followed by a dropout [SHK+14] layer to generalize the model. This MLP stage receives vectorized BoWs for Brick Tags as inputs and infers a set of TagSets the vector represents as outputs. Our input data are considerably hard because the Tags are in high dimensions but sparse. We have empirically evaluated other multi-label classification models such as classifier chains, random forests, but MLP outperforms other methods because of its dimensionality reduction ability.

4.2.4 Sample Selection

We cluster sentences based on the tokens to identify similar sentences [BVNA15a, BHC+15a]. Each sentence is converted to a vector of the BoW model with tokens usually defined by contiguous alphabets or special characters. This tokenization need not be precise as it just needs to recognize similarities among sentences. The vectors are clustered based on hierarchical clustering and a small threshold determines output clusters. Balaji et al. [BVNA15a] empirically show that if the threshold is small enough, sentences in a cluster have the same label for sensor type, which is the most complex information in metadata. When selecting an example to learn, we pick one randomly from the most uncovered cluster. The coverage is defined as the rate of sensors given examples over the number of sensors in a cluster. This method is generally applicable for determining what examples an expert should provide to derive the best learning speed.

4.2.5 Active Learning with Domain Experts

We iteratively update the learned model with the target building's sample given by an expert. Target building's raw metadata may contain some points unobserved from the source buildings such as new systems and new conventions that should be taught by an expert. We have to carefully select samples for experts to answer so that we can achieve the fast learning ratio with minimal examples, which is called Active Learning (AL). In general AL, we evaluate unlabeled samples with the learned model and pick the most informative samples based on a certain query strategy. A domain expert provides labels for the samples, with which we update a new model. Then, we iterate the entire procedure.

As Scrabble consists of two-stages with one from characters to Tags with CRF and the other from Tags to TagSets with MLP, a query strategy should pick good examples covering both stages. We exploit two types of query strategies for different stages. The first one is to find the lowest confident inference at the CRF stage and the lowest entropy of inferences at the second stage. CRF is a sequence model that maps a sequence to labels. Among various ways of querying strategies [SC08], we select the least confident inferences (LC) for its simplicity, interpretability and less computational complexity. Settles and Craven [SC08] show that LC's performance is competitive compared to the others with a less computational cost. Confidence of a CRF inference is an inferred sequence's conditional probability but we normalize it with its length because the probability highly depends on the length of the sequence.

$$\Phi(\boldsymbol{s},\boldsymbol{b},\theta) = log(P_\theta(\boldsymbol{s},\boldsymbol{b}))/length(\boldsymbol{s}), \tag{4.1}$$

where $\boldsymbol{s}$ is a sentence with characters, $\boldsymbol{b}$ is BIO tags inferred from the model θ. P_θ is a CRF's loss function[1].

The second stage is multi-label classification and single label confidence cannot represent how the inference is confident in general. We instead exploit an assumption that identified Tags should be fully mapped to certain TagSets and query how many Tags are exploited in the current inferences. The assumption is based on the observation that the original installers put only meaningful information into the raw metadata as the space for the metadata is limited and it needs to provide useful information for maintenance and operations. For example, if given Tags are Temperature and Sensor, mapping it into Sensor is incomplete as Temperature is not used in the inference. Tags utilization is defined as follows:

$$U_{tags}(\Theta_i,T_i) = \frac{\sum_j usage_{tags}(\theta_{i,j},T_i)}{length(s_i) - \#(O\text{-}Tag)} \tag{4.2}$$

$$usage_{tags}(\theta_{i,j},T_i) = \begin{cases} 1, & \text{if } \exists t_{i,m} \mid \theta_{i,j} \in t_{i,m} \wedge t_{i,m} \in T_i \\ 0, & \text{otherwise,} \end{cases}$$

[1]For the exact CRF loss function, please refer to the original literature [LMP+01].

where Θ_i is a set of Tags identified for the point and T_i is a set of TagSets inferred from Θ_i. $\theta_{i,j}$ are Tags in Θ_i and $t_{i,m}$ are TagSets in T_i. O-Tags are ignored in the calculation as they have little meaning. We pick target building samples with low utilizations by the diverse random sample selection method. We dynamically choose outliers with utilizations less than an average of the entire utilizations subtracted by their standard deviation.

The entire process is summarized in Algorithm 2. Models for both Tags (intermediate representation) and TagSets (labels) are initially learned from a source building. We infer Tags and TagSets in the target building. Then, we calculate all utilizations of metadata at the target building and randomly choose N diverse samples. We set N as 0.5% of the target dataset size, but it is a hyper-parameter dependent on the heterogeneity of a target data set and the learning speed of a user's preference. An expert provides labels of the asked samples such as positions of words, their corresponding Tags and TagSets. All the models then are learned with the updated data set. These steps are iterated until all the utilizations of raw metadata in the target building is higher than a threshold.

4.3 Scrabble Evaluation

4.3.1 Experimental Setup

Datasets and buildings

We evaluate the framework with three buildings from campus A (A-1,2,3), one from campus B (B-1) and the other one from campus C (C-1). C-1 is one of the buildings used in ProgSyn [BHC+15a]. Due to the huge amount of human effort for labeling, we randomly choose 1000 examples per building for our evaluation in A-1,2,3 and B-1, and verify them manually for ground truth. However, we use the entire points in C-1 to exactly compare Scrabble's performance with ProgSyn. Table 4.1 summarizes the statistics of the buildings. Missing Tags indicate the

Table 4.1: Quantities of datasets for Scrabble evaluation.

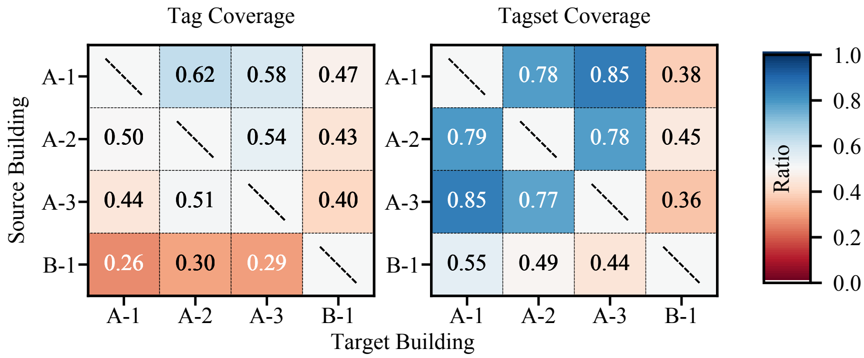
We choose 1000 points from each building randomly for the evaluation. The numbers of Tags and TagSets explain diversity of labels in buildings. The numbers are also averaged over data points. 1.1 Tags should be learned from example patterns, which accounts for 26% of a TagSet in average.

Building	Points (selected)	Total # of TagSets	Total # of Tags	Avg # of TagSets per point	Avg # of required Tags per point	Avg # of existing Tags per point	Avg # of missing Tags per point	Avg Length of metadata per point
A-1	4593 (1000)	129	122	6.30	8.67	7.83	1.09	67.7
A-2	1914 (1000)	120	96	7.12	9.23	8.62	0.83	67.9
A-3	4381 (1000)	137	111	6.90	9.85	9.00	0.94	74.9
B-1	8363 (1000)	90	73	4.37	6.36	6.30	1.35	59.4
C-1	2551 (2551)	68	65	3.03	5.17	3.69	2.71	21.7

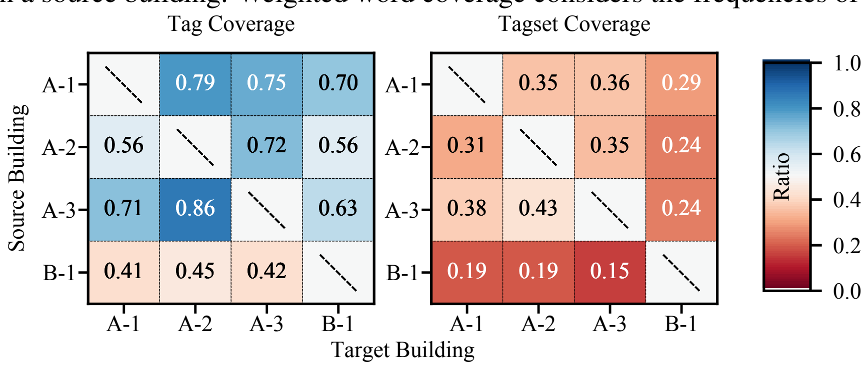
number of Tags shown by its corresponding TagSets. With fewer missing Tags, it would be more straightforward to map them to the target TagSets. A-1,2,3 and B-1 have more informative metadata with a longer metadata size and the contained TagSets. C-1 has relatively concise metadata with 21.7 characters metadata on average.

We first analyze the difference between the buildings for raw metadata and ground truth labels. Fig. 4.3a describes the rates of words in B_T occurring in B_S to show the possibility of reusing the model from source buildings. Words are defined as contiguous letters with a unit meaning that can be mapped to Brick Tags. Words coverages equally weight each word and the weighted word coverages include frequencies of words in each building. It shows that at least half of the words in B_T can be obtained given B_S, but the other half has to be learned with expert input. The gap between word and weighted word explains more frequent words are more common across different buildings. Thus, using B_S would help understand common words in B_T. These similarities are directly reflected in the learning rates of Scrabble in the later sections.

There are 119 TagSets and 101 Tags in a building on average among the 904 TagSets and 284 Tags defined in Brick. The similarities of Tags and Tagsets across buildings are shown in



(a) *Raw metadata similarity across buildings.* This shows how much a target building's raw metadata already exists in a source building. Weighted word coverage considers the frequencies of words.



(b) *Tags/TagSets coverage:* Coverage is the rate of common Tags and Tagsets between two buildings over those in the target building, showing Tags are more common than TagSets.

Figure 4.3: Similarity Comparison across Buildings' Datasets

Fig. 4.3b. Tags model would be more transferable between buildings as Tags are more shared between buildings than TagSets. We suspect Scrabble will perform better when the source and target buildings have similar metadata style. However, even when a target building metadata style is different, our hypothesis is that the learning rate would be better compared to learning from scratch.

Implementation

We implement Scrabble[2] in Python with PyCRFsuite [Oka07] for CRF, Keras [ker] for MLP and scikit-learn [PVG+11a] for the other machine learning algorithms. All the datasets are stored as files and the ground truth datasets are used as a domain expert iteratively providing

[2]Scrabble repository: `https://github.com/jbkoh/scrabble`

labels for active learning.

4.3.2 Evaluation Metric

We infer a set of TagSets from raw metadata. We use a label-based and an example-based metric [ZZ14].

- $Accuracy(h) = \frac{1}{p}\sum_{i=1}^{p}\frac{|Y_i \cap h(x_i)|}{|Y_i \cup h(x_i)|}$,
- $MacroF_1(h) = \frac{1}{q}\sum_{j=1}^{q} F_{1,j}$

where h is a model, p is the number of samples, q is the number of labels, x_i is an ith input vector and Y_i is ith label set. $h(x_i)$ produces x_i's inferred labels. $Accuracy$ calculates an average ratio between the number of correctly inferred labels and the sum of the correct labels, irrelevant labels and misclassified labels per sample. It captures how well the entire data set is classified, but the metric can be skewed by dominating classes. In contrast, Macro-averaged F_1 (Macro F_1) captures the normalized mean of measures F_1 scores across classes. It can exaggerate incorrect inferences of classes with rare samples though it provides a good estimation of the model's class coverage. In our dataset, a few classes such as the name of the *Building* occurs in all points, while a few classes, such as *Pump*, occur in very few. Such disparity in class samples causes the two metrics to differ significantly.

4.3.3 Baselines

For our baselines, we use two algorithms; the program synthesis (ProgSyn) [BHC+15a] and a multi-label version of Zodiac [BVNA15a]. ProgSyn is a promising solution for parsing strings but has several critical drawbacks as discussed in Section 4.9. In addition to the drawbacks, the algorithm cannot parse a string if there are repeated labels. In A-1,2,3, there can be repeating labels across metadata types, so ProgSyn simply cannot be executed over the datasets. Thus, ProgSyn is excluded in buildings other than C-1 in our evaluation.

Due to the limitation of the ProgSyn, we devise another baseline. Zodiac is an active learning framework for inferring point types from raw metadata while Scrabble extracts all possible labels - point type, equipment type, location. Zodiac vectorizes the raw metadata with BoW scheme and learns a multi-class classifier but it is limited to infer point types only. We modify Zodiac to infer multiple labels. We first use TF-IDF scheme instead of count vectorization to account for variation in word frequencies across different buildings. We use Classifier Chain of RF Classifiers for multilabel classification instead of a single RF classifier. Zodiac uses confidences of predictions to determine the most uncertain samples to ask experts, but there is no single confidence representing the inference on a sample in multi-label classification because each class has its own confidence. We instead use the entropy of class probabilities per sample [SC08]. The rationale is that if a sample is confidently inferred, its confidences of each class will be close to either zero or one. Entropy is high when a probability distribution is extremely spread, and we can know that a multi-label inference is not confident when an entropy of the inferences is low. An expert is asked to provide labels for 10 samples with the lowest entropy for each iteration. Note that mapping a BoW vector to a set of TagSets has a limitation of losing the relationship between actual words and the labels. For example, it cannot associate a *Room* label to the portion in the string representing the room including its identifier like room numbers. It is very often important to identify actual room number or device IDs for applications to properly take actions for the target object. Scrabble can traceback the mappings from characters to actual TagSets and identify actual names together.

Baseline results of five buildings are shown in Fig. 4.6. For C-1, ProgSyn's learning rate is steep in the early stage, but it slows down soon. Furthermore, it cannot reach 99% $Accuracy$ because it lacks the capability of accumulating more examples once it reaches the point where every metadata is qualified. For multi-label Zodiac, we observe that the words are similar in the buildings in the same campus A as their accuracies are initially higher than one being transferred from/to B-1. In the baselines, accuracies initially increase rapidly with the sample numbers

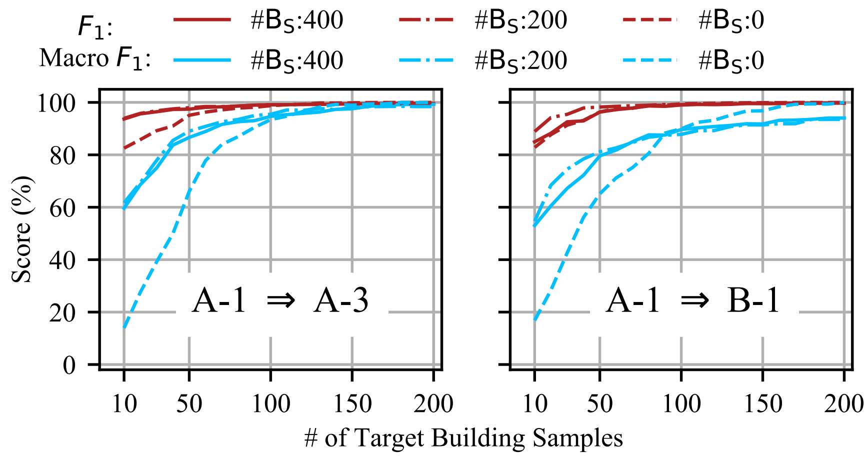


Figure 4.4: Learning rate of CRF mapping characters to Brick Tags.

It compares the learning rate with the different numbers of source building examples.

because samples are biased. A few examples can represent many others. Macro F_1 increases linearly which shows that the sample query mechanism is valid. In all cases, the baseline has no notable gain with source buildings initially with 10 examples. It shows that it is hard to exploit different buildings together with the naïve features. All the buildings can hardly achieve a high $Accuracy$ like 99% within 200 examples. For example, its best case is achieves 99% accuracy at 210 examples in A-1 and the worst case is 98% accuracy at 300 examples in A-3. We conclude that we would need a framework converging faster.

4.3.4 Experimental Results

We individually evaluate the first stage (characters ⇒ Tags), the second stage (Tags ⇒ TagSets), and the entire framework. Our experiments initially take 10 examples randomly chosen by the method in Section 4.2.4 and 10 examples per iteration based on the methods in Section 4.2.5 for 20 rounds. All results are averaged over four experiments.

CRF Evaluation

Scrabble maps characters to BIO token labels with CRF and concatenates them to form Brick Tags. Fig. 4.4 shows the results of learning the tags of a target building with source building data. Initially, 10 samples are randomly selected based on the mechanism in Section 4.2.4 and samples of the source building are uniformly randomly chosen. In each iteration, the models are learned and tested, and then the expert gives 10 different examples with the lowest confidences. The metrics in the figure are for Brick Tags, which are the features used in the next stage.

Fig. 4.4 shows that 200 samples from B_S improves F_1 from 82.6% to 93.7% and $MacroF_1$ from 14.0% to 61.7% in A-1⇒A-3, and from 82.8% to 88.9% and from 16.8% to 54.7% in B-1⇒A-1 initially than learning without B_S samples. A-3 easily benefits from A-1 as they are on the same campus with similar conventions as described in Fig. 4.3a. F_1 is high as 94% from the beginning with A-1. Even without A-1, F_1 is high as 82.0% because of several dominant words such as building names or location like Room occurring in most of the raw metadata. MacroF_1 also improves from 14.0% to 61.7 % and 60.0% by adding 200 and 400 examples from A-1, exploiting the similarity between A-1 and A-3. Even though the sample query mechanism is valid as the metrics monotonically increase, MacroF_1' converge around 100 samples. There is little difference between 400 samples and 200 samples for A-1 ⇒ A-3.

However, for buildings from different campuses (A-1 ⇒ B-1), we observe that MacroF_1 converges lower when there are source data than one without source data. Still, there is a large gain in MacroF_1 initially due to certain similarities of common words such as Room. The degradation of $MacroF_1$ in the later stage would be due to overfitting. Hyperparameter optimization needs to be investigated more. In general, adding samples from B_S improves learning rate in all cases initially, but may decrease $MacroF_1$ when there are many examples with different styles. It shows Scrabble's initial transferability but more generalizability with large data sets should be investigated more.

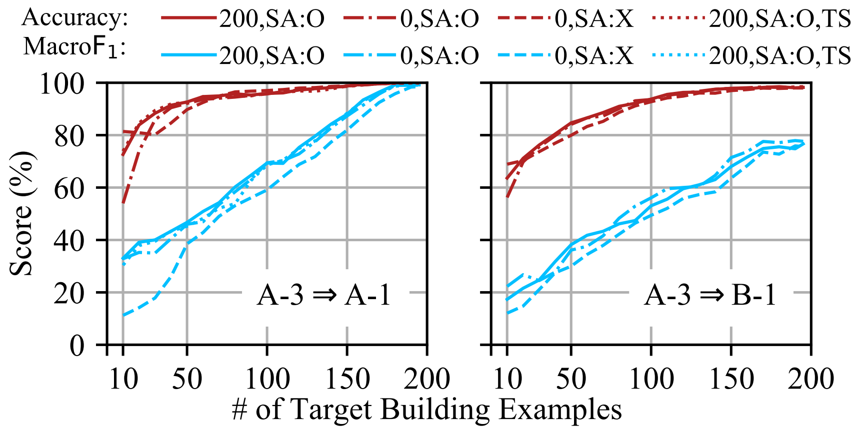


Figure 4.5: Learning Rate Comparison of Different Configurations for TagSet Classifier based on ground truth Tags. SA stands for sample augmentation, TS means learning with timeseries data, and the numbers are the number of samples from a source building.

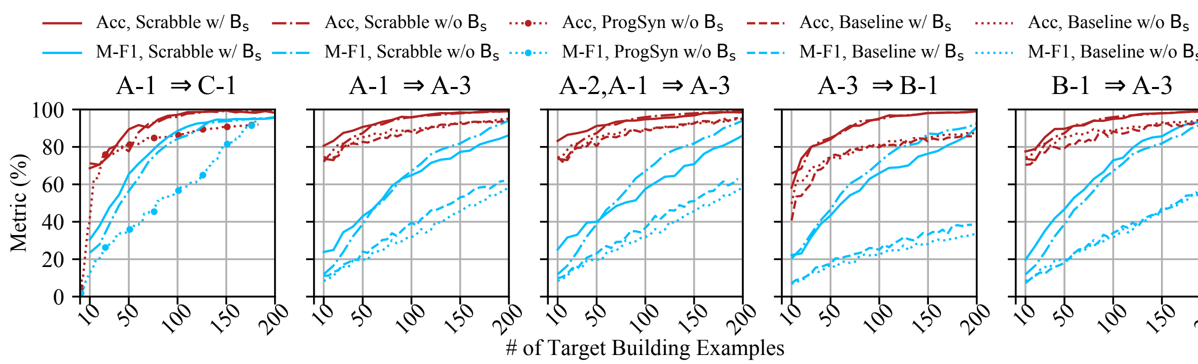


Figure 4.6: Learning Rate of Scrabble's Entire Process. On the left side of each arrow are source buildings and at the other side is a target building. Note that ProgSyn is only working with C-1 due to its limitations. ProgSyn also ceases before achieving high accuracy because of its deterministic algorithm. In general, Scrabble shows better performance exploiting existing buildings in the same campus with similar patterns. Even across different campuses, Scrabble does not degrade accuracies.

TagSet Classifier

To independently evaluate the performance of the TagSet Classifier, we presume the TagSet classifier receives correct Tags from the first stage. Fig. 4.5 shows how each component improves the learning rate of a target building based on *Accuracy* and Macro F_1. The naïve scenario without sample augmentation and samples from the source building achieves low $MacroF_1$ as 11.1% and moderate 81.2% *Accuracy* initially. Sample augmentation (SA) in the experiments includes all the two types discussed in Section 4.2.3. As it provides valid examples not included in the given small B_T's data set, $MacroF_1$ is also increased from 11.1% to 36.5% though the accuracy rather decreases. In the naive scenario, common TagSets such as Room can be simply identified by a few examples, represented by initial high accuracy but low MacroF_1. While the benefit of this layer is consistently higher than the naïve approach between similar buildings, it is unclear for buildings in different campuses.

Overall Performance

We evaluate five directions of active learning with knowledge transfer to cover different types of coverages found in Fig. 4.3a. While multi-label Zodiac and Scrabble select 10 examples per iteration, ProgSyn picks a sample in each iteration from the beginning. Due to the long experiment time as five minutes to a few hours per iteration in learning CRF model, we restrict samples per source building to 200 instead of the entire 400. The analysis in Section 4.3.4 already shows that learning with 200 B_S samples gives similar results to 400 samples for the first stage.

As an active learning framework, Scrabble outperforms both ProgSyn and multi-label Zodiac for $Accuracy$ and $MacroF_1$ in general. In C-1, Scrabble has 83.2% $Accuracy$ and 25.3% $MacroF_1$ while ProgSyn shows 42% and 12% initially with 10 samples. Scrabble also continuously outperforms and can reach 99% $Accuracy$ while ProgSyn cannot. Compared to multi-label Zodiac, $Accuracy$ and $MacroF_1$ of the best case of A-1,A-2 $\Rightarrow$ A-3 are 83.2% and 25.3% initially with 10 samples, while multi-label Zodiac's are 74.6% and 9.78%.

Furthermore, it shows the knowledge accumulation by showing adding more examples from source buildings results in similar or better initial inferences. $Accuracy$ and $MacroF_1$ of learning A-3 without source buildings are 73.6% and 11.9%, which 8.6% and 15.5% lower than learning with source buildings initially. However, we again experience $MacroF_1$ degradation in later stages possibly caused by overfitting. Our hypothesis is that source buildings' data distribution could confuse the evaluation metrics used for sample selection. The coverage of the words and Tags would affect the results as well. While using B-1 for A-3 improves the accuracies consistently, using A-3 for B-1 disturbs the learning rate. A-3 has more diverse patterns than B-1 as in Section 4.3.1, which would add noise to B-1's model more than the other way.

Overall, Scrabble always shows better performance than the other baselines including ProgSyn and the multi-label Zodiac. When 10 samples are initially given, Scrabble has 67%/31% $Accuracy/MacroF_1$ than ProgSyn's 42%/12% for C-1, and 83%/25% than the multi-label Zodiac's 75%/9.8% at best. To achieve 99% $Accuracy$, Scrabble requires 100 examples for C-1 while ProgSyn cannot achieve. For A-3 using A-1 and A-2, Scrabble requires 160 examples while the modified Zodiac needs 280 examples for 98 % $Accuracy$. Scrabble shows the possibility of reusing the models with intermediate representations though a model that is more generalizable across different data patterns is still a future research topic.

4.4 Scrabble Extensions

4.4.1 Learning from Time-Series Features

A way to augment the transfer learning process is by using time-series data, of which features are more common across different types of buildings [HWOW15b]. Tags can be differentiated using the features from the data collected by the building [GPB15b]. E.g., a *Temperature* related sensor may have an average of around 70^oF for indoor temperature. We choose the source building to be A-1 and the target building to be A-3. We extract 16 time-series features, such as mean and Fourier Transform values, from both buildings to train a random forest model as a multilabel classifier to model Tags that points represent in the source building. We infer Tags of interest at the target building with the learned classifier. The result is shown in Fig. 4.7, where we compare the precision and recall.Fig. 4.7 shows that validation and test sets track each other, which indicates that time-series features can be used to augment the transfer learning process. However, the competence to augment the transfer learning process is limited to how diverse the

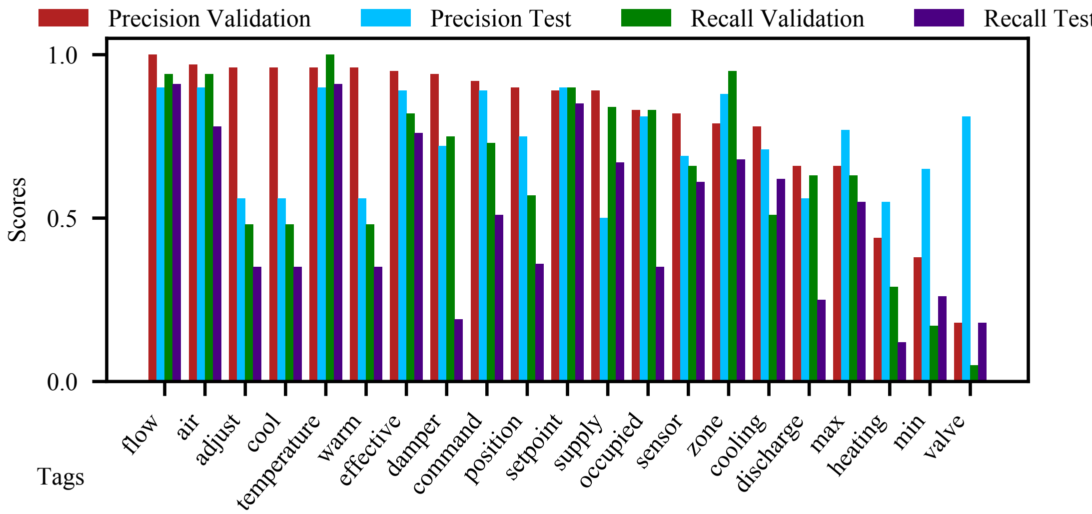


Figure 4.7: Tags Inference from Timeseries Features with Source Building (A-1) at the Target Building (A-3)

data is at the source building. From Fig. 4.3b we can see that the Tag coverage for A-1 ⇒ A-3 is 75%, this means that our model, initially, will not be able to represent 25% of the Tags for building A-3. Furthermore, Tags occurrences in buildings are significantly biased as discussed, there are quite a few Tags which only have time-series data representing those particular Tag, which is not enough data to train the model adequately. Only several Tags with significant numbers can be properly modeled by timeseries features.Hence, it limits the effectiveness of augmenting the learning process. We add the Tags determined by timeseries features to the ground truth and test the TagSet classifier's performance as shown in Fig. 4.5. Its accuracy is similar to the original TagSet classifier, but Macro F_1 decreases due to the improper representation of rare Tags by timeseries features.

4.4.2 Semantics Postprocessing

We identified TagSets from raw metadata. There is more information embedded in the raw metadata both explicitly and implicitly. We infer identifiers (IDs) of entities in the same way

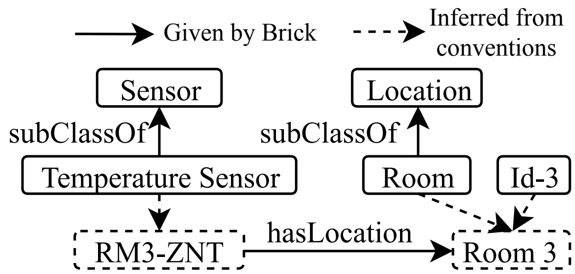


Figure 4.8: An Example of Semantic Postprocessing with Brick

as TagSets though they are not included in the evaluation metrics because they may overstate our performance. Two types of IDs such as equipment number or a hall name are positionally defined to indicate whether they qualify a TagSet right or left to the ID. We can glue the IDs to TagSets to reconstruct the original entities' name. In Fig. 4.8, the ID's position confines *Room* to Room 3. *Temperature Sensor* does not have corresponding ID, but rather it is common to use the entire name as an ID. Another type of information is relationships between TagSets. A *Temperature Sensor* may reside in a *Room*, which is not explicit in metadata but a domain expert can infer. Such relationships can be also inferred by a schema with relationships such as Brick defines canonical relationships between entities in buildings. In Fig. 4.8, we can explicitly know that "RM3-ZNT" is a *Sensor* and "Room 3" is a type of *Location*. The only possible relationships in Brick for them is *hasLocation* so we can explicit the semantic relationships among them without human intervention. The above two mechanisms are excluded in our analysis due to the lack of technical novelty but notable for practical use.

4.4.3 Applying to Project Haystack

Project Haystack [hay] is another popular standard metadata schema. Instead of Brick's TagSet/class concept, a user associates only tags to an entity in Haystack. For example, a sensor may be associated with the tags as "zone", "temp", and "sensor". Such Haystack tags can be corresponding to a Brick Tag though Brick further aggregates Tags to construct a TagSet as a class name. Thus, Haystack is a subset of Brick in terms of expressivity so algorithms that can infer

Brick can easily do for Haystack as well. In Scrabble, the first stage using CRF alone is sufficient to infer Haystack as it maps raw metadata to Brick Tags, which is theoretically equivalent to Haystack tags.

4.4.4 Limitation of Scrabble

Assumption of Identical Probability Distributions

As we note in Section 4.3.4, while Scrabble improves knowledge transferability across different but similar buildings, its design assumes that the probability distributions across different buildings are the same. However, in practice, the collection of data might not come only from similar buildings so we should be able to accommodate different buildings. In future work, we can add domain adaptation using gradient reversal [GUA+16] or a neural adapter [CM19] to mitigate the difference among different buildings. While this can be easily added to the second layer (Tags $\Rightarrow$ TagSets), the first layer (characters $\Rightarrow$ Tags) would require a more sophisticated method because native CRF uses statically defined feature sets while domain adaptation techniques dynamically adjust the features.

Ignorance of Relationships between Entities

While efficiently extracting semantic information from raw metadata, Scrabble is limited to the information contained in the raw metadata. In other words, if not encoded in raw metadata, Scrabble cannot infer other information types such as relationships between different points and equipment. Even though inferring relationships is not in the Scrabble's scope, we have observed the necessity of incorporating different algorithms, which we will discuss more in-depth at Chapter 5. We will further investigate algorithms to infer relationships in the following sections as well.

4.5 Quiver: Using Control Perturbations to Increase the Observability of Sensor Data in Smart Buildings

Active control is a promising approach to address the lack of information available, as carefully designed control perturbations can reveal insights into system behavior that is not observed in regular operation. Recently, Pritoni et al. [PBCM15a] showed that the mapping between the Air Handler Units (AHU) and the corresponding terminal units in the building Heating, Ventilation and Air Conditioning (HVAC) system can be inferred with 79% accuracy with control perturbations compared to 32% accuracy with data analysis alone. Control perturbations have also been studied for Fault Detection and Diagnosis (FDD) [WAA+12, PC15] and fault-tolerant control [FBK09, PCC15] in HVAC systems as it eliminates mundane manual testing and fixes some classes of faults automatically.

We expand on these ideas and show that active control mechanisms can be used as an integral part of a data model. Control based interventions are not used in practice because of equipment and safety issues. We empirically explore control in a real building HVAC system. We build Quiver, a control framework that allows us to do control experiments *safely* on the HVAC system by constraining control input that satisfies criteria such as range of values, frequency of actuation and dependency between actuators. We deploy Quiver in our building testbed and use it to demonstrate three example applications that exploit control perturbations. First, we show that perturbations can be used to identify co-located sensors which has been shown to be difficult with data alone [HOWC13]. We co-locate data points in HVAC terminal units with 98.4% accuracy and 63% coverage. Furthermore, we map the dependency between sensors and actuators in the control system using control perturbations and probabilistic analysis. We identify dependency links between actuators with 73.5 % accuracy, with 8.1% false positives and 18.4% false negatives across five zones.

4.6 Quiver: Our Building Testbed

Modern buildings consist of hundreds of networked sensors and actuators for the operation and maintenance of various systems. These systems are typically overseen with Building Management System (BMS) which helps configure, monitor, analyze and maintain various systems. The sensors, actuators and the configuration parameters in the BMS are together referred to as *points*. We focus on building HVAC systems where BMSes are most commonly used.

Our testbed is a 150,000 sq ft building, constructed in 2004 and consists of a few thousand occupants and 466 rooms. The HVAC system consists of an Air Handler Unit (AHU) that supplies cool air to the building via ductwork using chilled water supplied by a central plant. A heat exchanger supplies hot water to the rest of the building using hot water supplied by a central plant. The cool air and hot water are used by local terminal units called Variable Air Volume (VAV) boxes to regulate the temperature of rooms. The area serviced by the VAV box is referred to as a *thermal zone*, which consists of a large room or multiple small rooms in our building. Figure 4.9 shows a schematic of the VAV box with the sensors and actuators installed for its operation.

VAVs have been commonplace since 1990s [Hyd03], and their basic working is well understood. The VAV regulates the amount of cool air provided using a damper, and if the zone needs to be heated, it regulates the hot water in the heating coil using a valve. The temperature sensor in the thermal zone provides feedback on how much cooling or heating is required. However, in the real VAV box, there are over 200 points that govern its working [vav03]. The essential sensors include: *Zone Temperature, Supply Air Flow, Reheat Valve Position* and *Damper Position*; and the actuator points include: *Reheat Valve Command, Thermostat Slider Adjust* and *Damper Command*. These actuators are controlled using many configuration points such as *Temperature Setpoint, Occupied Command, Air Flow Setpoint*, etc. These configuration points account for the majority of the points, and include nuanced parameters that ensure minimum airflow, set the PID loop settings, etc.

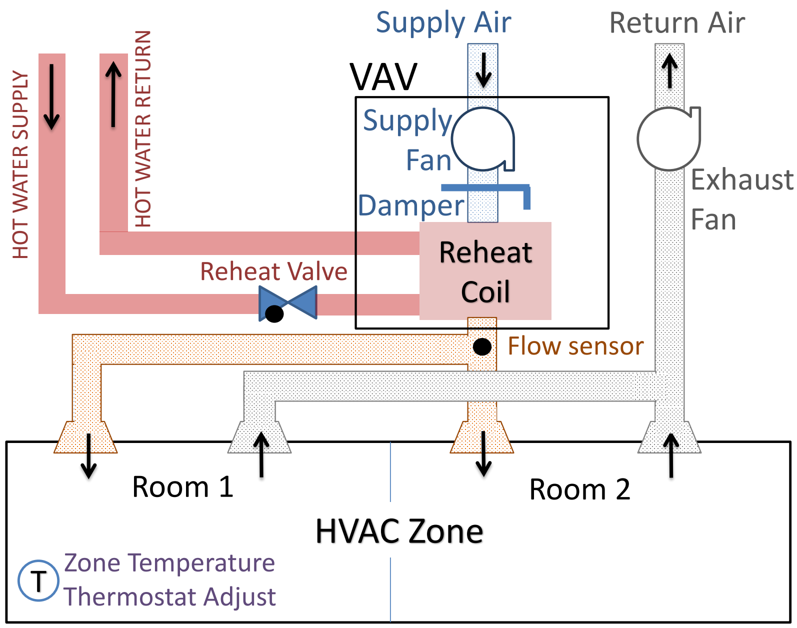


Figure 4.9: Sensors and Actuators in a Variable Air Volume (VAV) Unit Providing Local Control of Temperature in the HVAC system

Not all of these 200 points are reported to the BMS, and only the essential sensors and control points are exposed to limit resource usage and information overload for building managers. In our building testbed, 14 to 17 points are reported to the BMS for each VAV box. The points exposed to BMS changes depending on the vendor, type of VAV and the installation version used by the vendor. Even though the same model of VAV is used across all zones in our building, there are minor variations due to configuration changes, presence of supply/exhaust fans or lack of heating.

4.6.1 Data Collection and Control

The points in our building communicate with the BMS using BACnet [Bus97a], a standard building network protocol. We connect our server to this network to collect data and control the points in our building. We use BuildingDepot [WNA13], an open-source RESTful web service based building datastore to manage the points in the building, provide appropriate permissions to

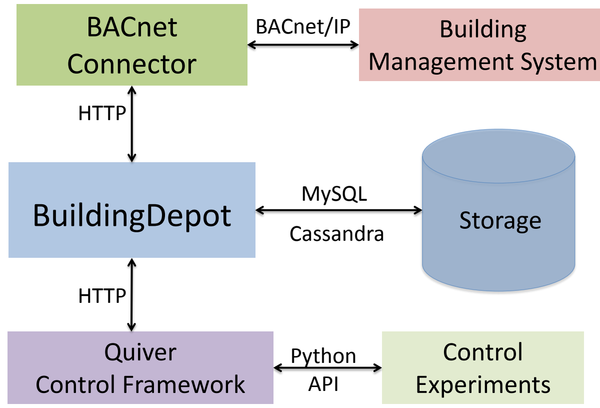


Figure 4.10: System Architecture of Quiver

Data collection and control is done via BACnet protocol using BuildingDepot web service [WNA13]. Quiver ensures that the control sequences of our experiments are safe and rolls back the system to its original behavior in case of failure.

developers, and search using a tagging mechanism. Our control framework Quiver works on top of BuildingDepot to manage control inputs from our experiments. Figure 4.10 depicts the system architecture of our deployment.

BACnet is a well-developed protocol with which developers can not only read and write points, but also schedule hourly control, mark holidays on a calendar, and even manage programs running in the embedded VAV controller. For simplicity, we only focus on read and write points, i.e., in BACnet terminology *Input, Output* and *Value* points. These points can have floating point, binary, or multi-state values, and in BACnet a floating point that can be written to is referred to *Analog Output*. Each of these *Output* points has an associated priority array. The default operation is performed at the lowest priority and the highest levels are reserved for emergency operations such as fire safety. Once a higher level priority is written to, the lower levels are ignored. An explicit write with value "0" needs to be written to the higher level priority in order to relinquish control to the lower levels.

The university's Facilities Management provides us with a fixed priority level in this priority array for our control experiments. We need to relinquish control back to the default priority level after our control experiments to ensure that our interference does not affect the regular operation of the HVAC system. Quiver ensures that all the points are relinquished after an experiment.

4.6.2 Points in Variable Air Volume Box

Figure 4.11 shows the points associated with VAV in our building BMS and how these points relate to each other. At the top of the figure, we have the zone *Temperature Setpoint* and *Occupied Command*, which in combination with thermostat input determine the temperature guardband within which the VAV is trying to keep the zone temperature. The temperature guardband is indicated by *Heating* and *Cooling Setpoints*, which represent the lower and upper bounds of temperature respectively. There are three occupancy modes: *Occupied*, *Standby* and *Unoccupied* during which the temperature bands are 4^oF, 8^oF and 12^oF respectively. During the *Occupied* mode, minimum amount of airflow is maintained to ensure indoor air quality. The *Thermostat Adjust* allows changing the temperature setting by $\pm 1^oF$, and the *Temporary Occupancy* maps to a button on the thermostat which when pressed puts the zone to *Occupied* mode for two hours during nights/weekends.

The *Heating* and *Cooling Setpoints* determine the behavior of the VAV control system with the measured *Zone Temperature* completing the feedback loop. These three points determine the *Cooling* and *Heating Command* of the thermal zone. The *Cooling Command* determines the amount of cool air required for the zone and determines an appropriate *Supply Air Flow Setpoint* that is between the designed minimum and maximum supply air flow. When the *Cooling Command* is high (~ 100%), feedback is sent to the AHU to decrease the supply air temperature to meet the cooling needs of the thermal zone. The *Heating Command* determines the amount of reheat required by controlling the *Reheat Valve Command*. During heating, the airflow is set to the minimum to reduce chilled airflow from AHU, and this airflow is increased when high *Heating Command* (~ 100%) fails to heat up the thermal zone sufficiently. A high *Heating Command* also sends a signal to the heat exchanger to increase the supply water temperature. Note that only one

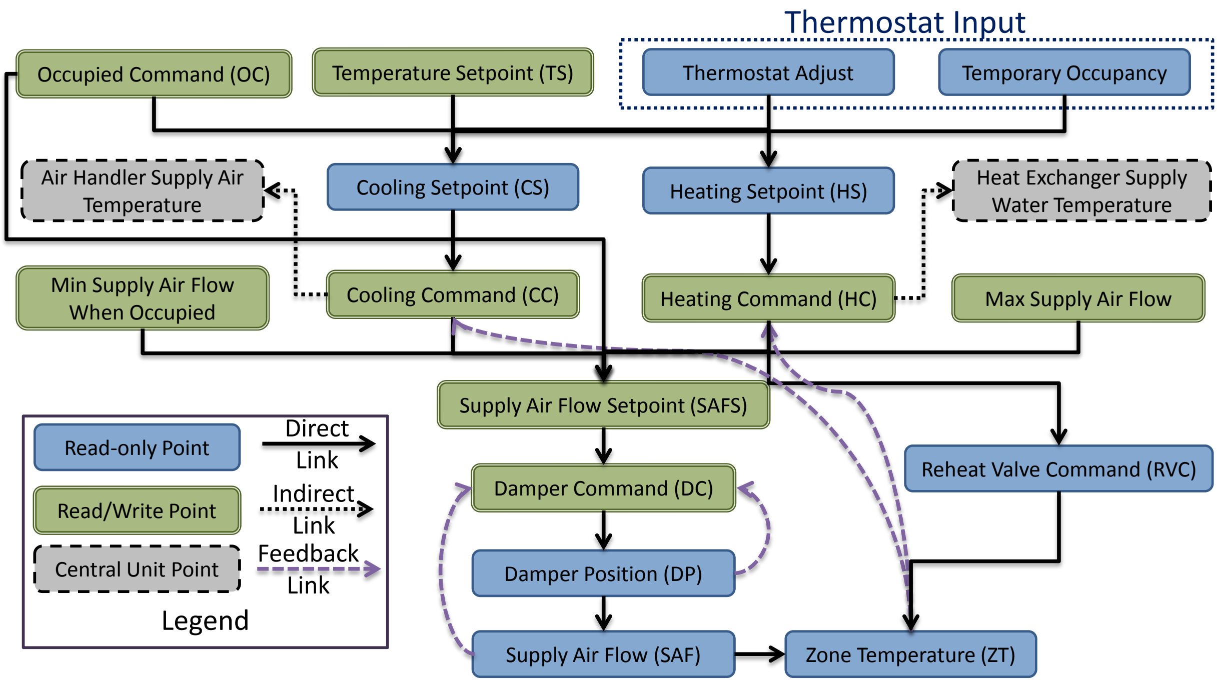


Figure 4.11: BMS points associated with VAV in our building testbed. The dependency between the points as shown by arrows is mapped based on domain knowledge. Read-only points are either sensors or configuration points which cannot be changed. Read/write points can be changed via BACnet.

of *Heating* or *Cooling Commands* can be >0% at a time.

The *Supply Air Flow Setpoint* determined by the cooling/heating requirements in turn determines the *Damper Command* which is the amount of damper actuation required to match the setpoint to the measured *Supply Air Flow*. The *Damper Position* sensor also provides feedback to set the appropriate *Damper Command*. There is a separate PID loop associated with setting each of *Heating Command, Cooling Command, Supply Air Flow Setpoint* and *Damper Command*, and there are PID parameters such as those that govern proportional gain and integration time, but these are hidden from the BMS.

4.7 Quiver: Learning with Control Perturbations

We use Quiver to learn more information about the sensor and actuator points inside a building using control perturbations. We define control perturbation as any changes made to actuators that deviate from typical HVAC operation. We confine all of our control experiments to nights/weekends, or in unoccupied zones only, to alleviate any effects on occupant comfort. We focus our control experiments towards addressing three important smart building applications:

- Identifying points which are co-located with a VAV box.
- Mapping the dependency between VAV actuator points.

All of our data analysis is implemented using Python Scikit Learn library[3].

[3]scikit-learn: `https://scikit-learn.org/`

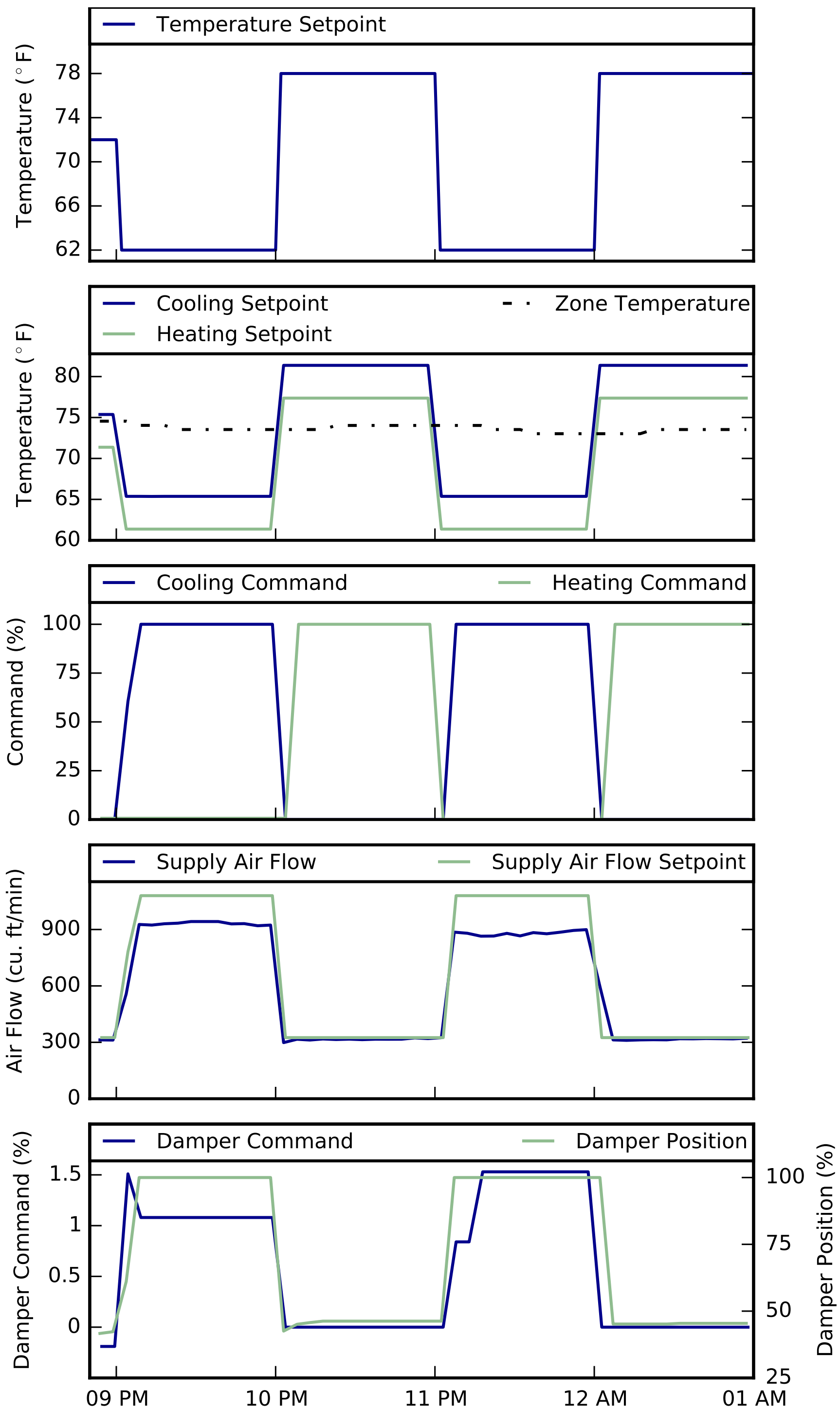


Figure 4.12: A sample of co-location experiment

The *Temperature Setpoint* is oscillating between $62^o F$ and $78^o F$ for four hours at night (the top graph). The VAV points which react to these changes (remaining four graphs) can be co-located by using temporal data analysis of this controlled period.

4.7.1 Control Perturbation for Co-location of Points

The location of sensor and actuator points might not be readily available in the BMS for old buildings. Even in buildings where location information is available, it could be inconsistent due to errors in manual labeling processes [BCH$^+$15, GPB15a]. It is also difficult to co-locate points using historical data alone as many VAVs function similarly, and the variation of data is not enough to distinguish them apart [HOWC13]. Control perturbations can be used to force the control system to unusual operating points, and co-located points that respond to this perturbation can be clustered together by data analysis.

We assume that we already know the type of points in the building which can be obtained using recently proposed methods [BCH$^+$15, HWOW15a], but do not know if they are co-located or how these points relate to each other or affect the control system. We do not use the location information already integrated into Quiver for these experiments. We perturb the actuator point identified as the *Temperature Setpoint* (TS) of a randomly chosen zone, and identify the corresponding co-located points using the temporal data features of other points. Towards the end of this section, we discuss how we can relax the assumption of knowing the point type apriori.

Figure 4.12 shows an example control sequence, where we change TS four times across four hours from low ($62°F$) and high ($78°F$), and the corresponding VAV points in the same zone that react to its changes. We chose such an oscillation of TS as it deviates substantially from the normal operation so that we can easily distinguish the controlled zone from the rest of the zones under the normal operation. This control sequence was chosen empirically, and we show that even such simple control sequences can be effective for co-location of points. However, as we show with our experiments, the effect of control sequences do affect the quality of results. We do not focus on designing generic control sequences in this paper.

We extract basic features such as amplitude, mean and standard deviation from the observed timeseries data. We also extract the Dynamic Time Warping (DTW) distance [BC94] between the applied TS signal and the point under consideration. DTW compensates for the time delay in the reaction and change in sensor values due to a control action and quantifies the difference between the shape of the signals. In addition, we exploit our pulse control and analyze the Fast Fourier Transform (FFT) after normalizing the data and use the Euclidean distance between the FFT of the point signal and FFT of the TS signal. We refer to this feature as “L2 norm of FFT” or “LFT”. We ignore frequencies beyond 0.0005 Hz, i.e. a period of 30 minutes, because we only focus on changes caused by our low-frequency control sequence.

We extract these features for all the VAV points in the building and identify the outlier points. In principle, the point which deviates the most from regular control operations would be co-located with our TS points with high probability.

Figure 4.13 shows the distribution of all the *Zone Temperature* (ZT) points in our building across three features – DTW, LFT and range – with a control sequence of two changes to TS. The zone under control is marked in red, and as observed, the red point is far away from most of the points from the other zones in the building. However, there are still a few points which are also differed significantly from most zones and it is difficult to distinguish the red point from those outliers. When we examine the data from the experiment where we made 4 changes to TS, the corresponding to changes to the ZT in the same zone resulted in much higher variation from those in uncontrolled zones. This is captured by our features as shown in Figure 4.14. Hence, with the help of a well-designed control perturbation, it is possible to mold the behavior of the control system for end-use applications.

We analyze the data for other point types to check if we can co-locate the zonal points successfully. In practice, we find that LFT feature alone is sufficient to distinguish the controlled zone points from the rest. Figure 4.15 shows compares the LFT of the controlled zone with other zones for point types: *Zone Temperature* (ZT), *Supply Air Flow Setpoint* (SAFS), *Supply Air Flow* (SAF), *Reheat Valve Command* (RVC), *Heating Command* (HC), *Damper Position* (DP), *Cooling Command* (CC), *Heating Setpoint* (HS) and *Cooling Setpoint* (CS). We performed this control experiment on eight zones in our building, and we co-located the listed points with 98.6%

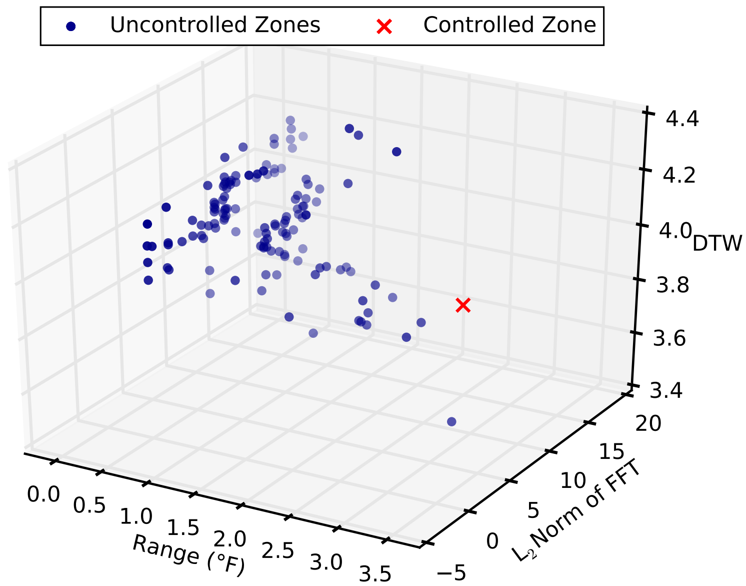


Figure 4.13: Co-location of Zone Temperature by perturbing the Temperature Setpoint. Two changes of Temperature Setpoint are applied over 2 hours in the controlled zone.

accuracy. We only failed to identify the correct *Damper Position* point for one of the zones, leading to a drop in accuracy.

The *Damper Command* (DC) is a differential actuator that sets the change that needs to be made to the damper. There are several VAVs in the building which constantly change their DC for minor variation in the airflow, and the features we extracted – DTW, FFT, mean, variance, number of changes – failed to differentiate the DC of the zone under control from the rest (Figure 4.15). More sophisticated data analysis or perturbance signals are required for co-location of DC points. We could only co-locate DC points in two of the eight zones with our current method.

Another issue with these control experiments is that we can only co-locate those points which react to changes in TS (see Figure 4.11). Points such as *Occupied Command* (OC) and *Thermostat Adjust*, which are external inputs to the VAV control system, cannot be co-located. To remedy this, we perform a second set of experiments which oscillates the OC similar to our TS control perturbations. We successfully co-locate OC and points such as *Cooling Setpoint* and

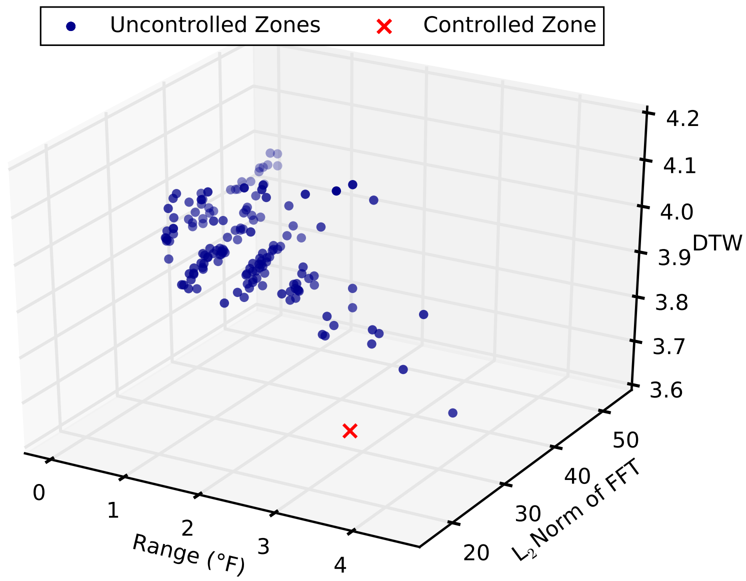


Figure 4.14: Co-location of Zone Temperature by perturbing the Temperature Setpoint

Four pulses of Temperature Setpoint is applied across 4 hours in the controlled zone.

Supply Air Flow that have been already co-located with their corresponding TS point. Thus, all of these points can be marked as being co-located in the same VAV. We performed the *Occupied Command* oscillation experiments across four zones with 100% success rate in their co-location results. We could not perform similar experiments on the thermostat points (*Thermostat Adjust* and *Temporary Occupancy*). They cannot be controlled by our platform as thermostats produce their data contiguously, and we acknowledge this is a limitation of our proposed method.

We repeated our TS control experiments on a hot day, and found that the same control perturbations cannot co-locate heating related points – *Heating Command* (HC), *Reheat Valve Command* (RVC) – as they are not triggered sufficiently due to hot outside weather. The zone cannot be cooled down enough to activate heating (HC, RVC) when its TS is changed to the high value. We need to change our control perturbations to excite these points specifically. Thus, the perturbation signature needs to be sensitive to external conditions and confounding factors.

Note that in Figure 4.15, the LFT feature of the controlled zone for most point types

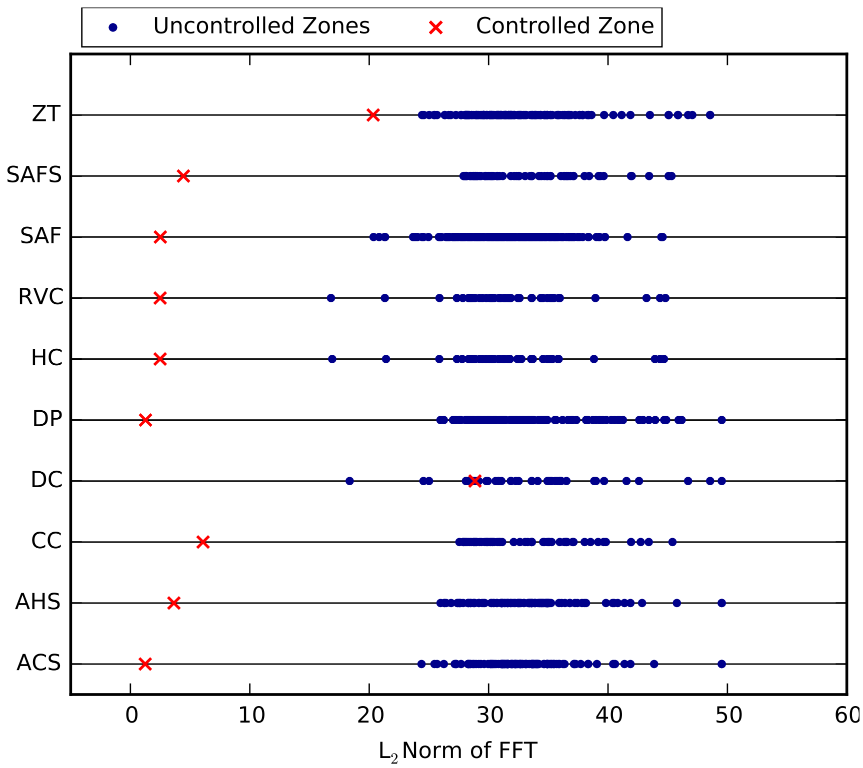


Figure 4.15: Comparison of the L2 norm between FFT of VAV points and the FFT of the controlled temperature setpoint

The points corresponding to the controlled VAV have a much lower L2 norm compared to regular zones for eight point types, but fails to capture the difference in *Damper Command*.

except ZT and DC differs significantly from the rest of the points. The controlled zone's LFT feature of ZT is not distinguishable from other types' LFT features of the other zones though it can be co-located within the same type. Contrary to DC's different operational behavior, ZT's signal response is slower than the other types due to the heat capacity of zones. If we assume that we do not know the point type, then the points except ZT and DC can be obtained as outliers from the points belonging to normal zones. However, we would need to design an appropriate threshold or clustering technique to identify the outlier points correctly.

Overall, with our control perturbations based co-location, we successfully co-located 10 out of 16 point types (63% coverage) with 98.4% accuracy across eight VAV units. We co-located the *Occupied Command* points using auxiliary control experiments for four zones. We failed to co-locate *Damper Command* due to its divergent behavior. We also do not co-locate 4 point types which do not respond to control perturbations.

4.7.2 Control Perturbation for Causal Relationships between Points

We now focus on understanding the working of the VAV, and how the points relate to each other. As the types of points exposed are different across vendors and equipment, it is necessary to understand the context of these points, and map it to a model that can be used by other applications. These models can be built using domain knowledge, technical documents and historical data analysis [FRS+13, Jen96] as demonstrated by our dependency graph in Figure 4.11. We propose control perturbations as an alternative to these methods, which can be used for either verifying already developed models or used for older buildings where the available information is insufficient for modeling using other methods.

We assume that we already know the point type and the co-located points in a VAV. We focus on modeling the dependencies between the actuator points (or read/write points). We write to the *Temperature Setpoint* (TS) of a zone with a randomly chosen value every 20 minutes for 6 hours. The goal of this control sequence is to identify the points that react to changes in TS, and so we choose random values within limits for perturbing different operating points of the control system. For every change in TS, we analyze the behavior of the VAV points for 10 minutes, and note the points whose values change during this period. The threshold of change is one standard deviation for values observed for the past 12 minutes. We chose these times so that we can isolate changes that occur due to the change in TS rather than other external factors such as solar radiation. For the duration of the experiment, we calculate a final probability for each of the points as the ratio of the number of changes observed for the point and the number of changes in TS. We repeat this experiment by perturbing all the actuator points - *Occupied Command* (OC), *Cooling Command* (CC), *Heating Command* (HC), *Supply Air Flow Setpoint* (SAFS) and *Damper Command* (DC). Note that we ignore the points related to minimum and

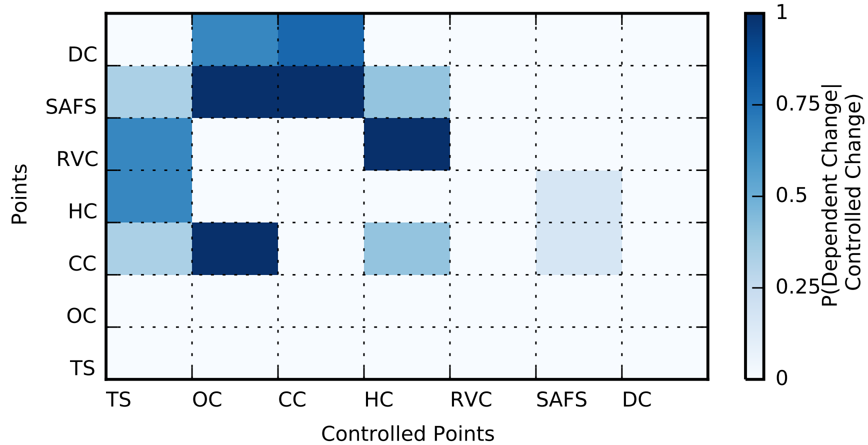


Figure 4.16: Color map showing the changes induced by control perturbations of each actuator point on other actuators. Probabilities are calculated as the ratio of number of changes observed in a non-controlled actuator and the number of changes made by the controlled actuator.

maximum supply airflow as they are constants.

Figure 4.16 shows a color map representing the probabilities obtained by perturbing each of these points. The changes to TS affects all the actuators except OC and DC, while changes to actuators like CC cause changes only in DC and SAFS. With the help of this color map, we can understand which points are being affected by each of the actuator points. However, this does not precisely decide the dependency between points as points which are lower in the dependency tree such as SAFS get affected by almost all of the actuation experiments. We find the behavior of DC to be unpredictable, and the changes that occurred in DC with our control perturbations were lesser than our set threshold. We perform these perturbations across five zones.

Figure 4.17 shows the relationships obtained as a result of our analysis. The green solid links indicate relationships that are true and confirmed with the analysis. The red dashed links show relationships that are not true, but are shown to be related by the analysis. In general, the above experiments cannot identify the true links when a “cycle” is formed in the graph. Here, by “cycle” we mean that there are multiple paths from one point to another in the graph. We perform more control experiments to negate the red links, and also confirm the blue dotted links which form a cycle with the green links.

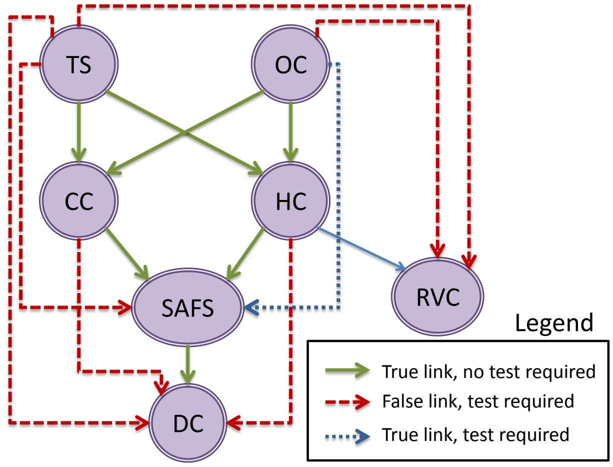


Figure 4.17: Dependency Links Obtained by Perturbing each of the Actuators

The links which require further testing are verified using conditional control perturbations.

Consider the cycle formed between the points TS, CC and SAFS. In order to verify if the link between TS and SAFS is correct, we perform a conditional control perturbation, where we change TS but force CC to be unchanged, which is called *Graph Surgery* [Pea09]. We repeat this experiment for at least four changes of TS. If TS were directly affecting SAFS, we would observe that SAFS changes even when CC is held constant. We verify each of the red and blue links this way. When there are more than three points in a cycle, such as that with TS, OC, HC and RVC, we ensure several combinations of TS and OC are performed to test the validity of the link. In some of these experiments, we preset the TS value to a fixed value for appropriate conditions that can activate other points such as HC. As we note at Section 4.6.1, an external variable, ZT, may disable HC or CC in certain condition though ZT is not an actuator. We let the VAV control system to settle to a steady state after our change of TS before performing any dependency experiments.

We performed these experiments on five zones in our building, and verified the links with 73.5 % accuracy with a false positive rate of 18.4 % and false-negative rate of 8.1 %. All the false positives and 64.7 % of the false negatives are due to the external variable, ZT. Thus, we can discover the dependency between actuator points in the VAV using our control experiments.

However, in order to discover the complete dependency map as shown in Figure 4.11, we need to use maximum likelihood data analysis as the behavior of read-only points cannot be controlled directly.

4.8 Quiver Future Work

Quiver is a control perturbation framework for increasing information observability of data. While Quiver shows the possibility of identifying relational information not encoded in any raw metadata, there are a few technical challenges to be adopted in practice. First, the control perturbation signal is intrusive to occupants' daily lives. An exemplary control signal, such as large pules over Temperature Setpoints, would be out of temperature bandwidths that the occupants can tolerate. This is often called exploration and exploitation trade-off where the goal is to maximize exploitation while minimizing the exploration time as much as possible [Mar91]. We need to explore the design space further, for which there a few candidates as (i) optimizing the perturbation signals with smaller amplitudes and (ii) shorter periods, and (iii) applying the signals only at nights or weekends when the rooms are unoccupied. While Quiver will be a base framework, we leave such design space exploration as future work.

Furthermore, we need more statistically robust perturbation signals that are effective in various situations. While our naïve pulse would have been unseen in most of the existing building control policies, a more robust perturbation signal should be able to cause novel patterns while effectively affecting all the rest of the dependent points. It further means that the perturbation signal might vary across different systems depending on the patterns of existing data. Though an optimal signal design has been widely studied for system identification [JNMM18, GTBC05] whose goal is to fully understand system functions, we need a subset of the functions such as co-location and causality whose optimality might be achieved in different ways.

4.9 Related Work

4.9.1 Entity Resolution

Normalizing unstructured or semi-structured data is a well-known problem in various areas including data matching. The goal of Entity Resolution (ER) is to identify data entries or records that represent the same entities in different databases [Chr12b]. Records for the same entities may vary across different sources, for the same reason in metadata related to SCADA systems. Most of the existing work in ER is to exploit string similarities between entries such as learnable string similarity metrics [BM03], declarative grammar rules [AK09] and learning similarities from examples [CCGK07]. However, it is often impractical to learn models from samples as producing labeled data requires significant effort. Active learning approaches have been proposed to reduce the number of samples to learn an ER model faster [SB02, CVW15]. Active learning frameworks learn models with a limited number of samples initially and iteratively choose either uncertain inferred samples [SB02], samples maximizing recall [BIPR12] or most heterogeneous samples [CVW15]. Though our goal is to identify entities with variations like ER, the raw metadata in SCADA systems contain multiple pieces of information whereas ER commonly identifies entities with a one-to-one mapping.

4.9.2 Transfer Learning

The lack of labeled data is a common problem in the real world. Domain adaptation consists of reusing a classifier learned in one domain and applying it in another. Sample reweighting with importance can adjust source samples' distribution to match the target domain's and there are various ways to infer the importance [Shi00, GSH+09, BBS07]. However, our application datasets often break the assumption that the source domain should contain the target domain labels. Zero-shot learning is a two-step classification framework where data inputs are mapped to semantic codes or attributes [PPHM09, FEHF09, RPT15] as an intermediate representations instead of labels. The relationships between attributes and actual labels can be given by domain experts or learned. One can easily infer a new class if its mapping to attributes is given because data inputs to attributes are already learned. We adopt this paradigm to reuse learned patterns as source data and infer undiscovered labels as a new domain.

4.9.3 Building Metadata Normalization

Prior works have proposed data mining approaches to normalize raw metadata in buildings. Proposed approaches include learning a multi-class classifier from Bags of Words (BoW) of raw metadata [HWW15, BVNA15a] and learning regular expressions from examples to parse raw metadata [BHC+15a]. All of them adopt active learning frameworks to reduce the number of examples to learn based on clustering BoWs. However, the proposed approaches either identify data point type only [HWW15, BVNA15a] or require knowledge about regular expressions with which domain experts are usually not familiar with [BHC+15a]. Furthermore, these works require many labels to normalize each building and do not apply the learned knowledge across buildings. Other supervised approaches learn classifiers on time series features to label point type [GPB15b] or find the best match between the raw metadata and corresponding labels with edit distance [SPG14a]. Hong et al. propose a transfer learning framework similar to our work for identifying point types [HWOW15b]. They use locally weighted ensemble methods [GFJH08] to learn classifiers from a source building's time series features based on BoW similarity with a target building. However, unlike Scrabble, their method is limited to identifying point types and have low precision/recall of 36% and 85% respectively.

4.9.4 Learning with Control Perturbation

The problem of discovering system characteristics and models using available data is called *system identification* [Lju98], and is a well-studied subject in control systems research.

Using control perturbations for system identification is also well known [Lju98, God93], and the design of control perturbations, also referred to as auxiliary or secondary signals, has been studied for modeling different types of control systems [GBT99, GTBC05]. System identification techniques have also been used for HVAC modeling [TC98], and some prior work has explored using control signals for system identification [SBV97, VCL95]. However, all of these works focus on modeling the control system or perform control optimizations and do not address the identification of contextual information such as location, point type or dependency graphs. Moreover, the control perturbation methods used for buildings are only verified using simulations.

Active HVAC control on real systems has been used for fault diagnosis [WAA+12] and fault tolerant control [FBK09]. These works and other simulation-based studies which propose fault tolerant control [LD01, WC02] assume the contextual information about the system is already available. We focus on discovering contextual information using active control.

Co-location of sensors has been studied before [FOCE12, HOWC13] but they use sophisticated data analysis algorithms. These methods fail when the points from different locations have similar data characteristics during regular usage. We show that with perturbations we can excite the local control system to unique operating points and co-locate points with high accuracy using simple data analytics.

Point type identification has also been studied earlier using both metadata [BVNA15b, BCH+15] and data analytics [GPB15a, HWOW15a]. These works show that metadata alone can be unreliable and requires significant manual input for accurate type identification, and the data analytics based method is useful for some point types but fails for others. Our results conform to the data analysis works, and we show that perturbations can be used to identify the points which are difficult with data analysis. We also note that Hong et al. [HWOW15a] focus on transfer learning across buildings, while we focus on transfer learning within the building across the different instances of VAV units.

4.10 Summary

Automatic metadata normalization is an essential tool to have Brick widely adopted as well as it is a common problem in any data management systems. We have shown two tools to efficiently normalize metadata of existing systems as Scrabble and Quiver. Scrabble is the best-in-class algorithm for extracting semantic information from raw metadata via a semi-supervised learning framework. It uses fine-grained labels (Tags) as an intermediate representation to improve the transferability of data across different buildings. Quiver is a control perturbation framework to generate new patterns in data to improve information observability. We exploit such new patterns for learning relationships across different entities such as co-location and control dependency, which are not observable in either existing historical timeseries data or raw metadata.

While Brick has shown the possibility of using a structured metadata schema as a base for application interface, we should various methodologies to easily use Brick. There are millions of buildings in the United States [Uni15] which deserve modern building applications for energy savings and better living conditions. Our set of metadata normalization methods will help to deploy a Brick-enabled infrastructure for those buildings with minimal human effort and eventually attract app developers to adopt Brick as a base information model. Our methods are a vital link for deploying building apps at scale.

4.11 Acknowledgment

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of the 5th ACM International Conference on Systems for Built Environments, 2018 by authors Jason Koh, Bharathan Balaji, Dhiman Sengupta, Julian McAuley, Yuvraj Agarwal and Rajesh Gupta. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, also contains the material as it appears in arXiv, 2016 by authors Jason Koh, Bharathan Balaji, Vahideh Akhlaghi, Yuvraj Agarwal and Rajesh Gupta. The dissertation

author was the primary investigator and author of this paper.

Chapter 5

A General Framework for Metadata Normalization Methods

For metadata normalization, prior works have proposed methods to partially automate metadata normalization [SPG14b, BHC+15b, BVNA15a, HWOW15a, KBS+18, KBA+16], with each of them focusing on particular aspects of metadata.Some methods recognize all entities using the raw metadata [SPG14b, BHC+15b, KBS+18], including the site, floor and room identifiers, and point type. Other methods identify only the point types based on either the raw metadata [BVNA15a, HWW15], timeseries data [GPB15b], or both [HWOW15a]. Yet other methods focus on inferring the relations between entities, including the spatial relationships [HOWC13, KAB14] and functional relationships [KBA+16, PBCM15b]. In order to reduce the manual effort, these methods either only exploit the structure available within each building [SPG14b, BHC+15b, BVNA15a, HWW15, GPB15b] or transfer information from one building to the next [HWOW15a, KBS+18]. Importantly, while all of these prior works exploit common attributes of each point — the alphanumeric text-based metadata and/or the numerical timeseries readings, they differ significantly concerning the inference scope, input/output format, and structure, algorithm interface, and evaluation metrics [WBK+18]. The resultant lack of compatibility among thevarious methods precludes the possibility of combining and comparing themsystematically. Most importantly, there is still no standalone, versatile solution so far.

Generic machine learning platforms such as MLJAR [mlj16], OpenML [VvRBT13], and MLlib [MBY+16] have recently emerged. However, while these ML platforms provide generic interfaces for standard machine learning tasks, they are too generic to serve as a usable interface for the unique building-specific human-in-the-loop process with diverse data sources and different input/output formats. We need a modular framework that provides a unified interface for exploring existing techniques as well as rapidly prototyping new algorithms, in order to advance the state-of-the-art in building metadata normalization. To this end, we design and implement Plaster, a modular framework akin to Scikit-learn[1] for building metadata normalization, which incorporates existing metadata normalization methods, along with a set of data models, evaluation metrics, and canonical functionalities commonly found in the literature. Together these enable the integration of different methods into a generic workflow as well as development and evaluation of algorithms. With the designed interfaces, Plaster can easily fit into existing building stacks, from commercial building management systems to open-sourced systems such as XBOS [AKC+17] and BuildingDepot [WNA13], that expose the access to metadata and timeseries data in buildings.

With Plaster, we also present the first systematic evaluation of the state-of-the-art metadata normalization methods via a set of unified metrics and datasets. Our evaluation covers a broad spectrum of metrics, such as how accurate each method is in inferring the same kind of labels, how many kinds of labels each method can produce, and how many human labels are required to achieve a accuracy. Our experiment results reveal that there is no one-size-fits-all solution and properly combining them could produce better results. This evaluation would not have been possible without Plaster, given the heterogeneity in earlier works. We believe Plaster provides a comprehensive framework for further development of new algorithms, techniques for metadata normalization within the building domain, as well as mapping buildings to a structured ontology

[1]scikit-learn, `https://scikit-learn.org/`, last access: 12/01/2019.

like Brick[2] [BBF+16a, BBF+18a, FKA+19], enabling seamless smart building apps.

5.1 Categories of Metadata Normalization Methods

We identify three dimensions along which various existing metadata normalization methods work: 1) the type of data sources exploited, 2) the kinds of labels produced, and 3) the degree of human input required.

There are three different types of **data sources** we can exploit in buildings. The raw **metadata** in BMSes, also referred to as *point names*, usually encodes various kinds of information about the control and sensing points, including the type of sensor, floor and room numbers, HVAC equipment ID, etc. The metadata within a building often exhibits clear patterns that can be leveraged, although it varies significantly across buildings and often does not generalize from one building to another. As a result, various works have leveraged such pattern for metadata inference [SPG14b, HWW15, BHC+15b, BVNA15a, KBS+18]. Secondly, modern BMSes also collect **time series** readings of each point in the building, which contain information that indirectly reveals what the point is and its relationship with others. For example, the range of the readings can indicate the type of sensor and the correlated changes in different streams can indicate the relationship. Works that leverage the characteristics of timeseries data include [HOWC13, KAB14, GPB15b]. Additionally, one may also perform **controlled perturbation** in a building, e.g., to manually turn off an air handling unit, and create new patterns in operations that help to reveal the functional relationships between entities more clearly [KBA+16, PBCM15b]. However, such control requires careful and sophisticated designs not to harm the system stability and occupants thermal comforts.

Existing metadata normalization methods focus on producing two **kinds of labels** — following the canonical definitions in Brick — entity types and relationships between entities.

[2] Brick: `https://brickschema.org`, last accessed: 12/01/2019

Table 5.1: Categorization of Metadata Normalization Methods

Method	Label Produced	Data Source	ML
Bhattacharya et al. [BHC+15b] Scrabble [KBS+18]	All entities	Raw metadata	AL
Zodiac [BVNA15a] Hong et al. [HWW15]	Point type	Raw metadata	AL
Fürst et al. [FCKB16]	Point type	Raw metadata	CS
BuildingAdapater [HWOW15a]	Point type	Raw metadata, Timeseries	TL
Gao et al.[GPB15b]	Point type	Timeseries	SL
Hong et al.[HGW17]	Point type	Timeseries	UL
Pritoni et al. [PBCM15b] Quiver [KBA+16]	Functional Relationship	System Perturbation, Point Label	UL

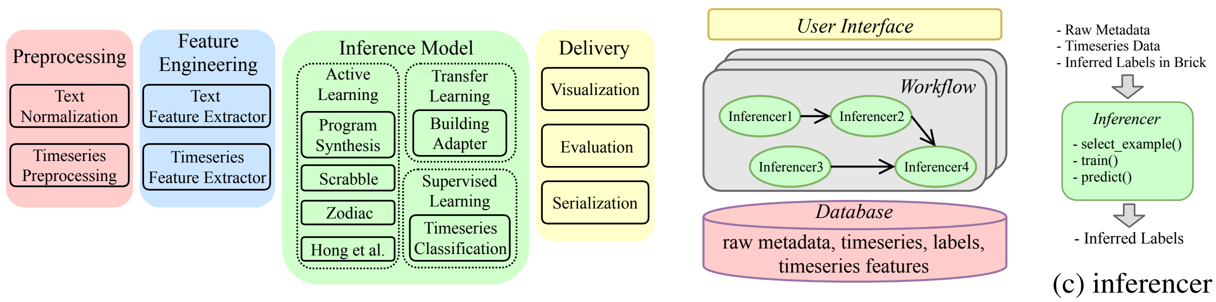
AL: Active Learning TL: Transfer Learning
SL: Supervised Learning CS: Crowd Sourcing
UL: Unsupervised Learning

The entity type refers to the type of measurement of a point and there is a wide variety in its possible set of labels, while the relationships include how points are connected to each other, whether they are in the same room/zone, etc. A few methods infer all the available information (e.g., both kinds of labels) encoded in the raw metadata [SPG14b, BHC+15b, KBS+18], whereas many others identify the point type only [HWW15, GPB15b, HWOW15a, FCKB16], which is the most important aspect of a point in buildings, or infer the relationships only [HOWC13, KAB14, PBCM15b, KBA+16].

While different methods all aim to reduce the amount of manual effort in normalizing metadata, the degree of **human input** required by each of them varies from fully supervised to semi-supervised to completely unsupervised. Particularly, supervision, or human input, in this context is the annotation or labels that a human expert provides to interpret the point for its type, location, relationship with others, etc. Supervised learning has been used to learn the point types based on timeseries data or raw metadata, where both clean, accurate labels from experts [GPB15b] and crowd-sourced labels from occupants in the building [FCKB16] have been explored in the literature. For the set of semi-supervised solutions, they employ active learning to iteratively select the most informative example and query an expert for its label to improve a model for normalizing the metadata, requiring the minimal amount of labels [HWW15, BHC+15b, BVNA15a, KBS+18]. On the other hand, transfer learning techniques have been developed to exploit information from existing buildings and completely eliminate human effort when inferring the metadata in a different target building [HWOW15a]. Similarly, Scrabble [KBS+18] is another method that exploits existing buildings' normalized metadata, but through an active learning procedure. Table 5.1 summarizes these methods with regard to the above criteria. In this work, we show that, while each of these techniques has its advantages, our proposed *meta*-framework – Plaster– can help to choose the right algorithm per user requirements as well as leverage different techniques in a complementary manner to yield better results.

Generic machine learning platforms such as MLJAR [mlj16], OpenML [VvRBT13], Microsoft Azure ML Studio [azu16], and MLlib [MBY+16] have recently emerged. These platforms have proved to be useful and facilitated tasks and research on machine learning. However, the building metadata normalization problem has more unique requirements: 1) it handles diverse types of input/output data, receiving as input timeseries data and/or encoded textual metadata, and produces a graph (such as Brick entity graph) as a final output, 2) it involves various types of learning frameworks including transfer learning, active learning, and supervised learning altogether, and 3) users would need to interact with the algorithm(s) through the abstraction of the building data such as a specific type of building metadata and identifiers of the data, rather than directly with the data itself. Consequently, existing frameworks cannot be adopted for metadata normalization tasks as are. Additionally, although not being directly related to the metadata normalization problem, there are frameworks in other domains that integrate different algorithms and create composable workflows, including general machine learning analytics [BORZ17], recommendation systems [Hug17, YBG+18], non-intrusive load monitoring [BKC+14, BKP+14]. Plaster is the first framework of its kind that enables the exploration and integration of various algorithms on building metadata normalization, as well as provides the ability to systematically compare related algorithms.

5.2 Plaster Framework



(a) Components in Plaster (b) Workflow (c) inferencer Interface

Figure 5.1: Plaster Architecture

Plaster adopts a modular design and incorporates a variety of components, among which the core is a family of inference algorithms. Each algorithm is abstracted as an ensemble of common functions, which allows the communication between different algorithms. Such a design enables not only the flexible invention of a workflow composed of any algorithms, but also the systematic comparison between different algorithms.

Plaster delivers a modular framework for benchmarking, integration, and development by providing two levels of abstractions common among existing methods. As the *first* level of abstraction, Plaster views a metadata normalization task as an ensemble of a key **inferencer** and several other reusable components that have canonical functionalities and interfaces. This way, we provide users with the flexibility in choosing the data model, learning scope, and inference algorithm as needed. As the *second* level of abstraction, an inferencer, which is the core component, comprises multiple common functions that we identify by summarizing existing metadata normalization solutions. Because of the unified interfaces and its modular design, Plaster facilitates the invention of new *workflows* where a user can connect different inferencers to essentially create a new algorithm without re-implementing prior algorithms.

5.2.1 Architecture

In Plaster, we abstract each method as an ensemble of components, and overall there are four categories of components as illustrated in Fig. 5.1a: preprocessing, feature engineering, inference models, and results delivery functions.

The *preprocessing* component includes standard functions such as denoising and outlier removal for timeseries data, and lowercasing and punctuation removal for textual metadata, via an interface to utilize existing libraries such as SciPy[3] and Pandas[4]. There are also database (DB) I/O functions for both the metadata and timeseries data. We use universally unique identifiers to identify points and one can access both the textual metadata and timeseries data through the identifiers. For the timeseries DB functions, Plaster builds upon an open-source library [arc14] piggybacked on MongoDB, which is dedicated and optimized for timeseries data operations on large data chunks. For *feature engineering*, there are a number of existing libraries, such as the most widely used scikit-learn [PVG+11b] and a recent effort – tsfresh [CBNKL18]. However, none exists as customized for the timeseries data from buildings, considering their uniqueness such as the distinct diurnal patterns. Hence, we incorporate and extend the feature sets[5] implemented by Gao et al. [GB18], which contain various feature functions customized for building timeseries data. In addition to the original feature sets, we provide straightforward programming interfaces for a user to select a subset of features out of these predefined features and a lightweight yet effective feature selection function based on LASSO [Tib96]. We shall demonstrate the effectiveness of the feature selector in Section 5.3.4. *Delivery* components consist of the set of evaluation metrics (detailed in Section 5.3.1), user interaction mechanisms to provide additional supervision for inferencer update as needed, and serialization tools that convert the inference results into the Brick format (e.g, triples and graphs).

[3]SciPy: `https://www.scipy.org/`, last access: 12/01/2019
[4]pandas, `https://pandas.pydata.org/`, last access: 12/01/2019
[5]We refer the readers to their original paper [GB18] for more details on each feature set.

5.2.2 Inferencer

At the core of Plaster is a collection of state-of-the-art metadata inference methods. We examine these algorithms and identify similar procedures among them. We therefore abstract these procedures as a series of common functions, encapsulate each as a parameterized interface, and formulate a standardized way of constructing an inference algorithm. We use an abstract class – inferencer– to represent an algorithm (e.g., Scrabble, Building Adapter, etc), which maintains its own model for metadata normalization under these abstract interfaces. Such abstraction decouples the complex procedures in individual algorithm and allows new algorithms to be easily included into the framework.

At a high level, an inference algorithm in the building metadata domain aims to achieve the best possible accuracy with the largest coverage using the minimal set of labeled examples. Therefore, an inferencer typically contains a few steps: 1) the algorithm selects as training set the most "informative" example(s) based on its own criterion and acquires the labels for the selected example(s) from a human expert; 2) the model updates its parameters based on the latest training set after the new examples are added in the previous step, and then 3) the model predicts all types of labels (e.g., point type, location, relationships, etc.) covered by the algorithm. Plaster abstracts each of the above steps as a function, viz, `select_examples()`, `train()`, and `predict()`, respectively, as shown in Fig. 5.1c; and we design an inferencer to be a composition of these functions. We shall note that, although these functions appear to be only able to compose an active learning-based procedure, we design the `select_examples()` function to be generic enough such that any fully to semi-supervised learning algorithm can fit into this template. When obtaining examples for a supervised or transfer learning algorithm, the `select_examples()` function simply includes all the labeled or transferred examples for training at one time, rather than being iteratively done as in active learning. In an *active learning* approach, these steps are repeated in iterations involving a human expert to best learn the model, while for a *supervised learning* or *transfer learning* approach, these steps are mostly executed just once with already labeled examples.

We also define standardized input/output interfaces for these common functions to enable the communication between different inferencers, which permits the creation of workflow as we shall discuss shortly. For inputs, an inferencer accepts three types of sources: raw metadata, timeseries data, and the corresponding labels of examples. We provide a wrapper to digest two types of raw metadata commonly found in existing systems: 1) point names accessible through vendor-given interfaces (e.g., Metasys) that are widely used in the literature, and 2) metadata in BACnet [ASH16] including entries such as `BACnet Description` and `BACnet Unit`. Timeseries data is stored as a series of timestamped values and the data for each specific point is associated with a unique identifier of the point for indexing and future retrieving. As each inferencer can learn and produce various types of labels as discussed in Section 5.1, an inferencer can take three kinds of labels at different granularities: point type labels, labels for all entities existing in the metadata, and character-level parsing with BIO tagging [RR09b].

The ultimate goal of each individual inferencer is to generate structured metadata. Since Brick is capable of representing different kinds of metadata such as the types of entities and the relationships between them, we express the `predict()` method's outputs of each inferencer following the Brick's format. Particularly, the outputs are a list of triples for entities and relationships in a building as explained in Section 3.1. Consequently, an inferencer is capable of representing different inference results in the same format. For example, Zodiac [BVNA15a] infers the point types, which can be represented as "X is a Y" triples, while Quiver [KBA+16] infers the co-location relationship for multiple sensors expressed as "X1 hasLocation Y" and "X2 hasLocation Y". Such different types of inference are serialized in the same format of Turtle [tur] using the vocabulary in Brick.

Additionally, for each inferencer we include the confidence of its inference results produced by the original algorithm in its output so that an inferencer is able to more flexibly sift through and use another inferencer's results. Specifically, we store the confidence for each produced triple within the inferencer. However, the notion of confidence is unique per inference algorithm with different meanings. For example, Support Vector Machine's confidence is usually measured by the distance to the hyperplane, while in Naïve Bayes, it is the probability of observing the example given the model's parameters. Thus, we restrict the interpretation of confidence score within each individual inferencer despite the values of those metrics being uniformly normalized to be between zero and one. We shall show how this is useful in real workflows in Section 5.3.3.

As a natural outcome, Plaster provides a standard benchmark for different metadata normalization algorithms. With the unified interfaces in inferencer, to do so is straightforward as one only needs to specify the set of algorithms he/she wants to compare as well as the type(s) of data to ingest and designate a building for the comparison. We will demonstrate with concrete examples in Section 5.3.2.

5.2.3 Workflow

The standardized interfaces in inferencer also enable the creation of a workflow for metadata normalization. A workflow is a hybrid method comprised of multiple algorithms, each being an inferencer in Plaster, where the output of an inferencer is passed to another while each inferencer executes its inference procedure independently. While a single inferencer usually only infers one aspect of the metadata, a workflow would potentially be able to infer multiple or all the aspects of the metadata by employing different inferencers. Each inferencer may have a different learning objective as described in Section 5.1, and Plaster helps to systematically leverage the advantages of each. For example, Building Adapter [HWOW15a] (BA) is a transfer learning-based algorithm that infers point types without any human inputs, but usually with a potential low recall. Instead of starting from scratch, the output of BA can constitute an initial training set for Zodiac [BVNA15a] to jump-start its learning procedure and potentially reduce the amount of manual labels required. Various use cases of workflow enabled by Plaster are elaborated and evaluated in Section 5.3.3.

When executing, the workflow function call will invoke the corresponding functions (i.e., `select_examples()`, `train()`, and `predict()`) in each of its inferencers in the order specified in the workflow, with an additional connecting step that obtains and applies the previous inferencer's prediction results to the next. The process of applying a preceding inferencer's results vary across different inferencers so that a human integrator should specify how to digest such predictions inside the inferencer's methods. If the confidence of some of the inferred relationships by the previous inferencer are low, the next inferencer should filter the results or simply avoid using them. On the contrary, if the previous inferencer's inference is higher confidece than the current inferencer's, it can discard its own inference and adopt the previous one. In the example of connecting BA and Zodiac, Zodiac would need to be able to select only the prediction results with high confidence from BA and subsequently add them into its own training set inside `select_examples()`.

5.3 Evaluation

In this section, we demonstrate how Plaster enables systematic comparisons of different metadata normalization algorithms, the creation of new workflows by connecting multiple algorithms, and the programming interfaces for algorithm development such as feature selection.

5.3.1 Experimental Setup

Datasets

We obtain a subset of the study buildings used in Brick [BBF+16b], which consists of five buildings from four different campuses, including the raw metadata and timeseries data for about a month. Table 5.2 summarizes the details of each test building. While this collection of five buildings is not comprehensive for building metadata research, we argue that they are representative enough with regard to the diversity in vendors, sizes, years of construction, etc.

Table 5.2: Case Study Buildings Information: These are office buildings in universities. JCI and ALC stand for Johnson Controls and Automated Logic, respectively. The number of unique words represents the complexity of the metadata.

Building	Location	Vendor	Year	Size (ft^2)	# Points	# Point Types	# Unique Words
Engineering Building Unit 3B	UC San Diego, San Diego, CA	JCI	2004	150,000	4,594	108	426
Applied Physics and Mathematics	UC San Diego, San Diego, CA	JCI	2004	150,000	4,357	111	369
Rice Hall	Univ. of Virginia, Charlottesville, VA	Trane	2011	100,000	1,300	60	290
Sutardja Dai Hall	UC Berkeley, Berkeley, CA	JCI	2009	141,000	2,300	31	116
Gates Hillman Center	Carnegie Mellon Univ., Pittsburgh, PA	ALC	2009	217,000	8,292	147	179

For building D1, the original author did not release the timeseries data, and therefore, we shall note that D1 will not be included later in evaluations that involve timeseries data.

Evaluation Metrics

Overall, we consider three aspects when evaluating each algorithm:

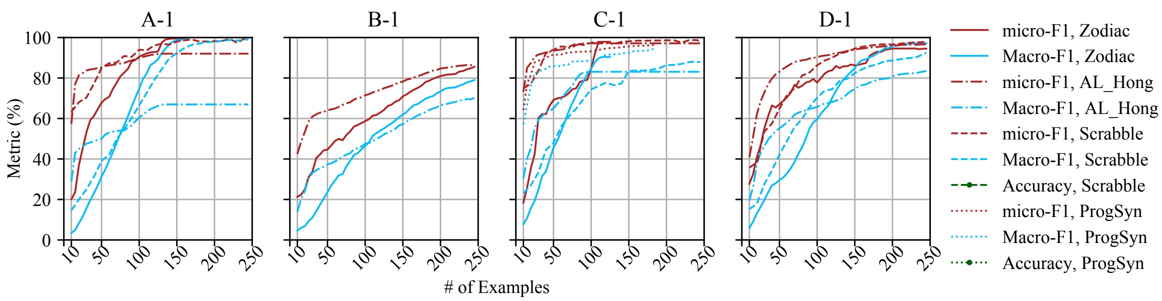
- *Inference Accuracy*: How accurate are the predictions of an algorithm in terms of its original learning purpose?
- *Inference Coverage*: What kinds of labels can an algorithm infer?
- *Human Efforts*: How many examples does an expert need to provide in the learning process of an algorithm?

In this study, each algorithm infers one or multiple kinds of labels for a point. For example, Zodiac [BVNA15a] infers only one kind of label, which is the point type, whereas Scrabble [KBS+18] also identifies other kinds of labels such as location aside from the point type. For each kind of label, every possible Brick tagset is treated as a class (e.g., for point type we have room temperature, supply air temperature, etc), and we evaluate the inference performance considering all kind(s) of labels each algorithm produces. To measure how accurate the inference results are for an algorithm, we calculate the Micro-averaged F1 (MicroF1), Macro-averaged F1 (MacroF1), and example-level accuracy. MicroF1 globally counts the total true positives, false negatives and false positives regardless of the class, while MacroF1 calculates the same quantities for each class and then finds their unweighted mean. MacroF1 indicates how many different classes can be correctly inferred, which is an important metric for a building dataset that typically has an (extremely) imbalanced class distribution, with the points related to heating and cooling clearly dominating. For example, while `Zone Temperature Sensor`s might frequently exist in HVAC systems, specialized points such as `Gas Meter`s are generally rare. For example-level accuracy, it is defined as the ratio of the number of correctly labeled examples over the total number of examples. Specifically, an example is considered to be correctly labeled if and only all of its labels are correctly predicted. We use this metric along with the F1 scores when an algorithm produces more than one kind of label.

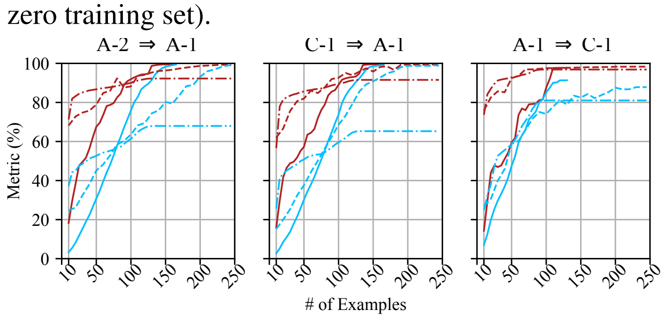
We measure human efforts by the number of examples labeled by an expert during the model learning process. For point type inference, an example is usually a mere point type label given the raw metadata of the point. For the examples used for inferring all possible entities, they contain more information aside from the point type label, such as equipment ID and location. Although the amount of information in the examples is different, we consider the effort for labeling an example to be the same because the required knowledge per example is similar.

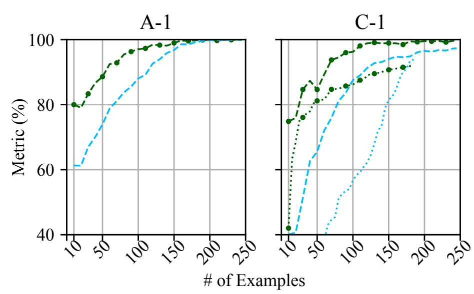
Inference Algorithms Included in Plaster

We have refactored and incorporated the following algorithms into Plaster: Hong active learning [HWW15] (referred to as AL_Hong hereafter), Bhattacharya et al. [BHC+15b] (referred to as ProgSyn), Zodiac [BVNA15a], Building Adapter [HWOW15a], and Scrabble [KBS+18]. We exclude algorithms from the evaluation that require the actual actuation and control in buildings [KBA+16, PBCM15b] because such experiments are not practical in most buildings. However, they fit into Plaster well as part of a workflow in the real world such as building commissioning. Plaster is open-sourced and implemented in Python. The API documentation, running examples, together with the data sets can be found at
https://github.com/plastering/plastering.



(a) Learning rate for inferring point type by different algorithms on 4 buildings starting from scratch (i.e., zero training set).





(b) Learning rate for inferring point type, exploiting an existing building's normalized metadata. X $\Rightarrow$ Y indicates applying X's normalized metadata to initialize the learning for Y.

(c) Learning rate for inferring all entities in the raw metadata from scratch.

Figure 5.2: Comparisons of Different Algorithms on Various Buildings

(atop each figure) The alphabet represents a campus and the number represents a building on that campus (e.g., A-1). We leave out Scrabble's results for B-1 and ProgSyn's results for A-1, B-1 and D-1 due to the limited types of labels in these buildings. All experiments are averaged over four runs and the legend is shared across all figures.

5.3.2 Benchmarking

Enabled by the unified interfaces in inferencer, Plaster allows a user to easily select a method and specify the type of input ingested, the test building to use, and the evaluation metric; this facilitates systematic comparisons of different algorithms, i.e., *benchmarking*. We present the results of three representative scenarios.

Active Learning for Point Type Inference

In this scenario, we evaluate a set of active-learning-based algorithms for their learning efficiency in inferring point types, the most important aspect of building metadata. We include two algorithms that exclusively work for this purpose – AL_Hong [HWW15] and Zodiac [BVNA15a], together with another two algorithms that can infer multiple aspects in metadata (type, location, equipment, etc) – ProgSyn [BHC+15b] and Scrabble [KBS+18]. Although the latter two are designed to learn all aspects of metadata, we make each to infer only the point type in this set of experiments. We run each algorithm on four different buildings, starting with zero training examples, and calculate the MicroF1 and MacroF1 of inferred type labels. The results are shown in Fig. 5.2a.

We see that AL_Hong marks a stark contrast to all the other algorithms for its steep learning rate (by MicroF1) in the early stage for the first 75 examples. This is because of its clustering-based example selection strategy, which excels in quickly selecting representative examples that are also informative for model training. However, we also see that Zodiac and Scrabble are able to catch up after 75 to 125 examples, surpassing in MacroF1, and even achieve 100% in F1 for some cases (on building A-1) after converging. These results suggest that Zodiac and Scrabble are better in learning the *minor* point types that appear less frequently in a building, which AL_Hong is not able to learn even with more examples. We would also like to point out that, due to the deterministic nature of the algorithm, ProgSyn and Zodiac may terminate early (e.g, on C-1). Zodiac runs with a preset confidence threshold, and as it gradually acquires training examples, whenever the algorithm has high enough confidence in every testing instance, it will terminate. For ProgSyn, it decides whether the learned rules are able to parse every example and stops when it becomes the case. Furthermore, there is no clear winner in this set of experiments. The implication, however, is that if one wants to quickly label the types with reasonably high accuracy (e.g., 85%), AL_Hong is an appropriate choice. When one desires better coverage of less frequent types in the long run, Zodiac or Scrabble would be a better choice.

Jump-started Active Learning

All the original active learning-based algorithms [BVNA15a, HWW15, KBS+18] are designed to work only within the same building, meaning that they do not consider or leverage any information from other existing buildings. However, because the inferencer design in Plaster makes it convenient to start from any training set, we will next show what the learning results would be if we run an active learning-based algorithm using information from another building for inferencer initialization. More specifically, we use another building's point names along with their labels (e.g., from A-1) to formulate the initial training set for an inferencer and then run the algorithm on another building (e.g., C-1) as we did in Section 5.3.2. The results are shown in Fig. 5.2b.

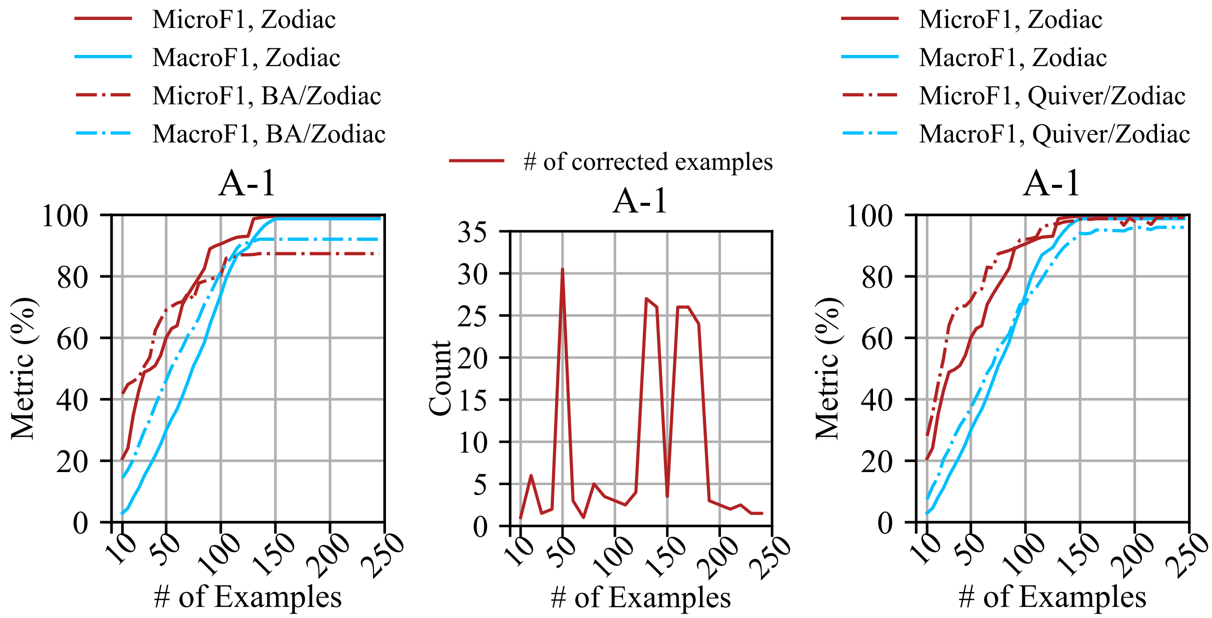
When added a building from a different vendor with almost completely distinct naming conventions (e.g., A-1 ⇒ C-1 and C-1 ⇒ A-1), the type inference performance either remains unchanged or even deteriorates in the early stage. This is expected as such transfer would introduce more irrelevant patterns to the same point type for the algorithm to learn, which is almost equivalent to injecting noise. Nonetheless, we still notice an increase in MicroF1 for Scrabble in the early stage in the case of A-1 ⇒ C-1. This is largely due to Scrabble's underlying intermediate representation, which is able to learn more general patterns with different buildings. On the other hand, when we add a building from the same vendor with a similar vocabulary for point types (see A-2 ⇒ A-1), we observe a better starting point (71% in the first figure in 5.2b vs 58% in the first figure in 5.2a) and also a better converged MicroF1 for AL_Hong. We observe similar improvements for Scrabble and Zodiac in this case. We thus conclude that having building(s) with a similar naming convention is useful for inferring subsequent buildings by transferring the information in the raw metadata.

Active Learning for Multiple Entities

While detecting the point type is important, other types of entities encoded in the metadata, such as the associated room and equipment, are also essential for building applications. We, therefore, evaluate Scrabble and ProgSyn for their ability to identify multiple types of entities from the given raw metadata, including the point type, room location, and associated equipment ID. As shown in Fig. 5.2c, Scrabble outperforms ProgSyn in both MacroF1 and example-level accuracy. The gain in performance of Scrabble is attributed to its more sophisticated representation learning procedure where it first maps the input to an intermediate representation and then to actual labels, while ProgSyn maps the raw metadata directly to final labels. For example, for a string `ZNT`, Scrabble first learns its nuanced character-level BIO tags and then maps to the Brick tagset (i.e.e, `Zone Temperature Sensor`), while ProgSyn directly learns its mapping rule to the tagset via regular expressions.

5.3.3 Workflow

Having seen the results on comparing different algorithms individually, we next show how they can interact with each other in Plaster. A key feature of Plaster is the ability to try out different workflows, which integrate different algorithms in a different order. We present and evaluate three exemplary workflows where two inferencers are connected together for better performance than if they are used individually.



(a) BA ⇒ Zodiac (b) Zodiac ⇒ Scrabble (c) Quiver ⇒ Zodiac

Figure 5.3: Learning Efficiency for Inferring Point Type by Different Workflows

They all demonstrate synergistic improvement in performance.

Transfer Learning Benefits Active Learning

A completely automated method such as Building Adapter (BA) [HWOW15a] is able to achieve relatively high inference precision for point types in a target building, though for only a fraction of the points. It would be natural to connect and feed the labeled examples by BA to an active learning-based method, such as Zodiac [BVNA15a], as a better starting point. This appears to be similar to the jump-started active learning scenario in Section 5.3.2, in that both provide a better starting point for active learning. However, a fundamental difference is that a method such as BA, which transfers the learned model via *timeseries data* from a different building to facilitate another learning process based on *textual data*, which is independent from these two buildings' naming conventions, while in the previous scenario we will only see benefits when transferring from a building with a similar naming convention. We implement such a workflow of combining BA and Zodiac to again infer point types, with Fig. 5.3a showing the comparison results. We see that the combination achieves both higher MicroF1 and MacroF1 up to 70 examples, benefiting from the transferred information. However, the incorrect labels from BA's predictions (though only a handful) remain as negative training examples to Zodiac and it cannot recover from such noise in such a naïve integration. These inherited incorrect labels would be corrected or filtered out at the beginning if Zodiac could have the ability to quantify BA's results based on its own criterion. Yet, this will require additional modifications to the original algorithm and is hence out of the scope of Plaster.

Specialty Complements Versatility

Some algorithms have high precision while others have high recall in their inference results. For example, Zodiac infers only point types but with high precision, while Scrabble can identify multiple kinds of entities with high recall. Thus we can filter Scrabble's results by using Zodiac's results without compromising the results of either. More specifically, we feed Zodiac's results to Scrabble's prediction and if there is a disparity between the two on an instance, Scrabble will adopt Zodiac's prediction for point type. As shown in Fig. 5.3b, we see there are about 1,500 corrections made to the point type predictions in total (note that we only count the number of corrections made by this strategy, and an instance could be corrected multiple times) with little additional computational cost.

Mutual Benefits between Different Types of Inference

Learning functional relationships often relies on perturbations to the control systems (e.g., Quiver [KBA$^+$16]) and, to correctly perform perturbations on a target point such as a VAV on/off command, it requires knowing the point types apriori. Thus, it is natural to apply an active learning algorithm (Zodiac) to infer point types as a prior step to a perturbation-based relationship inference algorithm (Quiver). Furthermore, the inferred relationships can in return help examine whether the point types have been correctly inferred. For example, the fact that a VAV typically contains only one for each type of its sensing and control points can help identify mistakes made in type inference. Concretely, based on the manual perturbation to a VAV on/off command, Quiver [KBA+16] identifies a group of co-located points and finds that there are two supply air temperature sensors; it is highly likely that Zodiac has made a mistake in the type inference. For this experiment, we emulate the above procedure by first running Zodiac to infer point types, and for each predicted VAV on/off command, we use the ground-truth for the co-located points in that VAV (since we are not able to actually run Quiver) and examine if there is any duplicate type among these points, in order to correct any type mis-predictions. We see modest improvement in the type inference results because we only consider ~15 most common point types existing in VAVs, as Quiver can only find the co-located points for VAVs. Although a workflow as such exploits certain domain knowledge, it would be generally useful to practitioners with special demands in building applications.

5.3.4 Timeseries Feature Selection

We also empirically inspect how well each timeseries feature set performs and how effective the feature selection is in Plaster. To this end, we create a workflow that feeds the timeseries data to each of the feature extraction modules included in Plaster, passes the features to a random forest classifier (which is identified as the best performing classifier [GB18]), and predicts the point type for evaluation. Fig. 5.4 summarizes our results.

We observe that each individual feature set roughly performs on par except the second set. A simple fusion of all the dimensions from each feature set (marked as All in the figure), which equates to a 106-dimensional feature set, does not yield much better performance. However, the selected set of features does give a 3% increase overall than the best set with the number of features reduced to 60. This demonstrates the usefulness of the feature selection and integration provided by Plaster.

We notice the performance here by using timeseries data features is less competitive than

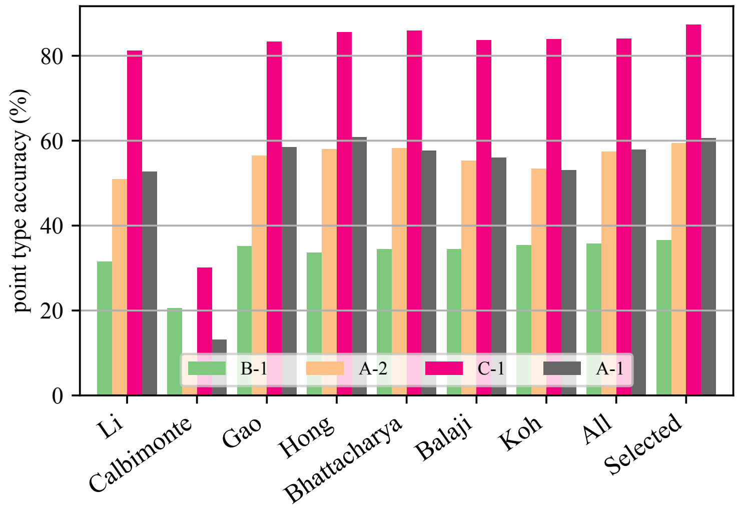


Figure 5.4: Results for timeseries data based type inference

We show 7 different feature sets (each is shown as a group of bars and each bar represents the result on a building), along with a fusion of them (All) and a better subset selected by Plaster (Selected).

using textual metadata. The algorithms using textual metadata in Fig. 5.2) can achieve more more than 95% accuracy with labels of less than 5% of the entire building. However, the implication is that, as data features better suit transfer-learning-based tasks [HWOW15a], the better feature set we have identified here would help to improve such a procedure, for instance, Building Adapter. Moreover, we would also like to emphasize that subsequent users and/or researchers can easily register their own feature set in the feature extractor interface in Plaster, and also perform feature selection with our provided method to obtain an even better set of features for their target metadata normalization problem.

5.3.5 Programming APIs and Examples

We next showcase a simple code snippet on how to evaluate an inferencer in Plaster following the unified interface design. We see from the example that one only needs to specify an algorithm along with some configurations including the buildings involved for evaluation and parameters to the algorithm. We shall note that for more running examples on benchmarking,

```python
target_building, source_buildings = 'ap_m', ['ebu3b']
SAMPLE_SIZE = 100
srcids = LabeledMetadata.objects(building=target_building).distinct('srcid')
training_srcids = random.sample(srcids, SAMPLE_SIZE)
test_srcids = [srcid for srcid in srcids if srcid not in training_srcids]
scrabble = ScrabbleInterface(target_building, srcids, source_buildings)
scrabble.update_model(training_srcids)
metrics = scrabble.evaluate(test_srcids)
```

Figure 5.5: Example for Evaluating an Inferencer in Python

workflow, etc, one can refer to our documentation[6] for details.

5.4 Plaster User Interface

For all the components in the process, Plaster [KHG+18] defines a common programming interface in Python as well as a unified database model for all the necessary data types. It enables the benchmark and integration of different algorithms and eases the process to develop a new algorithm. However, due to the diversity of the algorithms, end-users, such as building managers and commissioners, need better guidance in actual user interfaces.

As discussed in Section 5.2.2, the canonical functions in Plaster are `insert_examples`, `select_examples`, `update_model`, and `infer`. These functions are used throughout Plaster UI's workflow, whose steps are following:

1. *Task Configuration*: A user chooses an algorithm or a combination of algorithms based on their optimization targets. The user also needs to load data.

2. *Interactive Labeling*: A user provides examples for metadata normalization. Algorithms can choose the most informative examples with non-redundant patterns. It improves the sample efficiency reducing human effort eventually. This is an iterative process that involves training models, requesting new examples, and submitting labels.

[6]https://github.com/plastering/plastering/blob/master/examples

3. *Result Review*: Once enough examples are given in the previous stage, the user can visually review the result of metadata normalization in a graph or a table. The user can also export the result to use it in external systems.

In the workflow, Interactive Labeling is the most complicated process as users need to iteratively interpret visualized information and provide corresponding labels. Fig. 5.6 shows the visualization, and the caption describes each component in detail. While we provide the flexibility to revoke the actions, we visually guide users to only the required actions among many possibilities. For example, in Fig. 5.6, Plaster has provided the entity's type, so deactivate the insertion action by showing "Inserted", and then activate "Next".

5.5 Discussion

Plaster provides a common programming model for different algorithms in metadata normalization for buildings. The ultimate goal of metadata normalization research would be to establish a grand model that can utilize any kind of data and labels for any purpose. We discuss several algorithmic challenges for the goal.

A Robust Framework for Integrating Different Methods

Different algorithms are methodologically independent of each other and interpretation of their results is solely up to the users. For example, assume Zodiac [BVNA15b] infers an entity as `Temperature_Sensor` with 90% confidence and Building Adapter (BA) [HWOW15b] infers the same point as `Temporary_Occupancy_Status` with 80% confidence. We cannot judge which one we should trust more only with the results inferred independently. An experiment where we integrate BA and Zodiac confirms the observation as in Figure 5.3a. While Zodiac receives benefits from BA's inference in the early stage, Zodiac's sample selection algorithm suffers from BA's false predictions in the later stage. Even though we could improve the results by putting

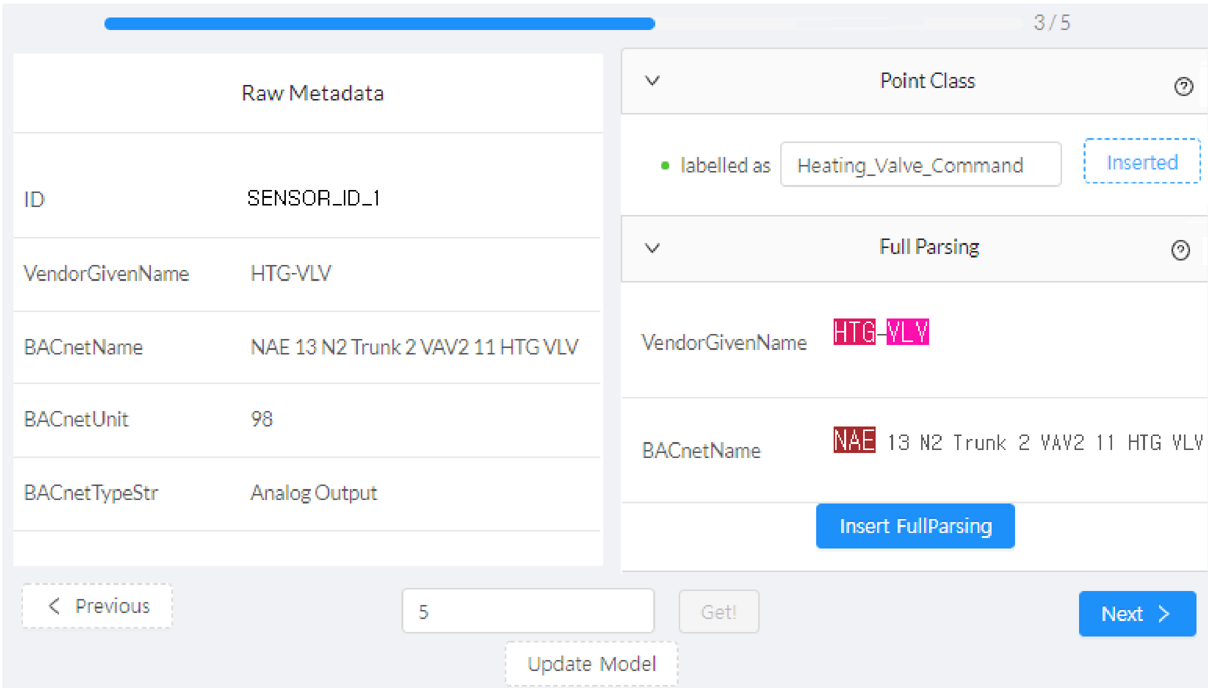


Figure 5.6: A Screenshot of Plaster UI.

The bar and the ratio at the top of the screen show the progress of the current round. The left panel visualizes different types of Raw Metadata. At the right panel, users can interactively provide different types of labels. Users may select a customized number of examples to label in a round with Get button, and move to different examples by Next/Previous buttons within the round. Update Model trains the machine learning model based on the training examples labeled so far. Blue buttons are the guided actions for the user, showing recommended next steps. White buttons indicate the user have completed the actions but the user may revoke them. Gray buttons are for inactivated actions at the current status.

some threshold to filter out BA's less confident predictions, the filter's design is still based on personal interpretation of the results.

We should, in the future, introduce a statistically robust framework such as weakly-supervised learning [RBE$^+$17]. Weakly supervised learning frameworks learn dependencies across different classifiers, assuming each classifier is imperfect, and learn a meta-model accommodating them. Within the framework, we could generalize a collection of different algorithms with the same goal as a higher-order algorithm. Still, dependencies across classifiers require an extensive data set to be robust per label, while the datasets we have observed so far do not have enough samples to be fed.

Noisy Labels

All the algorithms studied for metadata normalization assume that the labels are correct, which is often not true. Metadata normalization tasks would be commonly done by multiple domain experts, who could either make errors or have different understandings of the same content. It becomes especially important when we wish to accumulate knowledge over many buildings labeled by different people and institutions.

There are various frameworks for learning from noisy labels [NDRT13, WSL+19]. They propose diverse methods to reweigh samples by learning surrogate functions that together explain the credibilities of samples. Otherwise, we could also reuse the weakly-supervised learning frameworks with training a model with a data set from a single annotator. Though it would not distinguish incorrect labels within a model, the weakly-supervised framework would be able to find an agreement on labeling from different data sources.

Privacy Preserving Transfer Learning

Metadata normalization for buildings has been studied in many places, but the efforts are fragmented so far. One of the reasons is that people are hesitant to sharing their data publicly because the data may contain both security and privacy information. Timeseries data often have information about occupancy of buildings through temperature changes, energy usage, and just occupancy data. Raw metadata often contain information about the network architecture and system configuration Though such information cannot be directly used by attackers without compromising the building network, building managers are not likely to take risks at the cost of building compromise. Plaster's future deployment should consider either filtering out sensitive information from the beginning [FGC19] or developing a machine learning model robust to differential privacy attacks [PMSW16].

5.6 Summary

Plaster has provided a standardized programming model for metadata normalization as well as a Web interface for users to interact with data and algorithms. Through the standardization, we can 1) benchmark different algorithms with the same datasets and metrics, 2) integrate different algorithms to complement each other, and 3) ease the development process by providing essential components in hand. Primarily, we can compare different algorithms and show the validity of each algorithm in different cases. Our results reveal that each method has its pros and cons. Users should choose an algorithm according to their requirements and combine different methods complementarily to yield better results.

Plaster web service will also be the foundation for safely collecting data and collaborating across different institutions for metadata normalization. Without a collaborative effort, all the research in this problem will remain fragmented for specific buildings and needs. Future research should help to bridge between different algorithms and data sets to accumulate knowledge over many buildings.

Moreover, normalizing metadata is the first method needed to enable programming over structured metadata in buildings. Along with various algorithms including the proposed algorithms in Chapter 4, Plaster unifies the significant portion of the workflow to establish Brick-enabled building application platforms.

5.7 Acknowledgment

Chapter 5, in part, is a reprint of the material as it appears in Proceedings of the 5th ACM International Conference on Systems for Built Environments, 2018 by authors Jason Koh, Dezhi Hong, Rajesh Gupta, Kamin Whitehouse, Hongning Wang and Yuvraj Agarwal. The dissertation author was the primary investigator and author of this paper.

Chapter 6

An Access Control Model with Structured Metadata

We have presented Brick schema, an extensible metadata schema for portable building applications (apps), as well as the methods for instantiating Brick from unstructured data sources. They bridge the gap between heterogeneous Building Management Systems (BMSs) and apps agnostic to such systems via a standardized view over the BMS. Using BRICK, we envision developers can quickly implement and deploy building apps, akin to what we have in the smartphone ecosystem. Under this standardized ecosystem, apps will be more accessible; developers can work on a standard development environment of building apps targeting a scalable deployment. The end users of these apps (e.g., building managers and owners) would quickly install and compare different apps for their exact needs without demanding much engineering cost for customizing apps individually. The Apps can be registered in a marketplace, similar to the Google Play store, and building managers can download it into their building systems, and end-users could simply use the app. Genie [BKWA16] is an exemplary app that has been deployed at Engineering Building Unit 3B, UCSD for over five years. Genie is a Web service where users can remotely control their own rooms' temperature in commercial buildings. Ideally, a building manager could choose Genie from an app market and install it for a target building, and then tenants of the building could remotely access their rooms through Genie without further hassles.

However, in the deployment process, the building manager should be able to understand what types of resources Genie would access and be assured of Genie accessing only the required resources and is not over-priviledged. This enforcement mechanism is called *access control*, and it is crucial to limit access only to the necessary resources, restraining the damage from malicious attacks or inadvertant misconfigurations to its minimum. It is a well-known security design called *the Principle of Least Privilege* (PoLP) [SS75]. As a baseline compared with buildings, Android has an access control workflow [Goob] as follows:

1. the user installs the app,
2. the app shows the manifest and asks its permissions to the user,
3. the user reviews the manifest listing all the possible access to resources in her phone such as contacts and the camera,
4. the user approves the app's access either at runtime or install-time,
5. the app's permission is stored internally and referred to it whenever the app tries to access the corresponding resources.

Basically, a user can simply review what an app can access and delegate her exact authority to the app. Such a simple review-and-delegate process works well for smartphone systems because a user is the owner of the target resources and, at the same time, the only end-user of the app.

However, the buildings' app workflow is significantly different from smartphones', for which the review-and-delegate process is insufficient. First, the access requirements are too complex and broad for a single person to review easily. For example, Genie, the Web thermostat, needs to possibly access all the rooms for reading two sensors and two setpoints for each of the rooms, and there can be hundreds of rooms in a medium-sized building. It is cumbersome for a single manager to review all the resources and their properties one by one. Second, buildings are a multi-tenancy platform where multiple users would use a single app to access different resources, and each of the users just needs specific access temporarily. Thus, it is over-privileged to delegate the superset of all the potentially necessary permissions to a single app. For example, while there are hundreds of users that possibly use Genie to control their rooms, only a subset of the users use Genie per day. Thus, Genie does not need a continuous access to all the users' rooms, whereas Genie is authorized for all of them in the review-and-delegate workflow. Third, the resource owners and the end-users are different. Unlike a user, who can manage an app's permission in her smartphone, the entity approving an app should be aware of all the potential users of the app in the authorization process.

In this chapter, to harness the access control of building apps, we present a systematic workflow for authorizing apps in buildings that can guarantee the least privilege to the apps. We first summarize our study from 125 building app papers from two major venues to understand the access requirements of building apps [KHN+19]. Based on the study, we define *access patterns* with Brick augmented with other data sources as an information model for access control. To tightly enforce the access patterns, we also propose a dynamic authorization workflow where actual authority is evaluated at runtime and distributed across apps and end-users so that the minimal access is guaranteed per app and user. At the same time, the approver can simply review an access pattern instead of the whole list of resources. Furthermore, we demonstrate this workflow by implementing two apps, a Web thermostat and an energy dashboard, with the workflow.

6.1 Building Operating Systems and Applications

In this section, we briefly introduce a general form of Building Operating Systems (BOSes) and how they host apps. Unlike Building Management Systems designed for human operators, BOSes are for programs. BOSes provide Application Programming Interfaces (APIs) and related administrative functions, with which programs or apps can interact with the underlying resources. Example BOSes include industrial solutions such as NiagaraAX [nia] and open-source projects such as XBOS [AKC+17] and BuildingDepot3 [bd319].

These BOSes have common components as APIs, drivers, metadata databases, timeseries databases, resource managers, and authentication/authorization mechanism, depicted in Figure 6.1. *APIs* are the interface for apps to request required operations in a BOS, such as to get or update timeseries data and register entities. BOSes are an interface over actual resources as well, including sensors, setpoints, equipment, and even existing systems such as BACnet devices. *Drivers* integrate such resources with a BOS by continuously logging resources' status and deliver the BOS's requests to end-devices such as adjusting a temperature setpoint. BOSes would store actual timeseries data in a *timeseries database* so that they can be referred to based on apps' logic later. Metadata databases contain all the metadata about resources often in a structured format such as Brick. As we have discussed, metadata is the key information for identifying the right resources for either reading timeseries data or actuating devices. Thus, a metadata database often stores the pointers to the corresponding timeseries data streams in the timeseries database as well. BOSes are usually multi-tenancy platform, and multiple apps could try to execute the same operation at the same time. A *resource manager* is in charge of resolving such conflicts, for example, based on priorities. Lastly, the *authentication/authorization module* governs identifying the actor of a request and whether the request should be executed or not. There are various authorization mechanisms such as Access Control List (ACL) [IET07], the most common model as adopted in NiagaraAX, and Attributed-Based Access Control (ABAC) [HFK+13] in BuildingDepot3 [bd319].

In the context of buildings, *resources* refer to the entities essential to building apps [BBF+16b], including all the physical points (e.g., sensors), equipment (e.g., HVAC), the data generated by these points, and the physical space (e.g., offices). We assume all the resources are

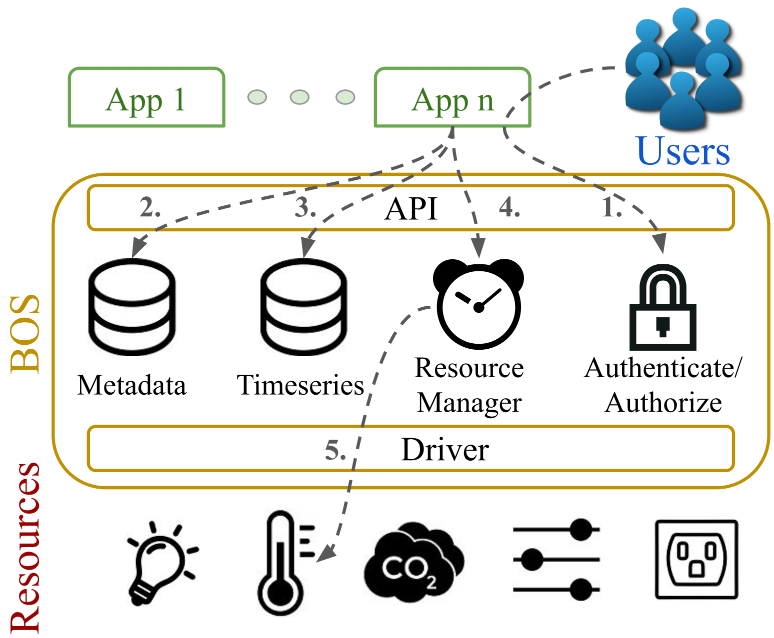


Figure 6.1: An Overview of Application Workflow with Building Operating Systems

managed through a BOS, which provides programming interfaces for apps to interact with the resources in the building.We thus regard the BOS as a trusted information source so that we can augment its security measures on demand.

An app can manifest the necessary resources, and the *owners* should approve the app's access. As a building is typically managed by a building manager on behalf of the owner, (s)he has the authority to arbitrate actions over the resources. However, a third-party company may also deploy and manage additional resources in a building, and could directly manage these resources or delegate the control to the building manager. In other words, multiple resource owners might need to work together to decide on the access policy.

The various *users* (e.g., occupants and building managers) would inevitably have different demands, and thus should be granted different permissions. For example, an occupant should be able to change the temperature in her office but not her colleagues'. Since different apps and the users need to access different resources, and there could be hundreds of apps with thousands of users. To manually create and maintain access control lists at this scale would require much effort from the corresponding resource owners and quickly become unwieldy and hard to reason about. Thus, the capability to rigorously represent and check what users can do with apps is crucial to scaling access control for building apps.

6.2 Exact Access Requirements of Building Applications

Traditional access control patterns fail for two main reasons for buildings. Access Control Lists (ACL) define fine-grained permissions to control what actions each user of an app can take on each object. Using ACLs, a resource manager would need to manually approve the access per user per app, which quickly becomes untenable. Grouping resources and users may provide more descriptive patterns for access control. However, existing patterns, such as attributes [HFK+13] and roles [San98], are not specific enough to represent only the exact resources and the access conditions in buildings. Access control patterns for building apps should be specific enough for resource managers to balance expressiveness, understanding, and scalability.

6.2.1 Analysis Setup

To holistically understand different access patterns, we have reviewed 125 papers published at BuildSys[1] from 2009 to 2018 and e-Energy[2] from 2012 to 2018. The authors of these papers have diverse industry and academic backgrounds, use heterogeneous testbeds ranging from residential to office buildings, and cover an extensive range of building apps. While Balaji et al. identified eight app categories [BBF+16b, BBF+18b], we expand them to 20 categories under six high-level domains[3]. The first and the second columns of Table 6.1 list the app domains and categories, respectively. The authors reviewed the papers and at least two authors cross-validated

[1] ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation.

[2] ACM International Conference on Future Energy Systems.

[3] The venues lack some app categories such as building security and healthcare though our analysis applies to undiscovered categories in a similar manner.

Table 6.1: Resource Access Patterns across Different App Categories.

An app may access only the resources meeting all the marked conditions. “rsrc” stands for resource and “↔” for a relation between the two classes. Brick [BBF+16b, BBF+18b] comprehensively models “rsrc type” and “rsrc↔rsrc” but not the rest of the table.

			Who					What					When					
App Domain	App Category	#Apps	occupant	building manager	other apps	energy provider	anybody	rsrc type	rsrc↔ rsrc	user↔ space	user↔ device	user↔ preference	user location	rsrc state	calendar schedule	dr event	onetime access	user request
Maintenance	FDD	6	◔	●	○	○	○	●	●	○	○	○	○	○	○	○	◑	○
	Space Management	2	○	●	◕	○	○	●	●	◕	○	○	○	○	◕	○	◕	○
Building Modeling	Environment Model	2	○	●	○	○	○	●	●	○	○	○	○	○	○	○	○	○
	Structural Model	1	○	●	●	○	○	●	●	○	○	○	○	○	○	○	●	○
Energy Analysis	Energy Disaggregation	20	◕	●	◕	◔	○	●	◕	◔	◔	○	◔	◔	○	○	◑	○
	Appliance Identification	1	○	●	●	○	○	●	●	○	○	○	○	○	○	○	●	○
	Thermal Comfort Model	7	◕	◑	◔	○	○	●	◕	◑	○	◔	◑	○	○	○	◔	◔
	Energy Model	8	◔	●	●	◕	○	●	●	○	◔	○	○	○	○	○	◕	○
	Energy Footprinting	5	●	●	◕	○	○	●	◑	◑	◔	○	○	○	○	○	○	○
Efficient Control	Model-Predictive Control	11	◔	●	◔	○	○	●	●	○	○	○	○	○	○	◔	◔	○
	Occupancy-Based Control	21	◑	●	◔	◔	○	●	◕	◑	◔	◔	◔	◔	◔	○	◔	○
	Demand Response	20	◑	●	○	◕	○	●	◕	○	◔	◔	○	○	◔	◕	◔	○
Occupancy Modeling	Occupancy Detection	13	◔	◕	●	○	○	●	◕	◔	◔	○	○	○	◔	○	◑	○
	Occupancy Identification	4	◕	●	◕	○	○	●	◕	◑	○	○	○	○	○	○	◑	○
	Activity Recognition	3	◕	●	◕	◑	○	●	●	◑	◕	○	◑	○	○	○	◑	○
	Behavior Modeling	2	○	●	●	○	○	●	◕	○	●	○	○	○	○	○	○	○
User Interface	Web Displays	2	●	◕	○	○	◕	●	◕	○	○	○	○	○	○	○	○	○
	Remote Controller	2	●	○	○	○	●	●	●	○	○	○	○	○	○	○	○	●
	Participatory Sensing	5	●	◔	○	○	○	●	◕	◑	○	○	◑	○	○	○	◔	◔

●: #apps > 75% ◕: 75% ≥ #apps > 50% ◑: 50% ≥ #apps > 25% ◔: 25% ≥ #apps > 0% ○: no apps

the results of each paper. The full analysis is available online[4], and we cite app papers with their

[4] https://tinyurl.com/building-apps-access-control

row number in this document as (R#). We identify three information dimensions that should be examined for building apps to access the required resources:

- **Who** may use this app?
- **What** are the resources this app can possibly access?
- **When**, or in which context, can a user use this app?

Table 6.1 shows the access patterns required for each app category, based on the three dimensions. They are complete in expressing the representative apps studied in this paper and we expect them to generalize to other building apps.

6.2.2 Resource Access Patterns

Who: User Type

We mainly identify five types of users within buildings. **Occupants** typically use apps to control their environment (e.g. manage temperature) or understand their behaviors (e.g., energy usage). **Building Managers** oversee the operation of building with regard to space management, equipment maintenance, energy efficiency, maintaining security, etc. They may use most of the apps except the apps that primarily produce indirect results such as Energy Models. Some apps are designed to be used by **Other Apps**: Analytical apps, such as a prediction model of an electrical device's energy consumption, may feed their output to other operative apps such as demand response. **Energy Providers** are a unique type of users not residing in target buildings. They collect information about a building's energy usage and use that to control the building equipment either directly or indirectly through utility pricing or demand response events, in order to stabilize the electrical grids. Some apps are agnostic to user types, e.g., a public energy dashboard (R9) or a location-based controller (R60) , which **Anybody** can use.

What: Resource Identification

Resource Types: Resource type is the most important class of information, as any building app must access some resource(s), whether it be a temperature sensor, a light bulb, or an office. The resource type is also critical for reasoning about security since different resource types have different capabilities, with different consequences if breached. For example, upon a security breach, a motion sensor may leak private occupancy information while an airflow setpoint could physically damage the controlled equipment.

Resource Relations: Relations between resources connote their relative functionalities, enabling precise resource identification. For example, the causal control dependence between points are critical for Fault Detection and Diagnostics (FDD) (R33) ; when the supply airflow of a VAV is anomalous, the corresponding actuator in the same VAV is required for analysis. Except for apps that need to access all the resources of some type (e.g., a tool visualizing all the building energy meters (R9)), resource relations are a crucial information dimension for describing apps' requirements.

Resource-User Relations: Resources serve, monitor, and/or are controlled by users, and thus expressing these relationships is a key. We identify three kinds of such relations. First, in buildings, **space** is often assigned to a person whether it is an office, a lab, or a desk. Consequently, a sensor in an assigned space reveals information about the person, e.g., presence or schedule. Thus, space-related apps, such as occupancy detection, need to get approval from the associated person who is being monitored when accessing these resources. Note that while theoretically everyone needs to approve an app's access to their data, this process may be delegated to someone like a building manager. Delegation is out of the scope of this paper though it is complementary to any access conditions. Second, people use personal or allocated **devices** to customize the indoor environment such as lighting (R60) . It is thus necessary to consider *what devices each user has*. Lastly, user **preferences** are frequently used in apps that improve the occupants' comfort and productivity. We discover that multiple apps need user preferences over conditions such as lighting and temperature to control the environment for the users based on their locations (R43).

When: Access Context

Resource access may be temporally granted based on the user/app context, so as to prevent overprivileged apps with constant access.

User Location: As a user is physically present in a particular space, apps related to occupants can refer to the user's location. For example, an app for occupancy-based HVAC control (R72) is currently granted access to all the VAVs in an entire building all the time, whereas it should access only some VAV when the user is nearby.

Resource State: An app may need to be active based on resources' states. For example, a lighting controller should be active only when the associated room is occupied. Note that, the difference between *user location* and *resource state* is that the user's location is specific to a target user while a resource's state is occupant agnostic.

Schedule: Temporal bounds may be explicitly defined, whether they are regular schedules or temporal bookings.

Demand Response: Demand Response (DR) events rarely occur – usually several times per year even for program participants. Only when a DR event happens should automated DR apps be active, which have a powerful capability to control the entire buildings.

One-time Access: Data-driven apps need to access historical data for training, but once they train their models, the data should not be accessible and only the trained models should be used.

User Request: Apps such as remote controllers convey users' intention to control the system. An app's request should be valid only when it can be verified to be from the actual user.

Table 6.2: Access Control Patterns Supported in Existing Smart Building and IoT Platforms

System	Who	What	When
Wave [AKA+19]	Individuals	Predefined groups	No
BuildingDepot3 [bd319]	Custom groups	Tag-based groups	No
NiagaraAX [nia]	Individuals & Roles	Each object	No
Cloud IoTs [aws19]	Individuals & Roles	Each object	No
ESO [SST18a]	Individuals	Each object	Yes

6.2.3 An Example for Access Pattern Evaluation

To evaluate and approve an access request from an app, a BOS needs to first identify the app and user, i.e., authentication, and then check its *access pattern* — whether the request satisfies the information dimensions (i.e., columns defined in Table 6.1), including the user's role, whether the user is trying to access permitted resource(s), whether the user's demand has expired, etc. For example, an HVAC remote control app can be authorized only when an *occupant* (user type) is trying to control the *temperature setpoint* (resource type) *of the terminal unit* (resource-resource relations) in *his/her office* (resource-user relation) when (s)he *sends a request* (user request event).

However, we shall note that, as each app may uniquely describe its required resources and the relations among them, i.e., resource group, the request approval process thus involves interpreting a potentially tremendous set of combinations. Since it is impossible to exhaustively predefine static resource groups in BOSes to cover all the possible patterns, and rather, BOSes should be able to verify different information at runtime.

6.2.4 Existing Access Control Platforms

BOSes (and IoT platforms) use different access control models for evaluating information sources, as summarized in Table 6.2.

General IoT cloud services (e.g., AWS IoT) and commercial BOSes (e.g., NiagaraAX [nia]) support only ACLs and roles [San98] for authorization.An ACL is a list of permissions for various users on a single resource and each resource maintains its unique ACL. Thus, although ACLs and user roles alone could provide stringent access controls, even in one building, a system manager would have to laboriously maintain an immense number of ACLs for tens of apps and hundreds of users. To overcome ACLs' limited expressibility, many access control patterns have been proposed for multi-tenant cloud platforms [NDdL16]. However, these models are often too specialized for computational resources, lacking interleaved relationships, and the context barely changes over time while the users' behaviors and built environments change over time.

Wave [AKA+19] and BuildingDepot3 (BD3) [bd319] support grouping resources for authorizing apps. Wave allows delegating the authorization of a group of resources to another entity, but does not specify the definition of groups, which could be based on the hierarchy of location or equipment. In addition, apps need different groupings to follow the principle of least privilege, as they may need different resources in the same group (e.g., on the same floor.) For example, assuming an instance of the HVAC remote control app implemented with Wave that groups the resources based on rooms, it will be allowed to take any actions on all the data points (commonly ~ 15) in a room, whereas it only needs to read the temperature sensor and change the temperature setpoint. Such grouping unnecessarily exposes more resources to the app and the ability to control safety-related points such as a minimum airflow setpoint.

BD3 [bd319] allows flexible grouping over resources based on tags. For example, a group may consist of all the sensors with the "temperature" tag. BD3's grouping is more flexible than Wave's as a resource may belong to multiple groups. Still, the groups are manually defined and might not cover all the possible patterns.

Furthermore, Wave and BD3 do not employ contextual information such as a "user request" event. An occupant might use the HVAC remote controller infrequently, e.g., when she is working outside the regular schedule or during atypical weather. Thus, the app does not have to be granted access for the user all the time. Instead, a BOS should incorporate a temporal authorization, such as to set an expiration date or to schedule timed activation, while tracking the relevant events from trusted sources. Environmental Situation Oracles (ESOs) [SST18a] perform access control based on events that can be generalized into `When`-type information. It is necessary to incorporate dynamic events with metadata from different sources for the most general and expressive access control patterns.

Still, while buildings are a multi-tenant platform, none of the aforementioned systems are designed for apps serving multiple users, but rather they consider an app as a standalone entity. When the HVAC remote controller is implemented with these systems, the app would have permissions for the superset of all the possible users' all the time, thus being unnecessarily overprivileged. In other words, the app would be able to access hundreds of the terminals units in the entire building all the time, while it should only access the rooms' terminal units that users actually request.

6.3 Access Control Patterns

We have shown that `Who`, `What`, and `When` are the three authoritative information dimensions to tightly describe all the apps' access patterns, which none of the existing BOSes represent them comprehensively. For the secure adoption of smart building apps at scale, we identify several design considerations for access control:

- Instead of preconfiguring resource groups, BOSes should adopt flexible groupings over the complete view of building systems due to the heterogeneous apps' access patterns. (`What`)
- BOSes should have a way to join a building's metadata with the users', and users should be a part of the system modeling process. (`Who` and `What`)
- The metadata of buildings and users should be rigorously maintained. Access control patterns would rely on the metadata, applied to all the resources. Automated verification processes for metadata correctness is also desirable.

- Resources' context should be easily verifiable and it should be describable inside access patterns from apps (`When`).
- An app's access capability should be dynamically determined based on its users and the context at runtime, instead of subsuming all the potential access requests.

Based on the above considerations, we introduce *Access Control Patterns* (ACPs) where an app's access behavior is represented through a pattern consisting of queries, instead of a static permission list such as ACL. With ACP, resource owners do not have to evaluate all the resources one by one manually. Instead, they need to interpret ACPs and put trust in the related data sources. Such data sources may include actual people governing metadata, such as Human Resource (HR) managers in charge of assigning offices to their employees, or sub-systems producing data, such as a localization system.

The basic model to represent `What` is Brick that can precisely represent entities via their types and relationships with each other. Besides, we add `Person` Class to Brick to represent any human actors with buildings. People's relationships with resources are a key signifier for determining apps' permission over resources. For instance, a tenant who has an office would be able to access the temperature information of the office. `Person` types are also important as identified in Section 6.2.2. For example, a building manager would likely be able to access equipment status throughout the building. As a starting point, we introduce a few concepts including person types (e.g., `Tenant` and `Building_Manager`) and their relationships with other Classes (e.g., `manages` and `hasOffice`) as visualized in Figure 6.2.

ACP should also include `When` type information, which is often represented by actual data of resources such as timeseries data and structural information. Timeseries data represents a device's status or a person's location, and a person's location could be exploited together with structural information, such as the geometry of the building, to identify proximity between a person and a particular space (e.g., whether the person is currently in a specific room or not.)

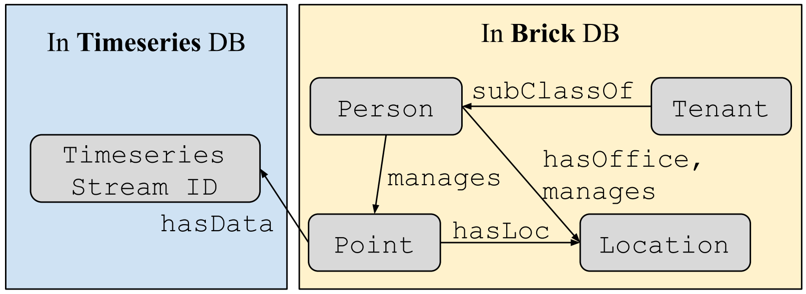


Figure 6.2: The Information Model for Access Control Patterns

It means that the combination of multiple databases represents access patterns, as described in Figure 6.2. Points are associated with timeseries streams in a timeseries database, and structure information of locations is stored in a structure database, for example, in the format of geometries[5].

Thus, we define ACPs as a set of federated queries to accommodate the different data models altogether. Figure 6.3a shows an example federation of heterogeneous queries. The ACP consists of two data sources as Brick/SPARQL for metadata and PostgreSQL[6]. SPARQL data model is explained in depth in Chapter 3. The timeseries data table in PostgreSQL, as in Figure 6.3b, has columns of UUID, time, and values that could be either number or text[7]. We can join the results from a set of queries sequentially over the shared variables. The queries together can represent any combination of `Who`, `What`, and `When`, and they are comprehensible to resource managers, especially when the queries are interpreted as a natural language. Figure 6.3c gives an example interpretation of the federated query in a human language, which can be either manually verified in advance or automatically translated [MLL99, NNBU$^+$13]. Further consolidation and optimization of query processing remain to be future work while there is existing work [EKB17, SHH$^+$11, DES$^+$15].

[5]We use PostGIS [pos] as an experimental implementation.
[6]More precisely, we use TimescaleDB extension in PostgreSQL. TimescaleDB: https://www.timescale.com/
[7]The value types could be extended further in the future.

```
queries:
  - - brick
    - >
      select ?user ?user_loc ?znt ?room where {
        ?user a brick:Tenant.
        ?znt a brick:Zone_Air_Temperature_Sensor.
        ?znt brick:hasLocation ?room.
        ?room a brick:Room.
        ?user brick:hasPoint ?user_loc.
        ?user_loc a brick:Person_Location.
      }
    - timeseries
    - >
      SELECT uuid = ?user_loc AND text = ?zone
        AND time > now() - interval '5 minute';
shared_variables:
  - ?user_loc
  - ?zone
```

(a) An Example Query Federation in YAML

This example query merges two different queries with Brick and timeseries data. The result of the first query, specified by `shared_variables`, is embedded into the next query. For access control, if the result of the merged query is not empty, the corresponding request is authorized.

```
 Column     Type      Nullable
--------+-----------+----------
 uuid    text        not null
 time    timestamp   not null
 number  double
 text    text
```

(b) The Table Schema for Timeseries Data

There are four columns in this table. `uuid` represents the unique identifier of a point. `time` is for the timestamp of the specific datum. `number` and `text` are the values while different points could have different data types.

```
Any registered tenant can read the temperature sensors
located in her/his office when she/he is in the office.
```

(c) A Natural Language Version of the Federated Query in Figure 6.3a

Figure 6.3: The Representation of an Access Control Pattern in a Federated Query

6.4 Dynamic Dual Authorization Workflow

With ACPs, we can precisely represent under which conditions an app is allowed to access specific resources. In this section, we present an authorization workflow to enforce ACPs in actual systems tightly.

Traditional authorization mechanisms such as OAuth2 [Har12a] approves an app to access a certain set of resources statically. However, such static authorization is insufficient for building app frameworks, as discussed in Section 6.2. First, building apps are often multi-tenant apps where multiple users can use an app to access different sets of resources. For handling all the requests for the multiple users, an app needs to maintain a superset of permissions that all the users need. Second, there are dynamic conditions (`When`) to tightly represent ACPs. The results of dynamic conditions are unknown when the app is approved.

We present a *Dynamic Dual Authorization Workflow* (DDA). In DDA, resource owners do not statically delegate their authority for using resources to an app but rather approve it based on its ACP which represents all the potentially needed resources and access conditions. ACPs are evaluated at runtime because it may include dynamic conditions as well as those preventing an app from subsuming all the potential authority continuously.

For the above properties, DDA consists of two processes as offline app approvals and online authorization. In the app approval process, an app is approved by owners of all the resources potentially it can access. The app's developers need to represent its ACP accurately, and resource owners need to understand what the app can potentially access from the ACP. As we have shown in Figure 6.3c, representing the patterns in the natural language would help the resource owners to understand the app's effects.

For resource owners, approving an app's ACP means that they put trust to the people who could control the related information. An ACP's related metadata and data change over time, and so do the results of the ACP. Assume an owner has approved a Web thermostat app

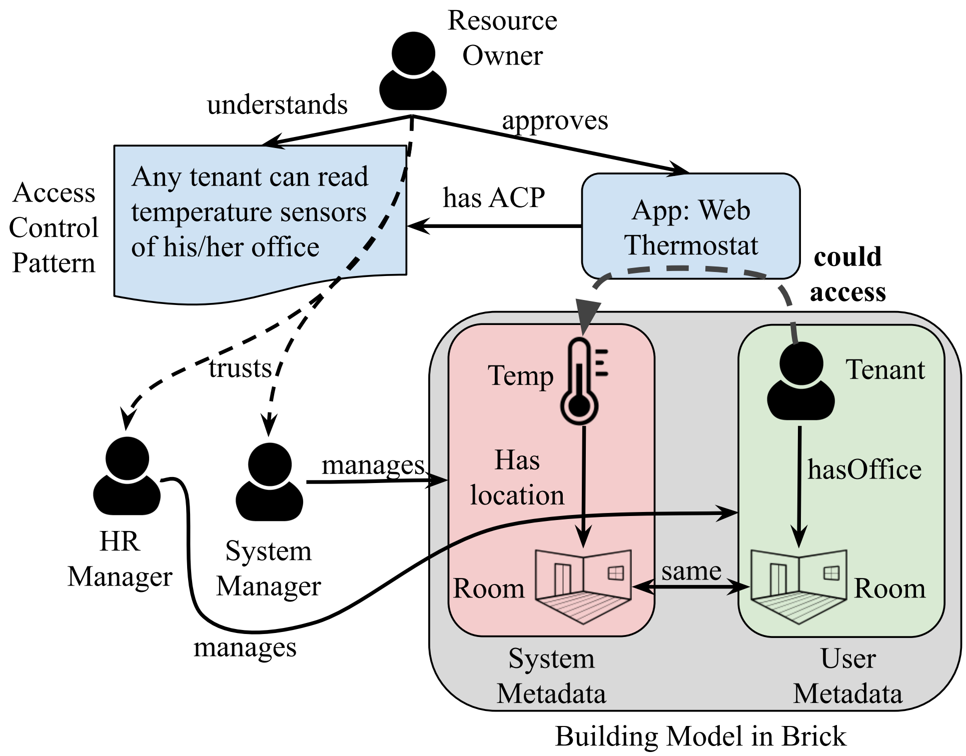


Figure 6.4: The Trust Model in Dynamic Dual Authorization Workflow

that controls the HVAC for occupants' rooms. Its ACP is, as described in Figure 6.4, that any tenant can read the `Temperature_Sensor` in her/his office. It means that the app can *potentially* access any rooms' AC given users, whether the users already exist when the app is approved or come after the approval. Thus, resource owners approve an app based on their trust in how the related metadata and data are maintained. For example, assume only an HR manager can register all the users. The approvers believe that the HR manager will behave in a trustworthy manner and that the BOS will allow only the HR manager to change user-related metadata. In this way, the resource owner can avoid putting too much authority to an app and instead distribute partial authorities to different actors for the app to be actually authorized. Figure 6.4 shows the relationships. Thus, an owner's authority delegation is partial and should be dynamically complemented by the app's access contexts.

Once an app is approved, users *could* use it if its ACP is satisfied at the moment of the

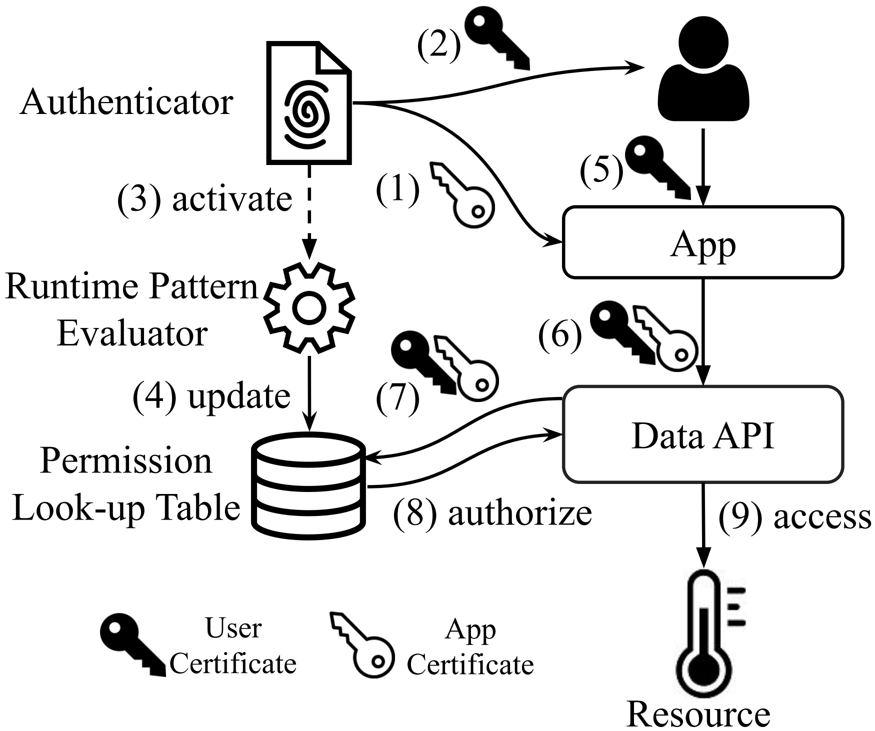


Figure 6.5: Dynamic Dual Authorization Workflow for Apps in Buildings

request. Figure 6.5 summarizes the entire online authorization process. (1) The approved app can be authenticated, for example, through a pair of client identifier and secret. (2) A user logins to the specific authentication endpoint that the BOS maintains for the app. Both of the authentication processes produce certificates that they are authenticated. This is to keep user information from apps as well as not to overprivilege apps. (3) When the BOS confirms the user's login, it activates the corresponding pattern evaluator that evaluates the app's ACP for the user, continuously if necessary. (4) The evaluator updates the permission look-up table caching the information about whether a user has access to a resource through an app. (5) The user from a browser can send requests to the app with the user certificate, and (6) the app can embed the combination of the app's certificate and the user's into an actual request to the API endpoint. (7-8) The API endpoint can check if the user within the app has access to the requested resources in the permission table, and then (9) the app can finally access the resource. Through this workflow, an app's request is dynamically evaluated under the exact context of requests instead of a superset of all the possible access throughout the app's lifetime.

6.5 Example Applications with Brick

Given the BOS components (Section 6.1), Access Control Patterns (ACPs) (Section 6.3), and Dynamic Dual Authorization Workflow (DDA) (Section 6.4), we exemplify the development procedure via the implementation of real apps, Genie, a Web thermostat, and VEnergy, an energy dashboard.

6.5.1 Genie, a Web Thermostat

Genie is a Web thermostat providing users control of the Air Conditioning (AC) unit for the users' offices [BKWA16]. Compared with physical thermostats, Genie is functionally superior; Genie is more accessible, informative, programmable, and robust. The primary function of Genie consists of informing the status of the room's AC to the user and allowing the user to control the room's AC.

Genie's app logic consists of identifying the right offices for the user and then appropriately visualizing/controlling the offices' relevant points. First, Genie identifies what rooms to control based on the office identification query as in Figure 6.6a, and we model such information as `hasOffice` between users and rooms. Genie can either potentially visualize the list of the rooms for a user to select which room to control. Figure 6.6b describes the query for the access pattern. To either understand the status or control, Genie needs to access the relevant points for a specific room, including `Temperature_Setpoint` for adjusting the target temperature, `Occupancy_Command` for turning on/off the corresponding HVAC unit, and `Temperature_Sensor` of the room. Additionally, Genie might visualize energy usage of the user's room represented by `Thermal_Power_Sensor`. All of these points are associated with a certain VAV governing the room's associated `HVAC_Zone`. Both of the ACPs specify the user type as `Occupant` to only allow

```
@prefix brick: <https://brickschema.org/schema/1.1.0/Brick#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

SELECT ?office WHERE {
    <A_USER>     brick:hasOffice   ?office.
    ?office      rdf:type          brick:Room.
    <A_USER>     rdf:type          brick:Occupant.
}
```

(a) Genie's ACP for Identifying Offices in Brick/SPARQL

A_USER may be replaced by an actual user's identifier. This query will return all the offices registered for the specific user.

```
@prefix brick: <https://brickschema.org/schema/1.1.0/Brick#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

select ?znt ?zntsp ?meter ?cmd  where {
    ?znt         rdf:type          brick:Zone_Air_Temperature_Sensor.
    ?zntsp       rdf:type          brick:Zone_Air_Temperature_Setpoint.
    ?cmd         rdf:type          brick:Occupancy_Command.
    ?meter       rdf:type          brick:Thermal_Power_Sensor.
    ?vav         rdf:type          brick:VAV.
    ?vav         brick:hasPoint    ?znt.
    ?vav         brick:hasPoint    ?zntsp.
    ?vav         brick:hasPoint    ?occ.
    ?vav         brick:hasPoint    ?meter.
    ?zone        brick:isFedBy     ?vav.
    ?zone        rdf:type          brick:HVAC_Zone.
    <A_OFFICE>   brick:isPartOf    ?zone.
    <A_USER>     brick:hasOffice   <A_OFFICE>.
    <A_USER>     rdf:type          brick:Occupant.
}
```

(b) Genie's ACP for Accessing HVAC Points

A_USER may be replaced by an actual user's identifier and A_OFFICE is replaced by the office's identifier of interest, which is retrieved from the previous query in Figure 6.6a. The query will return all the points functioning for A_OFFICE's HVAC unit.This query searches for VAVs but it can be generalized into HVAC if necessary. This access pattern does not have dynamic conditions which could provide a more tight bound over this pattern such as locational information as in Figure 6.3a.

Figure 6.6: Genie's Access Control Patterns (ACP) in Brick/SPARQL

While these patterns can be directly used as resource discovery queries inside the app, they also regulate the user type as Occupant, which provides more information about Who may use the app for resource owners to determine approval.

```
@prefix brick: <https://brickschema.org/schema/1.1.0/Brick#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

SELECT ?power ?target WHERE {
    ?power    rdf:type/rdf:subClassOf*     brick:Power_Sensor. # Or Energy_Sensor.
    # This includes Thermal_Power_Sensor, Electrical_Power_Sensor, etc.
    ?power    brick:isPointOf    ?target.
    ?target   brick:isPartOf*        ?building.
    ?target   rdf:type/rdfs:subClassOf*    brick:Location.
    FILTER ( NOT EXISTS{
        ?target   rdf:type/rdfs:subClassOf* brick:Room.
        # Exclude room-level energy usage.
    })
    FILTER ( NOT EXISTS{
        ?target   rdf:type/rdfs:subClassOf* brick:Zone.
        # Exclude zone-level energy usage.
    })
    <A_USER>  rdf:type     brick:Person.
}
```

Figure 6.7: VEnergy's ACP for Identifying Energy Sensors in Brick/SPARQL

any occupants to use Genie.

6.5.2 V-Energy, an Energy Dashboard

Visualization of energy consumption would increase awareness of energy usage for the people in an organization, which is the first step toward energy saving [AWG09]. Its basic functionality is to visualize the timeseries data of energy usage (`Power_Sensor`) for a particular area (`Location`) in buildings. However, it should not reveal too fine-grained energy consumption due to potential privacy leakage. For example, a room's energy consumption is an indicator of its occupancy. Thus, its ACP pattern, as in Figure 6.7, describes any type of `Power_Sensor` which measures a certain area, and the area could be a type of any location except types of `Zone` and `Room`.

6.6 Related Work

Access control has been well studied in various contexts over the past several decades. The most prominent topics explored in access control are access control policies and architectures to enforce access control. The basic categories of access control models are Discretionary Access Control (DAC) [GD71], Mandatory Access Control (MAC) [BL73], Role-Based Access Control (RBAC) [San98], and Attributed-Based Access Control (ABAC) [HFK+13]. In DAC (ownership), MAC (system rules), and RBAC (users' roles), access permission is determined based on explicit relationships between a user's identity and a resource. They all lack in expressivity of fine-grained access control policies and dynamic conditions of access patterns commonly required by IoT scenarios. On the other hand, ABAC can represent fine-grained contexts for access right evaluation, and many IoT frameworks have adopted such models [RLPZ19]. While ABAC is a general concept of using various attributes as a metric to make decisions on access control, it is less expressive to represent attributes by simple properties of resources. Several efforts have modeled and verified access policies using RDF for its expressivity [TMLK09, HPBL09] but they lack in expressing dynamic conditions as in [NSB14, SST18b]. We have designed access control patterns (ACPs) accommodating different data sources to represent various attributes in smart buildings while ACPs can also specify trusted data sources.

There are also two basic categories of architectures for enforcing access control policies, which are token-based architectures and policy-based architectures. OAuth [Har12b] is a widely adopted token-based architecture for various platforms such as Web, smartphones, and IoT. A user is once evaluated and receives an access token representing her permissions on resources. In a native OAuth scenario, resource owners need to statically validate users per resource, which is often not feasible for building apps that need many resources in dynamic contexts. IoT-OAS automatically generates OAuth tokens based on application logic for users [CPG+14]. However, its architecture relies on the OAuth workflow, and representing many resources at the same time needs many tokens for different services. XACML [ANP+03] is a standard architecture for policy-based enforcement where policy is evaluated for access requests (mostly online). Because access decisions can be automatically made based on a policy, policy-based architectures can handle complex conditions needed by building apps. Our Dual Dynamic Authorization (DDA) workflow additionally introduces multi-tenancy by considering apps and users at the runtime policy evaluation stage to minimize an app's access capability compared with static authorization mechanisms. Because DDA evaluates policies at runtime, we need a careful optimization for evaluating permissions, which we defer to future work.

6.7 Summary

While Brick by itself provides a complete representation for resource discoverability to apps, there are other features required to deploy an app to arbitrary buildings in practice. Representing and guaranteeing security measures is a necessity for building owners and property managers to deploy third-party apps safely. Existing access control models fall short of representing comprehensible access control patterns in smart buildings with multi-tenancy and a primarily relational resource representation. In this chapter, we have presented an extensive analysis of 125 app papers to find the exhaustive description of access requirements by apps as well as an architecture to tightly enforce the access control patterns.

In our analysis of the app papers from two major venues, we have identified three canonical information dimensions to represent access control patterns as `Who`, `What`, and `When`[8]. To represent all the dimensions needed in evaluating access permissions, we have to refer to the heterogeneous information sources including metadata and timeseries data. To accommodate different information sources for access control, we have augmented Brick with user information and proposed a model federating the augmented Brick and timeseries data.

[8] Again, `When` represents not just temporal information but dynamic conditions.

We also found that, for the multi-tenancy of building apps and the dynamic conditions in the access control patterns, users' requests should be evaluated at runtime with the context of users and apps together. Our proposed Dual Dynamic Authorization (DDA) workflow regulates what an app can *potentially* access by users, instead of statically delegating a large set of access permissions to the app. With DDA workflow, a resource owner approves an app with a comprehensible but exhaustive access pattern, and the app is not overprivileged but keeps the minimal authority in realtime. Our future work includes evaluating the usability and the performance of access control patterns and DDA workflow in practice. We have developed and tested two exemplary building apps, Genie, a Web thermostat, and VEnergy, an energy dashboard. Through the apps, we have shown the feasibility of our workflow and the simplicity of the access control patterns.

Our proposed access control patterns and security enforcement mechanism are the key components to validate our thesis: a large-scale building app deployment should be built upon a structured metadata. It is not scalable to list each of the components that an app can possibly access in different contexts, because an app might access many resources which human managers can hardly comprehend. Access control mechanisms should refer to the structured metadata, Brick, and its associated data sources, which can be tightly enforced while human managers can interpret the consequences at scale through access control patterns and DDA.

6.8 Acknowledgment

Chapter 6, in part, also contains material as it appears in Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation, 2019 by authors Jason Koh, Dezhi Hong, Shreyas Nagare, Sudershan Boovaraghava, Yuvraj Agarwal and Rajesh Gupta. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Future Work

The metadata models and methods presented in this dissertation have shown the feasibility of a metadata schema, namely Brick, as a base for metadata organization. A successful use of this schema can also enable a new regime of applications, Building Applications, that are developed third-party companies, deployed and used by building operators as well as used building occupants. This thesis provides early evidence that such applications and a new platform are indeed feasible. However, in order to have an impact, this work and underlying schema needs widespread education and adoption, activities that are beginning to appear.

The evolution of Brick Consortium (http://brickschema.org) represents a first step towards both requirements. The consortium is currently led by Johnson Controls while participation agreements are being pursued with both academic and industry partners. While non-profit consortia have a rich history of adoption and success, there are challenges faced by Brick consortium due to the nature of commercial activities and its intersection with security, safety and public policy. It is not clear at this time, how widespread this adoption or buy-in into the consortium will be, but it is a critical first step towards building a broader user community that can provide feedback and develop new features and capabilities to create overall value.

As we discussed in Section 3.8, our primary role in designing Brick was to ensure the soundness, generality and integrity of the schema, whereas actual users would be needed to fine-tune Brick's scope. Brick will have a diverse set of users as a software engineer, database experts, building engineers, policymakers, etc., and Brick will need to evolve to meet their needs continuously.

As demonstrated in our examples, applications can be developed using the Brick schema that makes them portable across buildings. Yet, the deployment process needs to be developed to ensure necessary safety and privacy requirements that have not been a subject of this thesis. More broadly, the existing building operating systems research has been centered on feasibility, optimality and to a limited extent on the cybersecurity of the systems. However, the guarantee of safety and privacy is relatively unexplored and a continuing work in progress in the research community. Without such guarantees, building owners would not simply install third-party application without a human-centered agent (i.e., an applications provider company). An incorrectly implemented application might malfunction equipment with significant financial or legal repercussions. Maturing formal verification or model checking methods and tools could be used [LBL+16] by ensuring capture and reasoning of the Brick model in a suitable logical framework amenable to these methods. Among the properties that will need to be ensure, there are specific needs stemming from potential leak of private information by an application. A sandboxing or information flow control architecture has been widely studied in the Web [GLS+12] and Android systems [BBH+15], and these are being explored for use in smart buildings by projects such as CONIX (http://conix.io).

As we examined the metadata models and methods for buildings, we have found that this is common in other system integration problems such as Internet of Things (IoT) and smart cities [KSB+17]. A few projects have attempted to define a meta-structure to define the core concepts that all the systems and possible apps would need [BAD+16a, BBB+14, KBKT15]. However, a major limitation continues to be an inadequate actual vocabulary to represent necessary entities in a uniform manner. In principle, it is possible to repeat a Brick-style experiment for bigger domains with the Brick's structure to coherently extend concepts throughout the

Classes. However, in the process, we would also need a much larger collaboration community to organize important concepts in the target domains, along the lines described in Section 3.1.2. Further, large-scale systems such as smart cities would face a similar problem of extracting semantics out of unstructured information such as building metadata normalization, but at a much larger scale and complexity. By contrast, Brick was possible because building metadata is significantly more accessible from existing building systems than would be the case with Smart Cities. Also, a city infrastructure is usually more operationally critical and privacy-sensitive subject to extensive regulations by multiple administrative entities. We would need significantly more infrastructure and workflows to carry out such research safely over such frameworks while maximizing accessibility to the data and systems.

Chapter 8

Conclusion

Recently, many researchers have shown the potential of optimizing and operating building systems algorithmically to improve the quality of living and working environments through the use of dynamic sensing and actuation capabilities. Indeed, buildings can be more energy-efficient, accessible and controllable for their occupants, and more reactive upon various events such as emergencies and dynamic pricing. Yet, such demonstrations have been limited to custom designed strategies for individual buildings due to the heterogeneity of building systems, their information captured through metadata generated and used by the building management systems. The inherent potential for building as a sensing, computing and actuation platform remained unrealized because of the manual "wiring" needed to tie each optimization procedure to individual buildings with different information conventions, nomenclature and different configurations of its instruments. This disparity between buildings and the needed application programming abstractions impedes the seamless deployment of apps at any buildings, because of the lack of standard in how to represent resources and their utilization methods.

This dissertation has focused on how to model and manage metadata for building systems that can be used for capturing resources and entities in a standardized way so that building-independent applications can consistently refer to underlying resources systematically across buildings, similarly to Hardware Abstraction Layer (HAL) in modern operating systems. With a standard representation, apps can be agnostic to the heterogeneity in underlying hardware and conventions.

In this dissertation, we have presented a metadata schema as a standard representation of resources for buildings, and how to manage them at scale:

- We have presented Brick, a metadata schema to uniformly represent resources in buildings that apps can refer to.
- We have presented two methodologies, Scrabble and Quiver, that can map unstructured information sources in buildings into Brick with minimal human effort.
- We have presented a meta-framework, Plaster, composed of a programming interface and a Web service, for different metadata methods.
- We have presented access control patterns with Brick and a dynamic dual authorization (DDA) workflow that can tightly control building apps' minimal access with Brick.
- We have exemplified Brick and DDA workflow with actual apps as Genie, a Web thermostat, and VEnergy, an energy dashboard.

By evaluating the models and methods over real-world data and implementations, we have shown the effectiveness and the feasibility of the models and methods. Brick can coherently model buildings by representing canonical concepts as classes. Apps can rely upon Brick to identify the right resources and represent their requirements. Our collection of metadata normalization methods helps to bridge the gap between the formal model, Brick, and unstructured information sources in the real world. Both the models and methods have been adopted in the industry in various formats. Companies have either directly adopted Brick or developed their own schema based on Brick. The research community and companies have used Scrabble and Plaster as a baseline of their research or in their development process.

Our broader vision is to enable buildings as a general application execution platform similar to, for instance, Android app markets. Brick and our metadata mapping methods are a stepping stone toward this vision as enabling coherent standardization of existing building systems. However, to put trust in third-party apps, same as in smartphones, the entire Building Operating Systems (BOSes) should provide usable security measures that building owners and managers can easily interpret and safely trust at the same time. We have designed an access control model based on Brick and Dynamic Dual Authorization (DDA) workflow based on a comprehensive study of 125 app papers. The access control model and workflow can tightly regulate apps' access requirements while providing app manifests that app approvers, such as building owners and managers, can easily interpret.

While we have demonstrated a definite progress towards greater accessibility to building data and systems, Brick and its ecosystem are ultimately dependent upon various community efforts including the other authors of the Brick code, community contributors, and actual users. Brick can be useful only when the community shares the same philosophy. Various interest parties should discuss and foster Brick schema and methods for the different parts of the building life cycle. While we have shown feasibility of metadata models and methods, we continue to pursue building an active community of system builders for smart building environments.

Our focus on buildings has enabled us to provide a successful demonstration of metadata capture and use, the broader problem of managing heterogeneous data and systems appears in many other domains. It would be a recurring theme whenever domains with heterogeneous subsystems (or multiple domains) need to exploit computing engines in the loop for decision making and knowledge discovery. Thus, even though its specification and implementation may vary, the quintessential point in our main thesis would remain same in any other systems: *always schema-first*.

Bibliography

[AFE+16] Rachit Agarwal, David Gomez Fernandez, Tarek Elsaleh, Amelie Gyrard, Jorge Lanza, Luis Sanchez, Nikolaos Georgantas, and Valerie Issarny. Unified iot ontology to enable interoperability and federation of testbeds. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 70–75. IEEE, 2016.

[AGM15] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In *International Semantic Web Conference*, pages 374–389. Springer, 2015.

[AK09] Arvind Arasu and Raghav Kaushik. A grammar-based entity representation framework for data cleaning. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 233–244. ACM, 2009.

[AKA+19] Michael P. Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E. Culler, and Raluca Ada Popa. Wave: A decentralized authorization framework with transitive delegation. In *28th USENIX Security Symposium*, 2019.

[AKC+17] Michael P Andersen, John Kolb, Kaifei Chen, David E Culler, and Randy Katz. Democratizing authority in the built environment. In *BuildSys*, page 23. ACM, 2017.

[ANP+03] Anne Anderson, Anthony Nadalin, B Parducci, D Engovatov, H Lockhart, M Kudo, P Humenn, S Godik, S Anderson, S Crocker, et al. extensible access control markup language (xacml) version 1.0. *OASIS*, 2003.

[arc14] Arctic, 2014. https://github.com/manahl/arctic, last visited: 11-01-2019.

[ASH16] BACnet-A Data Communication Protocol for Building Automation and Control Networks. Standard, ASHRAE, GA, USA, 2016.

[AWG09] Yuvraj Agarwal, Thomas Weng, and Rajesh K Gupta. The energy dashboard: improving the visibility of energy consumption at a campus-wide scale. In *BuildSys*, 2009.

[aws19] AWS IoT Authorization, 2019. https://docs.aws.amazon.com/iot/latest/developerguide/authorization.html, last visited: 11-01-2019.

[azu16] Microsoft Azure Machine Learning Studio, 2016. https://studio.azureml.net/, last visited: 11-01-2019.

[BAD+16a] D Brickley, Y Agarwal, LM Daniele, J Gluck, RV Guha, T Hardie, J Koh, A Keränen, M Koster, D Raggett, et al. Iot and schema. org: Getting started. Technical report, TNO, 2016.

[BAD+16b] Dan Brickley, Yuvraj Agarwal, Laura Daniele, Joshua Gluck, Ramachandra Guha, Ted Hardie, Jason Koh, Ari Keränen, Michael Koster, Dave Raggett, and Max Senges. Iot and schema.org: Getting started. Technical report, iot.schema.org, 2016.

[BBB+14] Pierfrancesco Bellini, Monica Benigni, Riccardo Billero, Paolo Nesi, and Nadia Rauch. Km4city ontology building vs data harvesting and cleaning for smart-city services. *Journal of Visual Languages & Computing*, 25(6):827–839, 2014.

[BBF+16a] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, et al. Brick: Towards a unified metadata schema for buildings. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*, pages 41–50. ACM, 2016.

[BBF+16b] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, et al. Brick: Towards a unified metadata schema for buildings. In *BuildSys*, 2016.

[BBF+18a] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, et al. Brick: Metadata schema for portable smart building applications. *Applied energy*, 226:1273–1292, 2018.

[BBF+18b] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, et al. Brick: Metadata schema for portable smart building applications. *Applied Energy*, 2018.

[BBH+15] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 691–706, 2015.

[BBS07] Steffen Bickel, Michael Brückner, and Tobias Scheffer. Discriminative learning for differing training and test distributions. In *Proceedings of the 24th international conference on Machine learning*, pages 81–88. ACM, 2007.

[BC94] Donald J Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *KDD workshop*, volume 10, pages 359–370. Seattle, WA, 1994.

[BCH+15] Arka Bhattacharya, David Culler, Dezhi Hong, Kamin Whitehouse, Jorge Ortiz, and Eugene Wu. Automated Metadata Construction To Support Portable Building Applications. In *Proceedings of the ACM Conference on Embedded Systems For Energy-Efficient Built Environments*. ACM, 2015.

[bd319] BuildingDepot 3.0, 2019. https://buildingdepot.org/, last visited: 11-01-2019.

[Bha16] Arka Bhattacharya. *Enabling Scalable Smart-Building Analytics*. PhD thesis, UC Berkeley, 2016.

[BHC+15a] Arka A Bhattacharya, Dezhi Hong, David Culler, Jorge Ortiz, Kamin Whitehouse, and Eugene Wu. Automated metadata construction to support portable building applications. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*, pages 3–12. ACM, 2015.

[BHC+15b] Arka A Bhattacharya, Dezhi Hong, David Culler, Jorge Ortiz, Kamin Whitehouse, and Eugene Wu. Automated metadata construction to support portable building applications. In *BuildSys*, pages 3–12. ACM, 2015.

[bid] Bidgely. http://www.bidgely.com/, last visited: 11-01-2019.

[BIPR12] Kedar Bellare, Suresh Iyengar, Aditya G Parameswaran, and Vibhor Rastogi. Active sampling for entity matching. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1131–1139. ACM, 2012.

[BKC+14] Christian Beckel, Wilhelm Kleiminger, Romano Cicchetti, Thorsten Staake, and Silvia Santini. The eco data set and the performance of non-intrusive load monitoring algorithms. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*, pages 80–89. ACM, 2014.

[BKP+14] Nipun Batra, Jack Kelly, Oliver Parson, Haimonti Dutta, William Knottenbelt, Alex Rogers, Amarjeet Singh, and Mani Srivastava. Nilmtk: an open source toolkit for non-intrusive load monitoring. In *Proceedings of the 5th international conference on Future energy systems*, pages 265–276. ACM, 2014.

[BKWA16] Bharathan Balaji, Jason Koh, Nadir Weibel, and Yuvraj Agarwal. Genie: a longitudinal study comparing physical and software thermostats in office buildings. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 1200–1211. ACM, 2016.

[BL73] D Elliott Bell and Leonard J LaPadula. Secure computer systems: Mathematical foundations. Technical report, MITRE CORP BEDFORD MA, 1973.

[BLHL+01] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[BM03] Mikhail Bilenko and Raymond J. Mooney. Adaptive duplicate detection using learnable string similarity measures. *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 39–48, 2003.

[BORZ17] Peter Bailis, Kunle Olukotun, Christopher Ré, and Matei Zaharia. Infrastructure for usable machine learning: The stanford DAWN project. *CoRR*, abs/1705.07538, 2017.

[bot] Building Topology Ontology. https://w3c-lbd-cg.github.io/bot/#, last visited: 11-26-2019.

[BPC15] Arka Bhattacharya, Joern Ploennigs, and David Culler. Short paper: Analyzing metadata schemas for buildings: The good, the bad, and the ugly. In *BuildSys - ACM Conference on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 33–34. ACM, 2015.

[BPV14] Harald Bauer, Mark Patel, and Jan Veira. The internet of things: Sizing up the opportunity. Technical report, McKinsey, 2014. Accessed: 2019-10-30.

[Bus97a] Steven T Bushby. BACnet™: A standard communication infrastructure for intelligent buildings. *Automation in Construction*, 6(5):529–540, 1997.

[Bus97b] Steven T Bushby. Bacnettm: a standard communication infrastructure for intelligent buildings. *Automation in Construction*, 6(5-6):529–540, 1997.

[BVNA15a] Bharathan Balaji, Chetan Verma, Balakrishnan Narayanaswamy, and Yuvraj Agarwal. Zodiac: Organizing large deployment of sensors to create reusable applications for buildings. In *BuildSys*, pages 13–22. ACM, 2015.

[BVNA15b] Bharathan Balaji, Chetan Verma, Balakrishnan Narayanaswamy, and Yuvraj Agarwal. Zodiac: Organizing large deployment of sensors to create reusable applications for buildings. In *BuildSys - ACM Conference on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 13–22. ACM, 2015.

[CBB+12] Michael Compton, Payam Barnaghi, Luis Bermudez, RaúL GarcíA-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, et al. The ssn ontology of the w3c semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, 2012.

[CBNKL18] Maximilian Christ, Nils Braun, Julius Neuffer, and Andreas W. Kempa-Liehr. Time series feature extraction on basis of scalable hypothesis tests (tsfresh – a python package). *Neurocomputing*, 2018.

[CCGK07] Surajit Chaudhuri, Bee-Chung Chen, Venkatesh Ganti, and Raghav Kaushik. Example-driven design of efficient record matching queries. In *Proceedings of the 33rd international conference on Very large data bases*, pages 327–338. VLDB Endowment, 2007.

[Chr12a] Peter Christen. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection*. Springer Science & Business Media, 2012.

[Chr12b] Peter Christen. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection*. Springer Science & Business Media, 2012.

[CKAK15a] Victor Charpenay, Sebastian Kabisch, Darko Anicic, and Harald Kosch. An ontology design pattern for iot device tagging systems. In *Internet of Things (IOT), 2015 5th International Conference on the*, pages 138–145. IEEE, 2015.

[CKAK15b] Victor Charpenay, Sebastian Kabisch, Darko Anicic, and Harald Kosch. An ontology design pattern for iot device tagging systems. In *5th Int. Conf. on the Internet of Things (IOT)*, pages 138–145. IEEE, 2015.

[CM19] Lingzhen Chen and Alessandro Moschitti. Transfer learning for sequence labeling using source model and target data. *arXiv preprint arXiv:1902.05309*, 2019.

[com] Comfy. https://www.comfyapp.com/, last visited: 11-01-2019.

[Con04] Gene Ontology Consortium. The gene ontology (go) database and informatics resource. *Nucleic acids research*, 32(suppl_1):D258–D261, 2004.

[CPG+14] Simone Cirani, Marco Picone, Pietro Gonizzi, Luca Veltri, and Gianluigi Ferrari. Iot-oas: An oauth-based authorization service architecture for secure services in iot scenarios. *IEEE sensors journal*, 15(2):1224–1234, 2014.

[CVW15] Peter Christen, Dinusha Vatsalan, and Qing Wang. Efficient entity resolution with adaptive and interactive training data selection. In *Data Mining (ICDM), 2015 IEEE International Conference on*, pages 727–732. IEEE, 2015.

[DdHR15] LM Daniele, FTH den Hartog, and JBM Roes. Study on semantic assets for smart appliances interoperability: D-S4: Final report. Technical report, European Union, 2015.

[DES+15] Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The bigdawg polystore system. *ACM Sigmod Record*, 44(2):11–16, 2015.

[DGL96] Giuseppe De Giacomo and Maurizio Lenzerini. Tbox and abox reasoning in expressive description logics. *KR*, 96(316-327):10, 1996.

[DNM16] Antonio De Nicola and Michele Missikoff. A lightweight methodology for rapid ontology engineering. *Communications of the ACM*, 59(3):79–86, 2016.

[EKB17] Charbel El Kaed and Matthieu Boujonnier. Forte: A federated ontology and timeseries query engine. In *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 983–990. IEEE, 2017.

[EKBD17] Charbel El Kaed, Matthieu Boujonnier, and Stephen Dillon. Forte: A federated ontology and timeseries query engine. In *The 3rd IEEE International Conference on Smart Data. IEEE*, 2017.

[ene] EnerNOC. https://www.enernoc.com/, last visited: 11-01-2019.

[FBK09] Nicholas Fernandez, Michael R Brambley, and Srinivas Katipamula. *Self-correcting HVAC Controls: Algorithms for Sensors and Dampers in Air-handling Units*. Pacific Northwest National Laboratory, 2009.

[FCKB16] Jonathan Fürst, Kaifei Chen, Randy H Katz, and Philippe Bonnet. Crowd-sourced bms point matching and metadata maintenance with babel. In *Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6. IEEE, 2016.

[FEHF09] Ali Farhadi, Ian Endres, Derek Hoiem, and David Forsyth. Describing objects by their attributes. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 1778–1785. IEEE, 2009.

[FGC19] Gabe Fierro, Sriharsha Guduguntla, and David E Culler. Dataset: An open dataset and collection tool for bms point labels. In *Workshop on Data Acquisition To Analysis (DATA'19)*, 2019.

[FKA+19] Gabe Fierro, Jason Koh, Yuvraj Agarwal, Rajesh Gupta, and David Culler. Beyond a house of sticks: Formalizing metadata tags with brick. In *Proceedings of the 6th Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*. ACM, 2019.

[FOCE12] Romain Fontugne, Jorge Ortiz, David Culler, and Hiroshi Esaki. Empirical mode decomposition for intrinsic-relationship extraction in large sensor deployments. In *Workshop on Internet of Things Applications, IoT-App*, volume 12, 2012.

[FPA+18] Gabe Fierro, Marco Pritoni, Moustafa AbdelBaky, Paul Raftery, Therese Peffer, Greg Thomson, and David E Culler. Mortar: an open testbed for portable building analytics. In *Proceedings of the 5th Conference on Systems for Built Environments*, pages 172–181. ACM, 2018.

[FRS+13] Aurélie Foucquier, Sylvain Robert, Frédéric Suard, Louis Stéphan, and Arnaud Jay. State of the art in building modelling and energy performances prediction: A review. *Renewable and Sustainable Energy Reviews*, 23:272–288, 2013.

[GB18] Jingkun Gao and Mario Bergés. A large-scale evaluation of automated metadata inference approaches on sensors from air handling units. *Advanced Engineering Informatics*, 37, 2018.

[GBT99] KR Godfrey, HA Barker, and AJ Tucker. Comparison of perturbation signals for linear system identification in the frequency domain. In *Control Theory and Applications, IEE Proceedings-*, volume 146, pages 535–548. IET, 1999.

[GD71] G Scott Graham and Peter J Denning. Protection: principles and practice. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, pages 417–429, 1971.

[GFJH08] Jing Gao, Wei Fan, Jing Jiang, and Jiawei Han. Knowledge transfer via multiple model local structure mapping. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 283–291. ACM, 2008.

[GG05] Cyril Goutte and Eric Gaussier. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In *European Conference on Information Retrieval*, pages 345–359. Springer, 2005.

[GGM+02] Aldo Gangemi, Nicola Guarino, Claudio Masolo, Alessandro Oltramari, and Luc Schneider. Sweetening ontologies with dolce. In *International Conference on Knowledge Engineering and Knowledge Management*, pages 166–181. Springer, 2002.

[GLS+12] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 47–60, 2012.

[God93] Keith Godfrey. *Perturbation signals for system identification*. Prentice Hall International (UK) Ltd., 1993.

[Gooa] Google. Google search api documentation: Understand how structured data works. https://developers.google.com/search/docs/guides/intro-structured-data. Accessed: 2019-10-30.

[Goob] Google. Permissions overview. https://developer.android.com/guide/topics/permissions/overview. Accessed: 2019-12-05.

[GPB15a] Jingkun Gao, Joern Ploennigs, and Mario Berges. A Data-driven Meta-data Inference Framework for Building Automation Systems. In *Proceedings of the ACM Conference on Embedded Systems For Energy-Efficient Built Environments*. ACM, 2015.

[GPB15b] Jingkun Gao, Joern Ploennigs, and Mario Berges. A data-driven meta-data inference framework for building automation systems. In *BuildSys - ACM Conference on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 23–32. ACM, 2015.

[Gru95] Thomas R Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, 43(5):907–928, 1995.

[GSH+09] Arthur Gretton, Alex Smola, Jiayuan Huang, Marcel Schmittfull, Karsten Borgwardt, and Bernhard Schölkopf. Covariate shift by kernel mean matching. *Dataset shift in machine learning*, 3(4):5, 2009.

[GTA+15] M Gollner, A Trouve, I Altintas, J Block, D Callafon, C Clements, A Cortes, E Ellicott, JB Filippi, M Finney, et al. Towards data-driven operational wildfire spread modeling. Technical report, Wifire Workshop, January 2015.

[GTBC05] KR Godfrey, AH Tan, HA Barker, and B Chong. A survey of readily accessible perturbation signals for system identification in the frequency domain. *Control Engineering Practice*, 13(11):1391–1402, 2005.

[GUA+16] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. *The Journal of Machine Learning Research*, 17(1):2096–2030, 2016.

[Har12a] Dick Hardt. The oauth 2.0 authorization framework. 2012.

[Har12b] Dick Hardt. The oauth 2.0 authorization framework. 2012.

[hay] Project Haystack. http://project-haystack.org/, last visited: 11-01-2019.

[HFK+13] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (abac) definition and considerations (draft). *NIST special publication*, 800(162), 2013.

[HGW17] Dezhi Hong, Quanquan Gu, and Kamin Whitehouse. High-dimensional time series clustering via cross-predictability. In *AISTATS*, pages 642–651, 2017.

[HK97] Markus Horstmann and Mary Kirtland. Dcom architecture. *Microsoft white paper*, 1997.

[HK18] Jakob Hviid and Mikkel Baun Kjærgaard. Service abstraction layer for building operating systems: Enabling portable applications and improving system resilience. In *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, pages 1–6. IEEE, 2018.

[HOWC13] Dezhi Hong, Jorge Ortiz, Kamin Whitehouse, and David Culler. Towards automatic spatial verification of sensor placement in buildings. In *BuildSys - ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 1–8. ACM, 2013.

[HPBL09] James Hollenbach, Joe Presbrey, and Tim Berners-Lee. Using rdf metadata to enable access control on the social semantic web. In *Proceedings of the Workshop on Collaborative Construction, Management and Linking of Structured Knowledge (CK2009)*, volume 514, page 167, 2009.

[Hug17] Nicolas Hug. Surprise, a Python library for recommender systems. http://surpriselib.com, 2017.

[HWKH19] Karl Hammar, Erik Oskar Wallin, Per Karlberg, and David Hälleberg. The realestatecore ontology. In *International Semantic Web Conference*, pages 130–145. Springer, 2019.

[HWOW15a] Dezhi Hong, Hongning Wang, Jorge Ortiz, and Kamin Whitehouse. The building adapter: Towards quickly applying building analytics at scale. In *BuildSys*, pages 123–132. ACM, 2015.

[HWOW15b] Dezhi Hong, Hongning Wang, Jorge Ortiz, and Kamin Whitehouse. The building adapter: Towards quickly applying building analytics at scale. In *BuildSys - ACM Conference on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 123–132. ACM, 2015.

[HWW15] Dezhi Hong, Hongning Wang, and Kamin Whitehouse. Clustering-based active learning on sensor type classification in buildings. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 363–372. ACM, 2015.

[Hyd03] Mark Hydeman. *Advanced Variable Air Volume: System Design Guide: Design Guidelines*. California Energy Commission, 2003.

[IET07] IETF Network Working Group. Internet Security Glossary, Version 2, 2007. https://tools.ietf.org/html/rfc4949, last visited: 11-01-2019.

[IMM+13] Rizwan Iqbal, Masrah Azrifah Azmi Murad, Aida Mustapha, Nurfadhlina Mohd Sharef, et al. An analysis of ontology engineering methodologies: A literature review. *Research journal of applied sciences, engineering and technology*, 6(16):2993–3000, 2013.

[ISO14] Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries. Standard, buildingSMART, April 2014.

[Jen96] Finn V Jensen. *An introduction to Bayesian networks*, volume 210. UCL press London, 1996.

[JHC+19a] Krzysztof Janowicz, Armin Haller, Simon JD Cox, Danh Le Phuoc, and Maxime Lefrançois. Sosa: A lightweight ontology for sensors, observations, samples, and actuators. *Journal of Web Semantics*, 56:1–10, 2019.

[JHC+19b] Krzysztof Janowicz, Armin Haller, Simon JD Cox, Danh Le Phuoc, and Maxime Lefrançois. Sosa: A lightweight ontology for sensors, observations, samples, and actuators. *Journal of Web Semantics*, 56:1–10, 2019.

[JKK+13] Deokwoo Jung, Varun Badrinath Krishna, Ngo Quang Minh Khiem, Hoang Hai Nguyen, and David KY Yau. Energytrack: Sensor-driven energy use analysis system. In *BuildSys - ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 1–8. ACM, 2013.

[JNMM18] Achin Jain, Truong Nghiem, Manfred Morari, and Rahul Mangharam. Learning and control using gaussian processes. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 140–149. IEEE, 2018.

[JSSJ11] Marco Jahn, Tobias Schwartz, Jonathan Simon, and Marc Jentsch. Energypulse: tracking sustainable behavior in office environments. In *Int. Conf. on Energy-Efficient Computing and Networking*, pages 87–96. ACM, 2011.

[KAB14] Merthan Koc, Burcu Akinci, and Mario Bergés. Comparison of linear correlation and a statistical dependency measure for inferring spatial relation of temperature sensors in buildings. In *BuildSys*, pages 152–155. ACM, 2014.

[KBA+16] Jason Koh, Bharathan Balaji, Vahideh Akhlaghi, Yuvraj Agarwal, and Rajesh Gupta. Quiver: Using control perturbations to increase the observability of sensor data in smart buildings. *arXiv preprint arXiv:1601.07260*, 2016.

[KBKT15] Nicos Komninos, Charalampos Bratsas, Christina Kakderi, and Panagiotis Tsarchopoulos. Smart city ontologies: Improving the effectiveness of smart city applications. *Journal of Smart Cities*, 2015.

[KBS+18] Jason Koh, Bharathan Balaji, Dhiman Sengupta, Julian McAuley, Rajesh Gupta, and Yuvraj Agarwal. Scrabble: Transferrable semi-automated semantic metadata normalization using intermediate representation. In *BuildSys*. ACM, 2018.

[KDHL+11] Andrew Krioukov, Stephen Dawson-Haggerty, Linda Lee, Omar Rehmane, and David Culler. A living laboratory study in personalized automated lighting controls. In *BuildSys - ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 1–6. ACM, 2011.

[ker] Keras. https://keras.io/, last visited: 11-01-2019.

[kgs] KGS Buildings. http://www.kgsbuildings.com/, last visited: 11-01-2019.

[KH18] Tobias Käfer and Andreas Harth. Rule-based programming of user agents for linked data. In *LDOW@ WWW*, 2018.

[KHG+18] Jason Koh, Dezhi Hong, Rajesh Gupta, Kamin Whitehouse, Hongning Wang, and Yuvraj Agarwal. Plaster: An integration, benchmark, and development framework for metadata normalization methods. In *Proceedings of the 5th Conference on Systems for Built Environments*, pages 1–10. ACM, 2018.

[KHN+19] Jason Koh, Dezhi Hong, Shreyas Nagare, Sudershan Boovaraghavan, Yuvraj Agarwal, and Rajesh Gupta. Who can access what, and when? understanding minimal access requirements of building applications. In *Proceedings of the 6th Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*. ACM, 2019.

[KNO+01] Neil E Klepeis, William C Nelson, Wayne R Ott, John P Robinson, Andy M Tsang, Paul Switzer, Joseph V Behar, Stephen C Hern, and William H Engelmann. The national human activity pattern survey (nhaps): a resource for assessing exposure to environmental pollutants. *Journal of Exposure Science and Environmental Epidemiology*, 11(3):231, 2001.

[Koh16] Jason Koh. Research exam: Adopting semantic ontology onto iot systems focused on smart buildings. Technical report, University of California, San Diego, March 2016.

[KSB+17] Jason Koh, Sandeep Sandha, Bharathan Balaji, Daniel Crawl, Ilkay Altintas, Rajesh Gupta, and Mani Srivastava. Data hub architecture for smart cities. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, page 77. ACM, 2017.

[LBL+16] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F Karlsson, Dongmei Zhang, and Feng Zhao. Systematically debugging iot control system correctness for building automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*, pages 133–142. ACM, 2016.

[LD01] Xiong-Fu Liu and Arthur Dexter. Fault-tolerant supervisory control of vav air-conditioning systems. *Energy and Buildings*, 33(4):379–389, 2001.

[Lef17] Maxime Lefrançois. 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,*, July 2017.

[Lju98] Lennart Ljung. *System identification*. Springer, 1998.

[LLHW19] Lu Lin, Zheng Luo, Dezhi Hong, and Hongning Wang. Sequential learning with active partial labeling for building metadata. In *Proceedings of the 6th ACM*

International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation, pages 189–192. ACM, 2019.

[LMP+01] John Lafferty, Andrew McCallum, Fernando Pereira, et al. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the eighteenth international conference on machine learning, ICML*, volume 1, pages 282–289, 2001.

[Mar91] James G March. Exploration and exploitation in organizational learning. *Organization science*, 2(1):71–87, 1991.

[MBY+16] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(1):1235–1241, 2016.

[MH09] Alan Marchiori and Qi Han. Using circuit-level power measurements in household energy management systems. In *BuildSys - ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 7–12. ACM, 2009.

[MJLM19] Elena Markoska, Aslak Johansen, and Sanja Lazarova-Molnar. A framework for fully automated performance testing for smart buildings. In *Third International Congress on Information and Communication Technology*, pages 235–243. Springer, 2019.

[mlj16] MLJAR, 2016. https://mljar.com/, last visited: 11-01-2019.

[MLL99] Darryl Jon Mocek, Kester Li, and Jonathan Michael Levine. Natural language translation of an sql query, July 13 1999. US Patent 5,924,089.

[NDdL16] Canh Ngo, Yuri Demchenko, and Cees de Laat. Multi-tenant attribute-based access control for cloud infrastructure services. *Journal of Information Security and Applications*, 27:65–84, 2016.

[NDRT13] Nagarajan Natarajan, Inderjit S Dhillon, Pradeep K Ravikumar, and Ambuj Tewari. Learning with noisy labels. In *Advances in neural information processing systems*, pages 1196–1204, 2013.

[nia] Niagara AX. https://www.tridium.com/en/products-services/niagara-ax, last visited: 11-01-2019.

[NNBU+13] Axel-Cyrille Ngonga Ngomo, Lorenz Bühmann, Christina Unger, Jens Lehmann, and Daniel Gerber. Sorry, i don't speak sparql: translating sparql queries into natural language. In *Proceedings of the 22nd international conference on World Wide Web*, pages 977–988. ACM, 2013.

[NSB14] Ricardo Neisse, Gary Steri, and Gianmarco Baldini. Enforcement of security policy rules for the internet of things. In *2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 165–172. IEEE, 2014.

[ocf]

[Oka07] Naoaki Okazaki. Crfsuite: a fast implementation of conditional random fields (crfs), 2007.

[OWL] OWL Namespace. http://www.w3.org/2002/07/owl#, last visited: 11-01-2019.

[PBCM15a] Marco Pritoni, Arka Bhattacharya, David Culler, and Mark Modera. Short Paper: A Method for Discovering Functional Relationships Between Air Handling Units and Variable-Air-Volume Boxes From Sensor Data. In *Proceedings of the ACM Conference on Embedded Systems For Energy-Efficient Built Environments*. ACM, 2015.

[PBCM15b] Marco Pritoni, Arka A Bhattacharya, David Culler, and Mark Modera. A method for discovering functional relationships between air handling units and variable-air-volume boxes from sensor data. In *BuildSys*, pages 133–136. ACM, 2015.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[PC15] Miguel Padilla and Daniel Choinière. A combined passive-active sensor fault detection and isolation approach for air handling units. *Energy and Buildings*, 99:214–219, 2015.

[PCC15] Miguel Padilla, Daniel Choinière, and José A Candanedo. A model-based strategy for self-correction of sensor faults in vav air handling units. *Science and Technology for the Built Environment*, (just-accepted):00–00, 2015.

[Pea09] Judea Pearl. *Causality*. Cambridge university press, 2009.

[Pin99] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta numerica*, 8:143–195, 1999.

[PMSW16] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael Wellman. Towards the science of security and privacy in machine learning. *arXiv preprint arXiv:1611.03814*, 2016.

[pos] PostGIS. https://postgis.net/, last visited: 11-01-2019.

[PPHM09] Mark Palatucci, Dean Pomerleau, Geoffrey E Hinton, and Tom M Mitchell. Zero-shot learning with semantic output codes. In *Advances in neural information processing systems*, pages 1410–1418, 2009.

[PVG+11a] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[PVG+11b] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[qud] QUDT. http://qudt.org/, last visited: 11-01-2019.

[RBE+17] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid training data creation with weak supervision. *Proceedings of the VLDB Endowment*, 11(3):269–282, 2017.

[RDFa] RDF Concepts Namespace. http://www.w3.org/1999/02/22-rdf-syntax-ns#.

[RDFb] RDF Schema Namespace. https://www.w3.org/2000/01/rdf-schema#, last visited: 11-01-2019.

[RLPZ19] Sowmya Ravidas, Alexios Lekidis, Federica Paci, and Nicola Zannone. Access control in internet-of-things: A survey. *Journal of Network and Computer Applications*, 144:79–101, 2019.

[RPT15] Bernardino Romera-Paredes and Philip HS Torr. An embarrassingly simple approach to zero-shot learning. In *ICML*, pages 2152–2161, July 2015.

[RR09a] Lev Ratinov and Dan Roth. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning*, pages 147–155. Association for Computational Linguistics, 2009.

[RR09b] Lev Ratinov and Dan Roth. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning*, pages 147–155. Association for Computational Linguistics, 2009.

[San98] Ravi S Sandhu. Role-based access control. In *Advances in computers*, volume 46, pages 237–286. Elsevier, 1998.

[SB02] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 269–278. ACM, 2002.

[SBCH06] Jeffrey Schein, Steven T Bushby, Natascha S Castro, and John M House. A rule-based fault detection method for air handling units. *Energy and Buildings*, 38(12):1485–1492, 2006.

[SBV97] JM Sousa, R Babuška, and HB Verbruggen. Fuzzy predictive control applied to an air-conditioning system. *Control Engineering Practice*, 5(10):1395–1406, 1997.

[SC08] Burr Settles and Mark Craven. An analysis of active learning strategies for sequence labeling tasks. In *Proceedings of the conference on empirical methods in natural language processing*, pages 1070–1079. Association for Computational Linguistics, 2008.

[sch] schema.org. schema.org. schema.org. Accessed: 2019-10-30.

[SGMS12] David Sturzenegger, Dimitrios Gyalistras, Manfred Morari, and Roy S Smith. Semi-automated modular modeling of buildings for model predictive control. In *BuildSys - ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 99–106. ACM, 2012.

[SHH+11] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. Fedx: a federation layer for distributed query processing on linked open data. In *Extended Semantic Web Conference*, pages 481–486. Springer, 2011.

[Shi00] Hidetoshi Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of statistical planning and inference*, 90(2):227–244, 2000.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[Sia14] Pierluigi Siano. Demand response and smart grids—a survey. *Renewable and sustainable energy reviews*, 30:461–478, 2014.

[sky] SkyFoundry. https://skyfoundry.com/, last visited: 11-01-2019.

[SM86] Gerard Salton and Michael J McGill. Introduction to modern information retrieval. 1986.

[spa] SPARQL Query Language. https://www.w3.org/TR/rdf-sparql-query/, last visited: 11-01-2019.

[SPG14a] Anika Schumann, Joern Ploennigs, and Bernard Gorman. Towards automating the deployment of energy saving approaches in buildings. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*, pages 164–167. ACM, 2014.

[SPG14b] Anika Schumann, Joern Ploennigs, and Bernard Gorman. Towards automating the deployment of energy saving approaches in buildings. In *BuildSys*, 2014.

[SPS17] Georg Ferdinand Schneider, Pieter Pauwels, and Simone Steiger. Ontology-based modeling of control logic in building automation systems. *IEEE Transactions on Industrial Informatics*, 13(6):3350–3360, 2017.

[SS75] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[SSK16] Markus D Steinberg, Sirko Schindler, and Jan Martin Keil. Use cases and suitability metrics for unit ontologies. In *International Experiences and Directions Workshop on OWL*, pages 40–54. Springer, 2016.

[SST18a] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Situational access control in the internet of things. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1056–1073. ACM, 2018.

[SST18b] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Situational access control in the internet of things. In *SIGSAC*, 2018.

[TC98] Jason Teeter and Mo-Yuen Chow. Application of functional link neural network to hvac thermal dynamic system identification. *Industrial Electronics, IEEE Transactions on*, 45(1):170–176, 1998.

[Tib96] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[TMLK09] Alessandra Toninelli, Rebecca Montanari, Ora Lassila, and Deepali Khushraj. What's on users' minds? toward a usable smart phone security model. *IEEE Pervasive Computing*, 8(2):32–39, 2009.

[tur] Turtle. https://www.w3.org/TR/turtle/, last visited: 11-01-2019.

[Uni15] United States Energy Information Administration (EIA). A Look at the U.S. Commercial Building Stock: Results from EIA's 2012 Commercial Buildings Energy Consumption Survey (CBECS). 2015. https://www.eia.gov/consumption/commercial/reports/2012/buildstock/, last visited: 12-08-2019.

[U.S] U.S. Energy Information Administration. Monthly energy review. https://www.eia.gov/totalenergy/data/monthly/#consumption. Accessed: 2019-10-30.

[vav03] VAV Terminal Control Applications Application Note. *Johnson Controls Technical Bulletin*, 2003.

[VCL95] GS Virk, JYM Cheung, and DL Loveday. Practical stochastic multivariable identification for buildings. *Applied mathematical modelling*, 19(10):621–636, 1995.

[VvRBT13] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2), 2013.

[WAA+12] James Weimer, Seyed Alireza Ahmadi, José Araujo, Francesca Madia Mele, Dario Papale, Iman Shames, Henrik Sandberg, and Karl Henrik Johansson. Active actuator fault detection and diagnostics in hvac systems. In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 107–114. ACM, 2012.

[WBD+11] Thomas Weng, Bharathan Balaji, Seemanta Dutta, Rajesh Gupta, and Yuvraj Agarwal. Managing plug-loads for demand response within buildings. In *BuildSys - ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 13–18. ACM, 2011.

[WBK+18] Weimin Wang, Michael R Brambley, Woohyun Kim, Sriram Somasundaram, and Andrew J Stevens. Automated point mapping for building control systems: Recent advances and future research needs. *Automation in Construction*, 85, 2018.

[WC02] Shengwei Wang and Youming Chen. Fault-tolerant control for outdoor ventilation air flow rate in buildings based on neural network. *Building and Environment*, 37(7):691–704, 2002.

[Web19] Web of Things Working Group. Web of things at w3c, 2019. Retrieved Dec 8, 2019 from ”https://www.w3.org/WoT/”.

[WNA13] Thomas Weng, Anthony Nwokafor, and Yuvraj Agarwal. Buildingdepot 2.0: An integrated management system for building analysis and control. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, pages 1–8. ACM, 2013.

[WSL+19] Zihan Wang, Jingbo Shang, Liyuan Liu, Lihao Lu, Jiacheng Liu, and Jiawei Han. Crossweigh: Training named entity tagger from imperfect annotations. *arXiv preprint arXiv:1909.01441*, 2019.

[YBG+18] Longqi Yang, Eugene Bagdasaryan, Joshua Gruenstein, Cheng-Kang Hsieh, and Deborah Estrin. Openrec: A modular framework for extensible and adaptable recommendation algorithms. In *WSDM*, pages 664–672, 2018.

[ZARP15] Nan Zhao, Matthew Aldrich, Christoph F Reinhart, and Joseph A Paradiso. A multidimensional continuous contextual lighting control system using google glass. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*, pages 235–244. ACM, 2015.

[ZM95] Ron Zahavi and Thomas J Mowbray. *The essential CORBA: systems integration using distributed objects*. Wiley, 1995.

[ZWW+11] Junping Zhang, Fei-Yue Wang, Kunfeng Wang, Wei-Hua Lin, Xin Xu, and Cheng Chen. Data-driven intelligent transportation systems: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 12(4):1624–1639, 2011.

[ZZ14] Min-Ling Zhang and Zhi-Hua Zhou. A review on multi-label learning algorithms. *IEEE transactions on knowledge and data engineering*, 26(8):1819–1837, 2014.