Self-Adapting Software for Cyberphysical Systems

Gabe Fierro

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-159
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-159.html

June 2, 2021

Self-Adapting Software for Cyberphysical Systems

by

Gabriel Tomas Fierro

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David E. Culler, Chair
Professor Joseph M. Hellerstein
Associate Professor Stefano Schiavon

Spring 2021

Self-Adapting Software for Cyberphysical Systems

Abstract

Self-Adapting Software for Cyberphysical Systems

by

Gabriel Tomas Fierro

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David E. Culler, Chair

The built environment — the buildings, utilities, infrastructure, cities and other constructed elements of the anthropocene — is becoming increasingly digitized. The complex array of equipment, sensors and other devices in these environments constitute *cyberphysical systems* which produce an incredible volume of data. However, this cyberphysical data is hard to access and understand because of the extreme heterogeneity and scale of the built environment: essentially every cyberphysical system is a custom-built "one-off" collection of equipment, devices and data sources that has been continually operated, retrofitted, expanded and maintained over years, decades and centuries.

This dissertation argues that existing barriers to widespread adoption of software-driven sustainable practices can in part be overcome through the adoption of rich, semantic metadata which enables the *mass-customization* of data-driven cyberphysical software. Applications will be able to query their environment for the contextual clues and metadata that they need to customize their own behavior and discover relevant data.

To realize this vision, this thesis proposes a linked-data ontology — Brick — which formally defines a graph-based data model for describing heterogeneous cyberphysical systems, and a set of ontology design principles for generalizing Brick to other domains. Brick models are created and maintained through a continuous metadata integration process also developed in the dissertation. New programming models are introduced which use graph-based metadata to implement self-adapting applications. Lastly, the thesis develops a novel data management platform, Mortar, which supports storing, serving and managing semantic metadata at scale. This demonstrates that standardized metadata representations of cyberphysical environments enable a fundamentally richer set of data-driven applications that are easier to write, deploy and measure at scale.

To my friends, family and colleagues.

Contents

Contents **ii**

List of Figures **iv**

List of Tables **viii**

1 Introduction **1**
1.1 Software for Cyberphysical Systems 2
1.2 Thesis Question 3
1.3 Challenges for Self-Adapting Software 4
1.4 Approach and Thesis Roadmap 5

2 Background **7**
2.1 Cyberphysical Systems in the Built Environment 7
2.2 Definitions 13
2.3 Metadata for Cyberphysical Systems 14
2.4 Ontologies and Linked Data 22
2.5 Summary 29

3 A Vision of Self-Adapting Software **30**

4 Managing Heterogeneity with Semantic Metadata **35**
4.1 Limitations of Existing Metadata Representations 37
4.2 Modeling Issues for Tag-Based Metadata 41
4.3 Ontology Design for Consistent Metadata 47
4.4 Brick+ Formal Implementation 55

5 Expressing Self-Adapting Software **65**
5.1 Using Metadata for Configuration 66
5.2 Programming Models for Self-Adapting Software 70
5.3 Evaluation of Staged Programming Model 78
5.4 Alternative Programming Models 85

6 Metadata Management for Self-Adapting Software **88**
6.1 Prior Work on Metadata Management 89
6.2 Extracting Semantic Metadata 91
6.3 Metadata Management Over the Building Lifecycle 96
6.4 Metadata Management Over Changing Semantics 107

7 Platform Design and Implementation for Self-Adapting Software **114**
7.1 Scaling Metadata Management 114
7.2 `reasonable`: Abstracting Ontology Management 117
7.3 Mortar: A Platform for Reproducible Building Science 123

8 Adoption and Impact **129**
8.1 Metadata Standardization and Semantic Interoperability 129
8.2 Open Data Research Efforts 133
8.3 Brick Consortium 135
8.4 Growing Adoption 136
8.5 Availability of Open Source Code 136

9 Conclusion **138**
9.1 Results and Contributions 138
9.2 Future Work 139
9.3 Reflections and Final Remarks 141

Bibliography **142**

List of Figures

1.1 Current practice: deploying software to different sites requires re-implementing and re-configuring the software for *each site* 3

2.1 Figure A-2 from ASHRAE's Guideline 36 [73], depicting a prototypical Variable Air Volume box with reheat capability. 8
2.2 Small, representative building consisting of an AHU (blue), 2 VAVs (green), and 2 HVAC zones consisting of rooms and lighting zones (red). 11
2.3 Screenshot of a recently-deployed ($\tilde{2}009$) building management system 12
2.4 Different metadata representations used over the course of a building's lifecycle . 15
2.5 Part of an IFC representation of a small, 2-story office building in Berkeley, CA 16
2.6 Part of an gbXML representation of a "big box" retail store. The AirLoop tag contextualizes a set of equipment for a particular segment of the HVAC system supplying air to a zone 18
2.7 Graphical representation of part of a Modelica model representing a VAV with reheat capability, sourced from the Buildings Library [150]. Modelica also supports exporting to a JSON-based format [98]. 19
2.8 Snippet of a document detailing point naming conventions for a particular BMS vendor 20
2.9 A damper position command point described using Haystack tags, as seen in the Carytown reference model 20
2.10 Simple, representative Brick model 21
2.11 Generic RDF *triple*, consisting of a subject and object (nodes in a graph) and a predicate (directed edge from subject to object) 23
2.12 Simple RDF graph — depicted both graphically and textually — describing a thermostat with an attached temperature sensor. 24
2.13 Example Turtle-formatted RDF graph for the Brick model depicted in Figure 2.10 26
2.14 Simple SPARQL query for the airflow setpoints and sensors associated with VAVs 27

3.1 Current practice: deploying software to different sites requires re-implementing and re-configuring the software for *each site* 30
3.2 Self-adapting software will query a metadata model of its environment in order to automatically configure a single implementation to operate on new sites. . . . 31
3.3 A high level vision of self-adapting software interacting with the built environment 34

4.1 The set of valid (blue + solid outline) and invalid (red + dashed outline) tagsets for a set of four tags. The class hierarchy is established from top to bottom;subclass relationships are indicated by arrows 45
4.2 An OWL-based ontology (right) encodes classes as named sets (left); a subclass relationship between two classes means that all the members/instances of the subclass are also members/instances of the superclass. 48
4.3 An example of a logical violation in an instance of a Brick+ model. 51
4.4 SHACL node shapes validating use of the Brick+ `hasPoint` and `isPointOf` relationships 52
4.5 A SHACL shape enforcing the two possible uses of the `isPartOf` relationship. . 53
4.6 An *erroneous* SHACL shape for defining two required points for VAVs 54
4.7 The correct SHACL shape for defining two required points for VAVs. 55
4.8 The three main class organizations for Brick+ entities 56
4.9 The Brick+ definition of a temperature sensor. 56
4.10 Part of the `brick:Temperature_Sensor` class definition from Figure 4.9, showing the unoptimized full implementation of the OWL intersection class 58
4.11 SHACL-AF implementation of tag→class inference for the Brick+ `Temperature_Sensor` class. Figure 9.2 in the Appendix contains the full expanded shape. 59
4.12 A subset of the Brick+ class lattice for sensors, showing the semantic properties which characterize each class 62
4.13 63

5.1 Logical workflow for application configuration and execution 66
5.2 Three different Brick models describing the same physical system. It is possible to author a single SPARQL query which will retrieve the names of the AHU and downstream rooms, independent of the specific structure of the graph. 69
5.3 A SPARQL query which finds the AHU and two rooms in each of the graphs in Figure 5.2 70
5.4 Architecture of an application written in the staged programming model 71
5.5 Two SPARQL queries used in the `qualify` step of a rogue zone detection application. 72
5.6 Dataframe definition in Python (left) and the resulting dataframes (right) . . . 73
5.7 View definition in Python (left) and resulting table (right) 74
5.8 A simple `clean` stage implementation in Python which filters out periods of time where either dataframe is missing a value 75
5.9 A `analyze` component implementation in Python for the rogue zone application which finds periods where the measured airflow is lower than the setpoint. . . . 75
5.10 A simple program written using the interactive model that finds regions of time when AHUs are both heating and cooling. 77
5.11 The SPARQL query implied by the execution of the program in Figure 5.10 . . 77
5.12 **Energy Baseline Calculation and Baseline Deviation Applications**: Predicted baseline using [92] plotted against actual building energy consumption. . 79

5.13 Heating coil and cooling coil valve commands over time for an AHU in a building demonstrating simultaneous heating and cooling. 82
5.14 A BAS [84] representation of a building's electrical, lighting, spatial and HVAC subsystems 85

6.1 Original Haystack entity from the Carytown reference model 92
6.2 Intermediate RDF representation of the Haystack entity; Haystack software-specific tags (e.g. `cur`, `tz`) are dropped. 93
6.3 Brick inference engine splits the entity into two components: the explicit point and the implicit outside damper equipment. 93
6.4 The distribution of the number of triples inferred per entity for each wrapper. . 95
6.5 The distribution of the *total* number of triples inferred by each wrapper. Note the log-scale on the X axis 96
6.6 Different tools for different stages: Many different metadata standards and technologies are applied over the course of a building's lifecycle, but are relatively siloed and thus non-interoperable. 97
6.7 Overview of the proposed approach: *wrappers* interface directly with existing metadata sources stored in local file systems, or accessed via file shares or networked services. Wrappers continuously publish inferred Brick metadata to a central server, which produces a unified model. 98
6.8 Example record published by the BuildingSync wrapper, showing the original metadata (`raw`) and the inferred Brick metadata (`triples`). 100
6.9 The phases of the reconciliation algorithm. The latest Brick metadata (far left) is stored by the integration server. 102
6.10 Example Brick metadata produced by BuildingSync and Project Haystack wrappers. The `rdfs:label` property denotes the original name or identifier of the entity in the metadata source. 104
6.11 The inferred unified metadata model for the triples in Figure 6.10. The most specific type is chosen for each entity, and that associated properties are carried through. 104
6.12 Visualization of the campus meter database demonstrating how the semantics change independent of the structure. Schema 1 was active until January 12th 2012 at which point Schema 2 becomes the active schema. 108
6.13 Two different versions of the NSF degree taxonomy. Colored and outlined nodes are involved in either a *split*, *merge* or *same* relationship across versions. 112

7.1 Logical architecture of `reasonable` and its interaction with other software components of Mortar 118
7.2 Two OWL 2 RL rules expressed in Datalog. `T` is the relation `(s,p,o)` corresponding to the triples in an RDF graph. 119
7.3 Definition of the `LIST[]` syntax in [100] and an example of a variadic Datalog rule that uses a variable-sized list. 119

7.4 The `cax-sco` rule implemented to take advantage of intermediate relations . . . 121
7.5 Comparing performance of `reasonable` with OWLRL and Allegro over more than 100 Brick models. 123
7.6 Histogram of number of data streams for all sites (μ =241). 124
7.7 Mortar platform 125

8.1 Technologies for design, operation, analytics, controls and modeling of buildings are siloed and rarely interoperable 134

9.1 Brick expression of the building and subsystems illustrated in Figure 2.2 157
9.2 Expanded RDF graph for the shape described in Figure 4.11. 158

List of Tables

4.1 An enumeration of the intended use and context of tags relating to heating and cooling, as given by the Haystack documentation. Note the differences in diction across compound tags, and how some compound tags could be assembled from more atomic tags. Some tags are used both for equipment and for points when equipment is modeled as a single point (such as VFDs, Fans, Coils) 42
4.2 Brick+ relationships for associating entities with each other 57
4.3 A set of semantic property name definitions 61

5.1 Applications: Brick LOC and App LOC indicate the lines of code needed to define the Brick queries and application logic, respectively. "% coverage" is what proportion of the testbed's buildings qualified for that application; the corresponding number of buildings is in the "# sites" column. 83

6.1 The results of merging multiple metadata models for two different sites, showing the diversity of the metadata available between the available metadata sources. The *% Contributed* percentages do not add up to 100% because the rest of the graph consists of inferred metadata not contained in any particular model. . . . 106
6.2 A subset of the split/merge/same relationships between the 2004 and 2010 versions of the NSF degree taxonomy in Figure 6.13 111

7.1 Count of streams and equipment available in the testbed data set, aggregated by type. AHU and VAV totals include related equipment such as fans and pumps. . 124
7.2 API operations supported on the Mortar platform. A `?` suffix indicates the parameter is optional. A `[parameter]` notation indicates the type is a list. 127

Acknowledgments

The work described in this dissertation is the product of years of collaboration, mentorship, friendship and support from a number of people.

I would like to thank my advisor, David Culler, for being so generous with his time, experience and wisdom in guiding me through my PhD. Even when I was an undergraduate working with one of his graduate students, David took the time to meet with me to discuss my research and, later, applications to graduate school. His ability to derive sharp insights from my mess of thoughts was an invaluable resource that continues to inspire me to be a better and more thoughtful researcher. I also owe a great deal to my other dissertation committee members, Joe Hellerstein, Stefano Schiavon and Marta González, who have provided crucial perspective and feedback on my dissertation work.

My involvement in research is due to the mentorship and guidance of Andrew Krioukov, who asked me if I wanted to "hack buildings" during a research mixer in 2012. Our collaborations and conversations underlie much of the work described in this thesis. Andrew, Stephen Dawson-Haggerty, Jay Taneja and Randy Katz and other members of the LoCal research group at Berkeley got me interested in the intersection of computer science and the built environment and have provided valuable advice and support over the years.

As a graduate student, I had the privilege of working with a thoughtful, friendly and talented group of people that all sat or worked together in 410 Soda as part of the Software Defined Buildings and Building, Energy and Transportation Systems research groups. Michael Andersen, Jack Kolb, Kaifei Chen, Sam Kumar, Moustafa AbdelBaky, Kalyanaraman Shankari and Hyung-Sin Kim were an endless source of ideas, discussions and advice — I have learned so much from each of you. Albert Goto has been a constant through it all: his friendship and support made many late nights and harrowing deadlines possible. The graduate student communities in the Berkeley CS department and the Computer Science Graduate Student Association have also been a wonderful support network over the years.

I have been extremely fortunate to work with very knowledgable, professional and caring administrative staff over my time at Berkeley. Shirley Salanio and Jean Nguyen in the department made sure I was on track to complete my degree, quickly answered any questions I had, and helped me navigate various processes. Kattt Atchley and Boban Zarkovich in the RISE lab always had the time to help me through reimbursements and getting paid, and I enjoyed our conversations about music and food.

Many thanks are due to my collaborators in the Brick community, who have helped me grow Brick into a recognized and increasingly adopted effort that is already influencing the industry. Jason Koh, Dezhi Hong, Shreyas Nagare, Yuvraj Agarwal, Mario Bergés, Ambuj Shatdal and Erik Paulson have each made unique and valuable contributions of their time, experience and ideas, and Brick is better for it. I am also indebted to my other colleagues who have collaborated with me on many grants, projects and papers: Therese Peffer, Paul Raftery, Carlos Duarte, Anand Prakash, Marco Pritoni, Michael Wetter, Sascha von Meier, Keith Moffat, Ed Arens, Hui Zhang, Joyce Kim, Tyler Hoyt and Carl Blumstein. Over

my PhD, I have been generously supported by the California Energy Commission, the U.S. Department of Energy, the National Science Foundation, Johnson Controls, Intel and SRC.

I want to thank my family — Mom, Dad, Michael, Ollie, Ben, Angelo and Monica — for providing respite, demonstrating interest in my research and keeping me sane over these years. To my friends not mentioned above, Brandon, Karoun, Nathan, Ava and Max: I would not have made it through this program without your friendship, humor and excellent taste in food and music. Lastly, to Sonia, my partner in everything: thank you for the unwavering support and compansionship. The best part of finishing my PhD is that I get to experience that with you.

Chapter 1

Introduction

The built environment — the buildings, utilities, infrastructure, cities and other constructed elements of the anthropocene — is becoming increasingly digitized. The complex array of equipment, sensors and other devices in these environments constitute *cyberphysical systems* which produce an incredible volume of data. This data enables a broad family of data-driven use cases across many sectors of the built environment enacting energy efficient and resilient operation: monitoring, analysis and grid-aware control of buildings [127, 148, 137, 111]; smart grids and distributed energy resources [132, 78]; and fault detection, predictive maintenance and advanced controls for water treatment [5, 135, 34]. Despite the benefits of these kinds of data-informed applications, most are not widely adopted due to the prohibitive cost of accessing and understanding cyberphysical data [57].

Cyberphysical data is hard to access and understand because of the extreme heterogeneity and scale of the built environment: essentially every cyberphysical system is a custom-built "one-off" collection of equipment, devices and data sources that has been continually operated, retrofitted, expanded and maintained over years or even decades. The processes implemented by cyberphysical systems are diverse, complex and can be enacted in a number of ways. Buildings alone contain different subsystems for heating, cooling, ventilation and transporting different kinds of water, air, refrigerant and other substances, in addition to fire, security, lighting and other facility management systems. Each of these subsystems can contain 10s or 100s of pieces of equipment, each monitored and controlled through a variety of sensors and actuators from different manufacturers and speaking different protocols.

Heterogeneity presents a challenge for the development and deployment of software in cyberphysical settings. Each cyberphysical system is so different that software must be written with specific knowledge of the structure, composition and function of the *particular* cyberphysical system at hand. However, there is no standard representation of these systems that can facilitate such an understanding. The cyberphysical data required by applications is poorly labeled, often following ad-hoc naming conventions that usually do not contain machine-readable information about the source, context or significance of the data. Data scientists already spend upwards of 40% of their time gathering, cleaning and understanding data [68]. This task is further impeded by a lack of standard representations that encode

this metadata in a structured manner [112, 64, 19, 4]. Without such representations, the rollout of energy efficiency and other data-driven measures involves customizing software implementations to each potential deployment site.

This thesis proposes a method for the *mass-customization* of data-driven cyberphysical software. Applications will be able to query their environment for the contextual clues and metadata that they need to customize their own behavior, i.e. to choose an appropriate algorithm to run or model to train based on the kind of cyberphysical system deployed and the data that is available. Such *self-adapting software* requires rich, semantic and machine-readable descriptions of the built environment that can represent the necessary information, regardless of the complexity and uniqueness of the cyberphysical system. This chapter summarizes how data is usually managed for cyberphysical systems, including current practices for metadata in the built environment. Then, the chapter identifies four key challenges to realizing self-adapting software and presents the central thesis statement and roadmap.

1.1 Software for Cyberphysical Systems

The intent of cyberphysical software is to extract data about the physical world in order to inform some decision-making process. Data may inform diagnostic or descriptive processes, such as dashboards displaying the current status of a system or alarms which detect faults or errors within the system. Data may also drive prescriptive processes that use models to predict when equipment will break or inform a control decision about how to achieve a certain goal. The digitization of cyberphysical systems, a phenomenon sometimes referred to as the "industrial internet of things", will increase both the variety and availability of data. This presents new opportunities for innovative uses of data across many sectors of the built environment: from building management to smart grids to renewable energy resources to water management and transportation.

At the same time, the characteristics of cyberphysical systems are such that making effective use of data requires a substantial investment of time, money and technology. The networked infrastructure that allows data to be collected and commands to be sent to actuators is designed around the constraints of the small, primitive controllers and sensors that are embedded in the physical world. Historically, in these settings, memory and storage space is at a premium. Any available space is mostly devoted to storing programmable logic, data registers and mechanisms necessary to implement an industrial communication protocol such as BACnet, Modbus, LonTalk or OPC. The protocol handles the receipt and delivery of data to and from the sensors and controllers in the environment. Metadata, the data that describes the data sources, is usually an afterthought.

Consequently, cyberphysical systems rarely support more than a simple text field for capturing the name or description of each data source. These names are often the *only* available descriptions of the data sources, and are sometimes the only available metadata about the entire cyberphysical system. The structure of these names is idiosyncratic and follow ad-hoc conventions and idioms which are site-specific and often inconsistently applied.

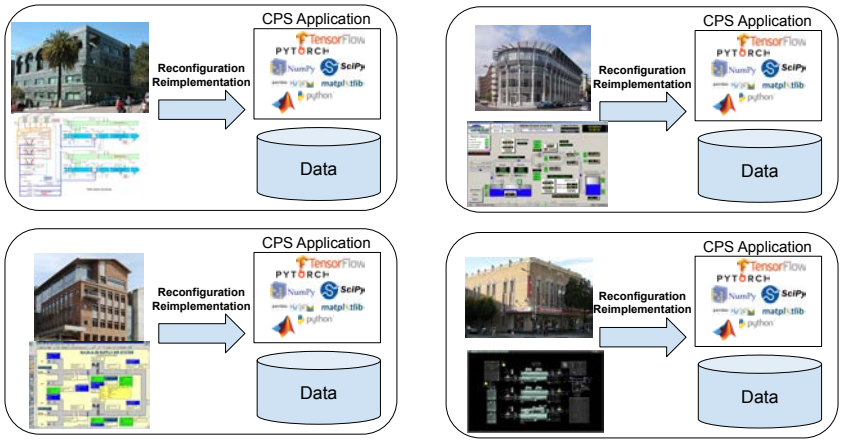


Figure 1.1: Current practice: deploying software to different sites requires re-implementing and re-configuring the software for *each site*

These names must serve as the digital "breadcrumbs" that allow an application developer to find the data sources they need. However, due to the lack of structured information, correctly and completely interpreting the names is a manual, time-consuming and error-prone task. This task is difficult to perform without an expert-level knowledge of how kinds of cyberphysical systems work and how the installer of the system decided to name the data [27, 25, 82, 131].

The lack of structured metadata has a significant impact on how software is written for cyberphysical systems. The complexity and uniqueness of cyberphysical systems means most software is authored for a particular cyberphysical system. Software uses hard-coded data source names to fetch required telemetry during operation. The algorithm, model and other aspects of the application are chosen with knowledge of how the deployment will look, what data sources and actuators and controllers are available, and what kinds of computational methods are the most appropriate. Most cyberphysical applications are written in this *non-portable* manner. Deploying a particular application in a variety of environments requires deriving an understanding of each environment, discovering the available data sources, and rewriting the application logic accordingly (Figure 3.1) [84]. The result is that most data-driven cyberphysical applications are not deployed at a large scale.

1.2 Thesis Question

Prior work calls for better, standard metadata which facilitates wider adoption of data-driven cyberphysical applications [19, 4]. However, this vision has yet to be realized. This thesis addresses fundamental barriers to enabling data-driven software for a variety of cyberphys-

ical systems by examining the relationship between how those systems are described and how software is written for them. Specifically, how can we construct and manage semantically rich, structured representations of complex cyberphysical systems which inform the automated customization and configuration of software? To answer this question, this thesis envisions a new paradigm, *self-adapting* software, in which software uses information about its cyberphysical environment to discover relevant data and execute appropriate application logic with minimal human intervention[1].

1.3 Challenges for Self-Adapting Software

There are four challenges to realizing self-adapting software for cyberphysical system.

Challenge #1: Descriptions of Heterogeneous Environments

The foundation of self-adapting software is a digital representation of the built environment that captures the information necessary for self-adaptation. These environments may contain different kinds of equipment, connected in different ways to enact different physical processes and are monitored and controlled in different ways. Despite this heterogeneity, software must still be able to discover salient details without any external or human-driven intervention. A key challenge is the definition of a metadata model that can effectively describe a variety of cyberphysical environments. Which principles should inform the design of the metadata model, and how should the model be expressed and queried?

Challenge #2: Effective Programming Model

Given the potential complexity of the cyberphysical systems at hand, it is not obvious how self-adapting software should be written. Traditional programming techniques for cyberphysical systems are still dominated by so-called "expert systems" which use if-then rules to express application logic, despite these practices falling out of mainstream use in other sectors [65]. This produces rigid programs that are constructed around the needs of a particular environment. The rich and detailed digital representations of cyberphysical environments will require a more expressive programming model. What does this programming model look like, and how can it make effective use of the available metadata?

Challenge #3: Metadata Management Over Time

Change is inevitable. Digital representations must be kept up-to-date with the cyberphysical environments they describe, and software must be able to make use of emerging information. Furthermore, the digital representations enabling self-adapting software must be created in the first place. A challenge for self-adapting software systems is how to bootstrap and

[1] More complete definitions of self-adapting software are examined in Chapter 3 and later in Chapter 5.

maintain the metadata representations, and how to detect and incorporate any changes in the environment into the model. What systems, protocols and techniques best support the management of cyberphysical metadata over time?

Challenge #4: Metadata Management At Scale

The value of self-adapting software is its potential to enable the adoption data-driven practices at the scale of thousands or even millions of environments, each with their own unique structures and representations. The databases, query processors and application runtimes that support self-adapting software must be built to accommodate heterogeneity at scale.

This dissertation argues that existing barriers to widespread adoption of software-driven sustainable practices can in part be overcome through the adoption of rich, semantic metadata. This presents challenges for the form, curation and maintenance of this metadata, and raises questions for how to effectively program against this metadata. This work proposes new semantic metadata management practices and techniques, a new programming model — self-adapting software — and novel data platforms which address these challenges.

1.4 Approach and Thesis Roadmap

To address these challenges and answer the thesis question, this thesis proposes the following:

- a linked-data ontology formally defining a graph-based data model for describing heterogeneous cyberphysical environments in buildings, and a set of ontology design principles for future ontologies targeting other sectors of the built environment;
- new programming models for using graph-based metadata to author self-adapting applications;
- techniques and a system for continuously maintaining metadata models of cyberphysical environments, even as those environments change;
- a data management system for storing, serving and managing metadata at scale.

These propositions are explored and evaluated over the following chapters.

Chapter 2 reviews the structure, composition and function of common cyberphysical systems that are potential targets of self-adapting software. The chapter provides an overview of current metadata practices for such cyberphysical systems, including recent academic and commercial work, and identifies existing holes where those practices fall short of enabling self-adapting software. The chapter summarizes ontologies and other linked data technologies that underlie the metadata model developed later.

Chapter 3 uses the background knowledge covered in Chapter 2 to more precisely illustrate the intended approach to self-adapting software described in this thesis. An architecture of a self-adapting software system is described, and the chapter outlines how each of the challenges above are addressed by components of the architecture.

Chapter 4 covers the design and implementation of a linked data ontology, Brick, which provides a semantically-rich and descriptive representation of cyberphysical systems in buildings. The chapter establishes crucial design principles that make Brick successful — interpretability, consistency and extensibility — and demonstrates how current popular metadata models fail to provide these properties. The chapter details how Brick implements these properties using formal logic.

Chapter 5 presents the design of two different programming models for self-adapting software: a staged execution model and an interactive execution model. The chapter explains how the models operate and how they may be implemented. The efficacy of the staged model is proved through the self-adapted execution of a family of representative data-driven applications over many real-world buildings.

Chapter 6 discusses how semantic metadata can be managed and maintained over time. The system presented in the chapter focuses on how this metadata can be inferred, mined or otherwise extracted from existing digital representations. These representations may be structured, semi-structured or unstructured. The chapter describes an algorithm which reconciles differences between the metadata extracted from several representations and merges these into a single, cohesive, model supporting self-adapting applications. The chapter presents the implementation of this approach in a real system and the evaluates its behavior on a set of real-world environments.

Chapter 7 presents two systems which support the storage, management and serving of metadata models and timeseries data to support self-adapting applications. `reasonable` is a software package which abstracts away the computationally expensive and time consuming components of ontology management behind a simple and performant interface. This facilitates the incorporation of linked data into existing databases. The second system, Mortar, is a data platform which hosts semantic metadata and timeseries data together to support data-driven and self-adapting analytics.

Chapter 8 reflects on how Brick, Mortar and other technologies and ideas described in this thesis are influencing data-driven practices in academia, industry and standards bodies.

Chapter 9 presents future work and conclusions, including a discussion on what other sectors of the built environment could be served by technologies like Brick and Mortar.

Chapter 2

Background

This chapter provides necessary background on cyberphysical systems in the built environment and the linked data technologies that will be leveraged to effectively describe them.

2.1 Cyberphysical Systems in the Built Environment

Cyberphysical systems, typified by the "Internet of Things" are engineered systems that are embedded in the physical world and provide networked sensing and actuation capabilities. These kinds of systems are ubiquitous in many facets of the built environment, including building operations and management, water transportation and treatment, power grid operations and transportation system management.

Above all, cyberphysical systems are characterized by extreme heterogeneity: any given cyberphysical system is an idiosyncratic assembly of equipment and other devices that has been custom-designed for a particular physical deployment. Cyberphysical systems are also constantly evolving. This is true both in the sense that recently designed cyberphysical systems often use newer and more efficient equipment and processes, but also in the sense that deployed cyberphysical systems experience repairs and retrofits that affect their composition and behavior.

This section provides a high-level overview of the kinds of cyberphysical systems that will be studied in this dissertation — primarily those in buildings — and reflect on how these systems characterize the challenges addressed by this work.

Cyberphysical Systems in Buildings

Buildings provide an excellent case study of the heterogeneity and complexity of cyberphysical systems. Modern buildings are often composed of several different *subsystems*, which each fulfill a distinct purpose.

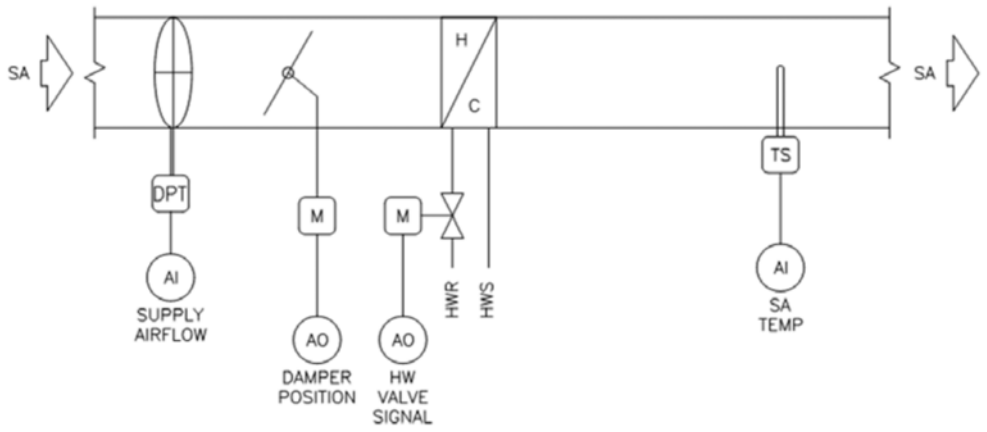


Figure 2.1: Figure A-2 from ASHRAE's Guideline 36 [73], depicting a prototypical Variable Air Volume box with reheat capability.

HVAC Subsystems

Heating, Ventilation and Air Conditioning (HVAC) subsystems manage the heating, ventilating, or air conditioning processes within a building. These systems vary widely in their construction and in the physical processes they use to achieve their goal, but there are several broad features that are common to most. The primary goal of HVAC systems is to heat, cool and ventilate spaces in the building. This can be accomplished using a number of physical mechanisms, the most common of which heats and cools the air supplied to different parts of the building (so-called "air-based HVAC"). This requires equipment to distribute and control the flow of the air, such as fans and dampers, which are often independently monitored and controlled using sensors embedded in the building and air distribution system. There are many different processes for heating and cooling air, most of which involve introducing a heat exchanger into the stream of air. Heat exchangers may be electric or gas-powered, or they may use hot or cold water which is supplied by distinct subsystems referred to as "hot water" and "cold water" loops. "Radiant systems" are an alternative design in which hot or cold water is pumped through pipes embedded in concrete slabs which then radiate or absorb heat to modulate the temperature in a space.

Each of these HVAC functions can be provided by a variety of equipment. Some major types of equipment include:

- Air Handling Units (AHUs): a large unit often containing dampers, fans or blowers, filters, heating and cooling elements that regulates and conditions air that is supplied to a building via connected ducts. AHUs are typically part of large HVAC installations.
- Terminal Unit: a smaller unit, typically downstream of an AHU, that regulates air volume, temperature or both and is connected directly to spaces in the building. A

Variable Air Volume box (VAV) is one common flavor of terminal unit (Figure 2.1) which incorporates a damper to regulate air volume.

- Rooftop Units (RTUs): a unit that contains all or most of the functionality of an AHU and VAV system, but in a single "packaged unit". Typically installed in small and medium-sized office buildings and is connected to a thermostat

The choice of equipment for a particular HVAC system depends on the needs of the building, including floor area, climate and projected occupancy. There are also many potential differences between equipment of similar classification (e.g. VAV). Manufacturers may incorporate different sensors, mechanisms, actuators and technologies into their equipment in order to provide more efficient operation or advanced control and fault detection strategies.

The behavior of HVAC systems is governed by a variety of control mechanisms, ranging from simple schedules to "sequences of operations" to advanced data- and model-driven optimal controllers. In all cases, HVAC systems make use of a variety of *sensors* which report the physical state of the system (such as the temperature and volume of air flowing to a room) and *setpoints* which constitute the targets or bounds of a controllers activity. Sequences of operations, such as those described in ASHRAE Guideline 36 [73], lay out precise rules for how to actuate components of an HVAC system in response to different relationships between sensed and measured values, and control setpoints.

Lighting Subsystems

Lighting subsystems serve many different roles, including exit lighting and fulfilling minimum levels of illumination for different occupant activities. These systems consist mostly of *luminaires*, which are devices consisting of a lighting element and enclosure. Luminaires are powered by *drivers*, which may regulate voltage and connect to a networked control system. Similar to HVAC, lighting control systems may operate on a simple schedule with manual control overrides (i.e. a wall switch for lighting, thermostat for HVAC). Advanced lighting systems operate by measuring illumination and sensing occupancy in the spaces served by one or more luminaires, and controlling lights and lighting fixtures to meet control targets or save energy.

Water Subsystems

Buildings contain several kinds of water systems, which typically consist of non-overlapping groups of equipment and are often siloed from one another.

- Chilled Water Loop: water is chilled through some mechanism (such as evaporation, or external coolant), and distributed throughout the building. In many HVAC systems, valves control water flow into coils placed in the air stream; modulating the valve in response to sensed water and air temperature allows fine-tuned control of downstream air temperature

- Hot Water Loop: water is heated through some mechanism and distributed throughout the building. Similar to chilled water loops, valves control the flow of hot water into coils placed in the air stream.
- Irrigation: water, from many potential sources, is distributed around a site or building in order to water plants. Water usage is metered and irrigation typically occurs on some schedule. Occasionally, sensors are deployed which provide feedback on the pH and hydration of soil
- Plumbing: water supplying bathrooms, kitchens and other human-facing building assets is also distributed and metered

Each of these systems incorporates a variety of sensors, setpoints, alarms and actuators which are used to drive controllers and user interfaces.

Electrical Subsystems

Electrical subsystems transform and transport electricity which is used to power the equipment involved in the above subsystems, as well as the multitude of electrical assets contained within the building. The lowest common denominator for electrical subsystems is the wiring supplying AC power to building assets, and the meter attached at the point between a building and the electric grid, which meters the amount of electricity flowing in either direction for the primary purpose of charging for usage.

Inside buildings, the electrical subsystem can be thought of as a tree with the main building meter as the root, and submeters and electrically-powered assets as the children. *Submeters* may be installed to independently monitor subsystems in the building; for example a submeter may measure all electricity used to power the HVAC system, or even just the chilled water loop.

The increasing penetration of renewable energy resources deployed at building sites — such as photovoltaic arrays and geothermal systems — in addition to the growing adoption of electric vehicles has introduced new families of supporting equipment which may be found at a site. These include inverters, voltage regulators and different kinds of batteries.

Spatial "Subsystems"

The set of logical, physical and administrative spaces that make up a building are not a "subsystem" in a traditional sense, but it is useful to treat these concepts as if they were a complex arrangement of equipment. Familiarly, buildings can be broken down into floors and rooms, but there are common concepts that do not fit cleanly into this formalization: what floor do stairs, elevators and atriums reside on? Is each cubicle in an office space its own room? Rooms can have many different uses, depending on the time of day, day of the week, time of the year or even the occupant or owner of a room.

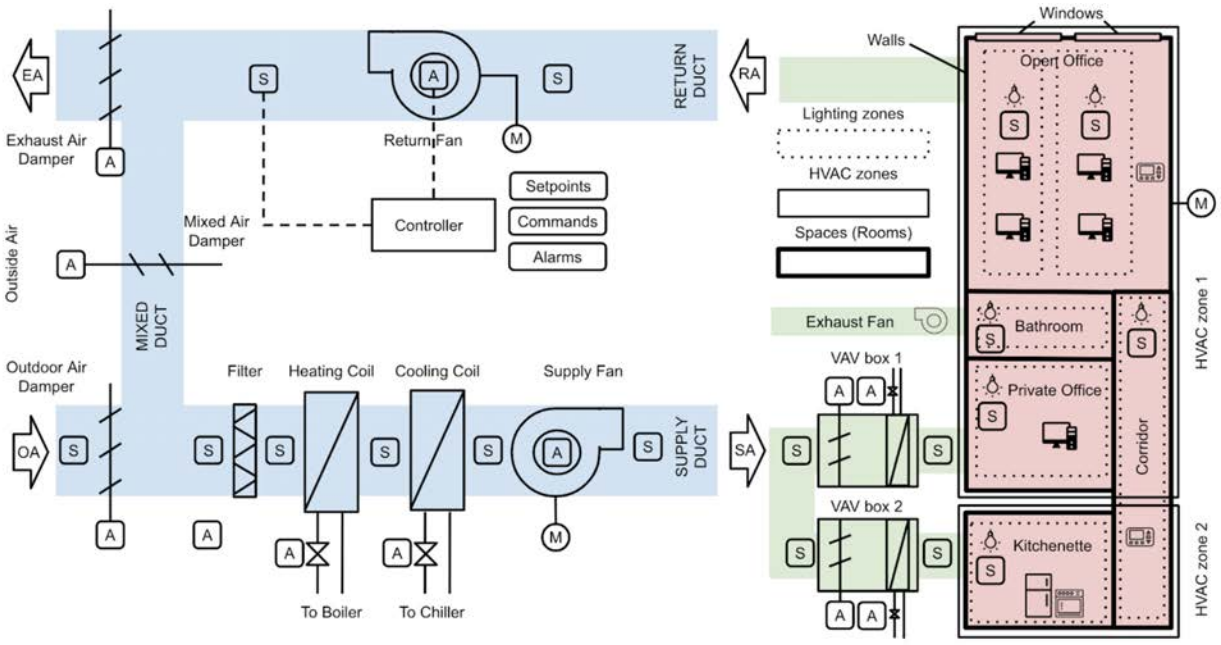


Figure 2.2: Small, representative building consisting of an AHU (blue), 2 VAVs (green), and 2 HVAC zones consisting of rooms and lighting zones (red).

Spaces in the building fall into logical groups — called *zones* — which are defined by their relationship to particular subsystems. An *HVAC Zone* is a collection of spaces that are all served by the same segment of an HVAC system, such as a VAV. A *Lighting Zone* is the collection of spaces that are illuminated by the same light bank. A *Fire Zone* is a subsection of the building which can be isolated from the rest of the building with fire wall and fire doors that help contain fires and assist in firefighting.

Zones and spaces interact in complex ways: it is not unusual for a space to contain multiple lighting zones, but be contained within a single HVAC zone. Additionally, spaces may be a subset of multiple HVAC zones.

In addition to the above, buildings also contain fire safety subsystems, security subsystems, navigation subsystems and many other assemblies of often digitized equipment, sensors and controllers that assist in the operation of a building. Each of these has their own array of dedicated, bespoke equipment and control systems.

Networked Monitoring and Control Systems

Building subsystems are controlled and monitored as networked systems. Networked sensors provide real-time data to both local and supervisory controllers. Local controllers, often placed on programmable logic controllers (PLCs), handle direct actuation of components such as valves, dampers and fans. This is in service of closed loop control, in which ob-

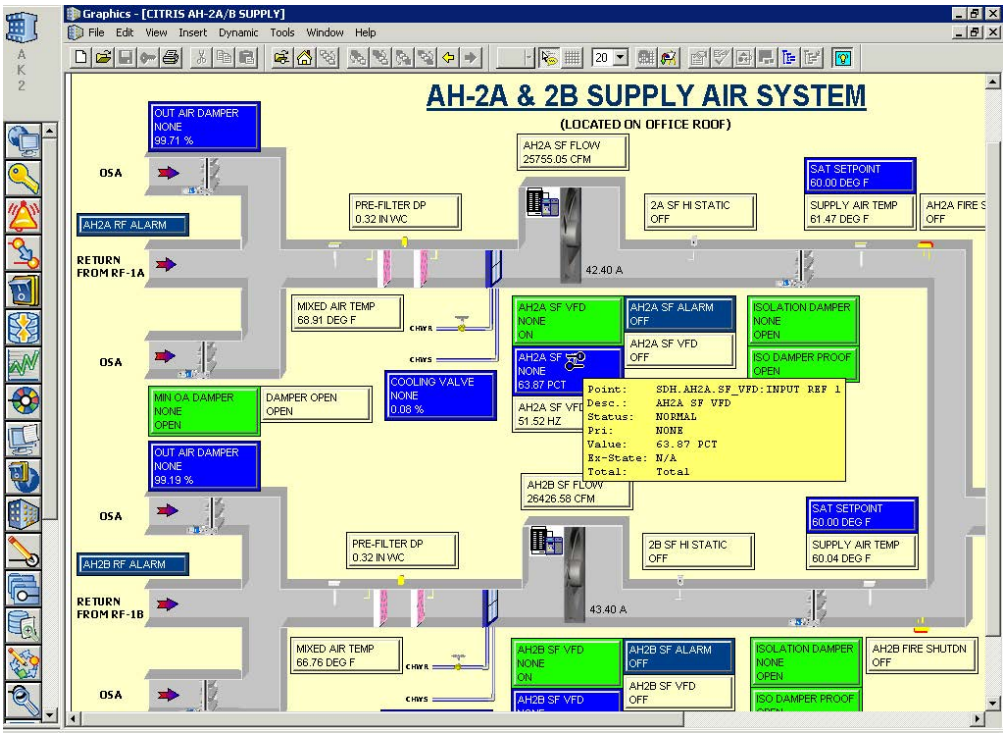


Figure 2.3: Screenshot of a recently-deployed ($\tilde{2}009$) building management system

servations from the environment influence future control actions in order to meet a control target, commonly called a *setpoint*. Common examples of closed loop control include temperature control: a controller calls for heating or cooling of supplied air in order to adjust the temperature.

Local controllers receive control targets from networked, supervisory controllers which send commands to devices over the network and monitor the status of devices and the values of sensors. Supervisory controllers commonly implement schedules, but can also coordinate the behavior of groups of devices to implement advanced control strategies.

Building Management Systems (BMS) are networked, digital control systems that implement supervisory control and provide programming environments and graphical user interfaces that facilitate the management of the building. These interfaces (Figure 2.3) illustrate the current state of equipment and offer configuration of schedules, direct override control of devices, and often simple trending and fault detection capabilities. A 2018 survey of U.S. commercial buildings found that 70% had digital HVAC controls and 50% had digital lighting controls [86]. This is up from the 10% of commercial buildings in the U.S. in 2005 [29]. However, the adoption of automated fault detection and building energy management systems was only 4%. BMS speak using protocols such as BACnet [14] and LonTalk.

Relevant to the work in this paper, each subsystem in a building may have its own BMS. These BMS are usually installed at different times, by different companies, and may

speak different protocols. Systems deployed in this way are said to be *siloed*. This presents integration challenges for any users of the system — human or software.

Supervisory Control and Data Acquisition (SCADA) systems are a common example of digital control systems that are not BMS. SCADA systems speak protocols such as Modbus [96], and are commonly found implementing industrial processes such as water management and manufacturing.

In contrast with the planned and designed nature of BMS and SCADA are the networked devices typical of the *Internet of Things*. These devices are deployed in a piecemeal and ad-hoc fashion, but implement a wide array of functionality and are being deployed at an increasing rate. Examples of IoT devices include temperature, CO2 and other sensors, thermostats, lights, plug load controllers, blenders, televisions, refrigerators and even beds and toasters. The promise of IoT is in ensembles of smart devices that can collaborate to save energy, increase comfort, or provide other automated conveniences. However, in practice, the IoT is plagued by fragmentation at the physical network level (devices may speak Bluetooth, WiFi, Zigbee, Enocean, Thread or any one of a number of other physical layers), network level and application level; this results in even more dramatic siloing than is found within the BMS domain. Industrial collaborations such as Connected Home Over IP [161] aim to standardize the application layer so that devices from different manufacturers may communicate.

2.2 Definitions

The following terms are used throughout the thesis and are defined here for clarity.

Definition 2.2.1 (Metadata)**.** *Metadata for cyberphysical systems comprises the digital representations and descriptions of the structure, composition, identity and functionality of cyberphysical systems, as well as their composing elements.*

Definition 2.2.2 (Metadata Representation)**.** *A metadata representation is a data model, schema, standard or convention which defines the structure, syntax and/or semantics of the metadata that can be expressed.* Structure *is the organization of metadata into data structures, objects and relationships.* Syntax *is the encoding and rules for how that metadata can be expressed and communicated.* Semantics *is the use of logical rules and statements to encode the "meaning" of metadata.*

Definition 2.2.3 (Metadata Model)**.** *A metadata model is an* instance *of a metadata representation. It contains the metadata for a particular site, deployment or other scoped collection of "stuff".*

Definition 2.2.4 (Tag)**.** *A tag is an atomic fact, annotation or attribute consisting of and identified by a simple textual label.*

Definition 2.2.5 (Entity)**.** *An entity is an abstraction of any physical, logical or virtual item; the actual "things" in a building.*

Definition 2.2.6 (Tag Set)**.** *A tag set is an (unordered) set of tags associated with an entity.*

Physical entities are anything that has a physical presence in the world. Examples are mechanical equipment such as air handling units, variable air volume boxes, luminaires and lighting systems, networked devices like electric meters, thermostats and electrical vehicle chargers, and spatial elements like rooms and floors.

Virtual entities are anything whose representation is based solely in operational software, such as a BMS or SCADA system. Examples are sensing and status points that allow software to read the current state of the world (such as the current temperature of air, the speed of a fan, or the energy consumption of a space heater), and actuation points which allow software to write values (such as temperature setpoints or the brightness of a lighting fixture).

Logical entities are those entities or collections of entities that are defined by a set of rules. Examples are HVAC zones and Lighting zones. Concepts such as class names and tags also fall into this category.

2.3 Metadata for Cyberphysical Systems

It has been said that "All models are wrong, but some are useful" [28]; this is as true for digital representations of cyberphysical systems as it is for the statistical models that are the subject of the original statement. The only "complete" or "correct" representation of a cyberphysical system is one that is 1-1 with both the digital ("cyber") and physical worlds. The intractability of such a model means that the metadata representations that are the study of this thesis are necessary simplifications. The effective design of a metadata representation of a building is rooted in choosing appropriate simplifications and abstractions that best support the intended use cases [54]. This section reviews prevailing approaches for metadata for cyberphysical systems and discusses their advantages, shortcomings and trade-offs.

Prevailing Metadata Representations for Buildings

Metadata representations for buildings capture different perspectives and levels of detail on buildings, their control systems, their subsystems, and the components therein. The intended use cases of a metadata representation inform the particular structure, syntax and semantics of the representation. One convenient way to categorize these perspectives is through the lens of the *building lifecycle*.

A Building Lifecycle Framing

The lifecycle of a building can be separated into several stages, each with their own participants, stakeholders and — importantly — metadata representations. Most existing research

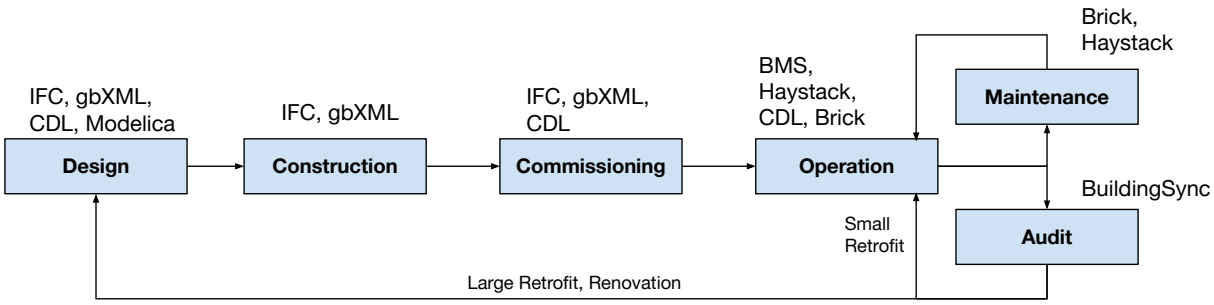


Figure 2.4: Different metadata representations used over the course of a building's lifecycle

into the building lifecycle uses a two or three stage categorization [143, 125] consisting of a design and construction phase, an operational phase, and a demolition or end-of-life phase. This discussion further subdivides the building lifecycle in order to call attention to the different metadata representations used.

Figure 2.4 presents a building lifecycle, with each stage represented as a box with commonly-used metadata representations. Briefly, these stages are:

- **Design**: During the design phase of a building, decisions are made about the layout, architecture and composition of the building and its subsystems, given specifications about the intended use and occupancy of the building.
- **Construction**: During the construction phase of a building, the design specifications from the prior phase are executed and realized in the physical world.
- **Commissioning**: During the commissioning phase of a building, sequences of operations for building subsystems are designed and implemented, and the building is measured to evaluate whether it meets the design specification as laid out in the design phase.
- **Operation**: The operational phase of the building consists of the use of the building and the operation and management of its subsystems. Typically the operational phase of the building is the longest.
- **Maintenance**: Buildings may undergo maintenance when the building is not operating as expected, for example due to broken equipment.
- **Auditing**: (Energy) audits are formal processes that evaluate the performance of a building relative to external standards or guidelines. Audits are typically conducted at regular intervals of a building's lifespan. For example, buildings in New York City over 50,000 square feet must be audited every 10 years [159].

```
#114= IFCPOSTALADDRESS($,$,$,$,('Enter address here'),$,'2087 Addison St','Berkeley','',
    'CA 94704');
#118= IFCBUILDING('1W9$dXIRj77hKpdvvBeVIz',#41,'',$,$,#32,$,'',.ELEMENT.,$,$,#114);
#124= IFCCARTESIANPOINT((0.,0.,-20.));
#126= IFCAXIS2PLACEMENT3D(#124,$,$);
#275062= IFCRELCONTAINEDINSPATIALSTRUCTURE('2p7RyAcN5DYP3YLiEj0ALn',#41,$,$,(#249085),#3066);
#269932= IFCSITE('1W9$dXIRj77hKpdvvBeVI_',#41,'Default',$,'',#269931,$,$,.ELEMENT.,
    (37,52,16,47363),(-122,-16,-6,-549682),0.,$,$);
#131= IFCAXIS2PLACEMENT3D(#6,$,$);
#132= IFCLOCALPLACEMENT(#32,#131);
#133= IFCBUILDINGSTOREY('1W9$dXIRj77hKpdvwqNWjI',#41,'Level 1',$,$,#132,$,'Level 1',.ELEMENT.,0.);
#135= IFCCARTESIANPOINT((0.,0.,17.1666666666667));
#137= IFCAXIS2PLACEMENT3D(#135,$,$);
#138= IFCLOCALPLACEMENT(#32,#137);
#139= IFCBUILDINGSTOREY('1W9$dXIRj77hKpdvwqNYsM',#41,'Level 2',$,$,#138,$,'Level 2',.ELEMENT.,
    17.1666666666667);
#141= IFCAXIS2PLACEMENT3D(#6,$,$);
#142= IFCLOCALPLACEMENT(#132,#141);
#144= IFCCARTESIANPOINT((3.55271367880050E-15,-8.88178419700125E-15));
#146= IFCAXIS2PLACEMENT2D(#144,#23);
#147= IFCRECTANGLEPROFILEDEF(.AREA.,$,#146,9.33333333333333,14.5116423880952);
```

Figure 2.5: Part of an IFC representation of a small, 2-story office building in Berkeley, CA

Issues uncovered during maintenance or auditing may result in *retrofits*, which may be as minor as updating control sequences and installing new equipment, or as major as adding/removing rooms or replacing the entire HVAC system. Retrofits experience the same broad phases of the building lifecycle, and often make use of the same metadata representations.

Building Information Modeling

Building information modeling (BIM) is a broad category of metadata representations that are designed to capture and communicate the properties of buildings, subsystems and constitutent equipment that are relevant for the design and construction phases of the building lifecycle. BIM representations commonly store the geometry of the building and its components, but can also capture the layout and connection of pipes, wires and other connections between equipment. The two major metadata standards for BIM are IFC and gbXML.

Industry Foundation Classes

Industry Foundation Classes (IFC) [1] is a standard format and data model for the exchange of data related to the design and construction of a building. This includes equipment and other assets, basic telemetry, wiring and connections between equipment, and the geometry and architecture of the building and its components. The IFC standard describes many common types of HVAC and lighting equipment as well as the sensors dispersed throughout the building.

IFC models do not capture the context of the equipment contained within. For example, an IFC model will contain a representation of the many fans in a particular building, but will not directly represent the configuration of those fans. The representation of a fan installed in a *supply* configuration (blowing air *into* a zone) will be indistinguishable from the representation of a fan installed in an *exhaust* configuration (blowing air *out of* a zone). A human expert looking at the IFC model will be able to differentiate the two configurations based on their familiarity with supply and exhaust systems, but the IFC model itself does not label the fans as such.

IFC models are expressed in the EXPRESS format (Figure 2.5), which presents challenges for integration with external tools. There are current efforts to develop more usable representations of IFC models, such as using RDF and OWL [115].

Green Button XML

Green Button XML (gbXML) [60] is an XML-based data model for the exchange of BIM data. gbXML focuses on capturing the 3D geometry of the equipment and spaces in a building. In constrast to other BIM representations like IFC, gbXML provides contextual information about a building's components by grouping related equipment and spaces together (Figure 2.6). The XML tags, attributes and enumerated values defined by the gbXML standard also include more contextual information about how building components are connected and configured.

Modelica

Modelica is a declarative, equation-based modeling language used to describe engineered systems [97]. Modelica decomposes systems into modular objects — defined by the user or in external libraries — which are coupled to each other to form systems (see Figure 2.7). The connections between Modelica components are objects themselves, and can have properties attached to them. Connections represent input and output ports for control signals as well as physical ports, such as the valve flange through which fluid flows. A model created with Modelica can be used for simulations of different system configurations, control strategies and equipment. Modelica's use is not limited to buildings, and many different libraries are available which define modular components for different domains. The Modelica Buildings Library [150] defines common component and system models for building and district energy and control systems. However, it is possible to define new or extend existing Modelica components for buildings that are not contained in the library.

Control Description Language

The Control Description Language (CDL) is a subset of Modelica used to express control sequences for building automation systems in a digital and vendor-independent manner [149]. By standardizing the representation of control sequences, CDL aims to facilitate the design, specification, deployment and verification of common and emerging control strategies [151].

```
<AirLoop id="West2" systemType="VariableAirVolume" controlZoneIdRef="aim19233">
   <Name>Air loop</Name>
   <Description>West exterior system using schedule 'FanSch-17' on the bottom floor
       interior system using schedule 'FanSch-17' on the bottom floor </Description>
   <TemperatureControl>
     <MinTemp unit="C">11.1</MinTemp>
     <MaxTemp unit="C">48.9</MaxTemp>
   </TemperatureControl>
   <AirLoopEquipment id="West2-Equip-8" equipmentType="Duct">
     <Name>3.5" wg static VAV duct system</Name>
     <Description>3.5" wg static pressure duct system</Description>
     <DeltaP unit="Pascals">-870.799987792969</DeltaP>
     <Cost>4.1</Cost>
   </AirLoopEquipment>
   <AirLoopEquipment id="West2-Equip-2" equipmentType="Fan">
     <Name>Std VAV Fan with variable speed drive (VSD)</Name>
     <Description>Forward curved fan and premium efficiency motor typically used
                 for larger packaged and AHU equipment.</Description>
     <OperationSchedule scheduleIdRef="FanSch-17" />
     <MinFlow unit="Fraction">0.3</MinFlow>
     <DeltaP unit="Pascals">870.799987792969</DeltaP>
     <Power powerType="Electricity" useType="Both" />
     <Efficiency standardsType="NEMA" operationType="HeatingAndCooling"
                                 efficiencyType="MotorEff">0.93</Efficiency>
     <Efficiency standardsType="No Defined" operationType="HeatingAndCooling"
                                 efficiencyType="MotorEff">0.75</Efficiency>
     <Control controlType="Fan" stages="Variable " />
     <Cost>0.733</Cost>
   </AirLoopEquipment>
</AirLoop>
```

Figure 2.6: Part of an gbXML representation of a "big box" retail store. The AirLoop tag contextualizes a set of equipment for a particular segment of the HVAC system supplying air to a zone

CDL sequences can be integrated with building models expressed in Modelica, permitting the simulation and evaluation of control sequences before they are deployed on actual hardware In fact, the Modelica Buildings library already contains CDL representations of high performance HVAC control sequences. CDL-driven simulations enable comparing the performance different control sequences, testing their correct specification, and commissioning their correct implementation in buildings.

Building Management System Metadata

In contrast with the structured and standardized representations detailed above, the metadata found in building management systems largely consists of ad-hoc text labels for the telemetry "points" that are exposed from the underlying digital control system. These labels are often rendered on a web interface (see Figure 2.3) that is exposed to a building or facility manager. Because these labels are simple strings, their structure is sometimes (but not always) informed by a site- or vendor-specific naming convention often consisting

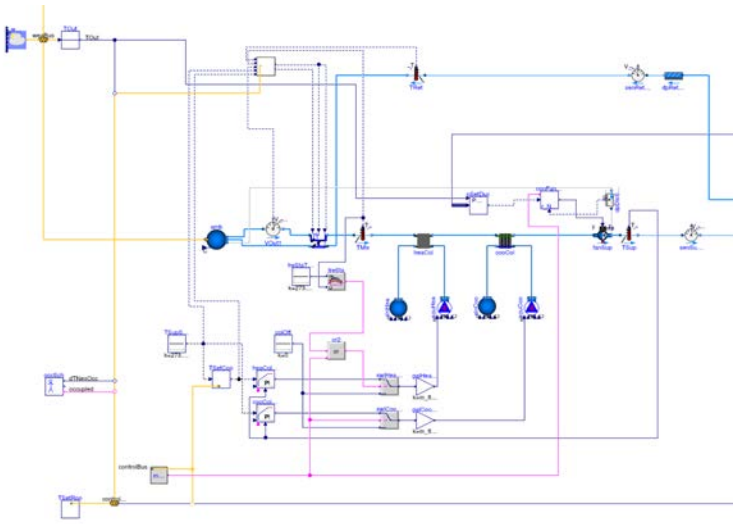


Figure 2.7: Graphical representation of part of a Modelica model representing a VAV with reheat capability, sourced from the Buildings Library [150]. Modelica also supports exporting to a JSON-based format [98].

of abbreviations describing related components or relevant features (Figure 2.8). These labels have several limitations. First, the lack of a standard structure (between BMS, and sometimes even within a BMS) means that it is difficult to extract any metadata about the building in a structured manner. Conventions and idioms are rarely written down, are inconsistent between systems (even those from the same vendor), or may not exist at all for a particular site. Second, the point labels only exist for sources of telemetry and control signal sinks, and do not describe the actual structure of the building subsystem they are monitoring or controlling.

As an example, consider a real BMS point label from the Computer Science department building on the UC Berkeley campus: `SODA1R410_ART` [25]. This label contains several pieces of metadata. First, the type of the point is communicated by the `ART` substring: a room air temperature sensor. The name of the room is given by `R410`; `R` identifies that the identifier `410` is for a room. Next, the name of the AHU supplying air to the room is denoted by `A`, and the identifier points out that AHU `1` is upstream of this room. Lastly, `SOD` indicates which building the sensor, room and AHU are located in.

Project Haystack

Project Haystack[1] is a commonly-used tag-based metadata scheme for buildings that uses atomic tags and key-value pairs in place of unstructured labels to describe buildings, equipment and points [118]. A Haystack model of a building consists of a set of documents — one

[1]Commonly abbreviated as "Haystack"

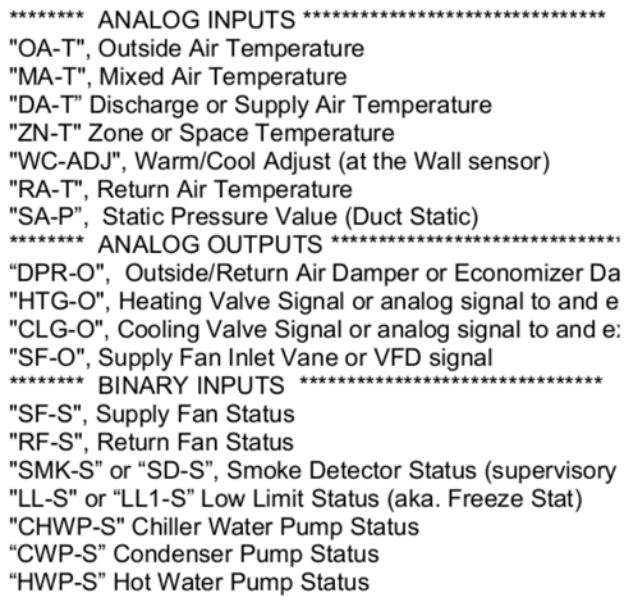


Figure 2.8: Snippet of a document detailing point naming conventions for a particular BMS vendor

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

Figure 2.9: A damper position command point described using Haystack tags, as seen in the Carytown reference model

for each equipment or point entity associated with the building. Each document consists of a number of "marker tags" and "value tags". "Marker tags" indirectly describe the entity's type and behavior by virtue of their association. "Value tags" which are key value pairs that communicate properties of entities and links between entities. Figure 2.9 contains an example of one of these documents; field names to the left of the colon are tag names. Tags with a ✓ value are marker tags; all other tags are value tags.

The intent of marker tags is to provide a "type" for an entity: AHUs have the `ahu` and `equip` tags, zone temperature sensors have the `zone`, `temp` and `sensor` tags. The current release of Haystack defines a dictionary of over 200 marker tags, which can be combined and

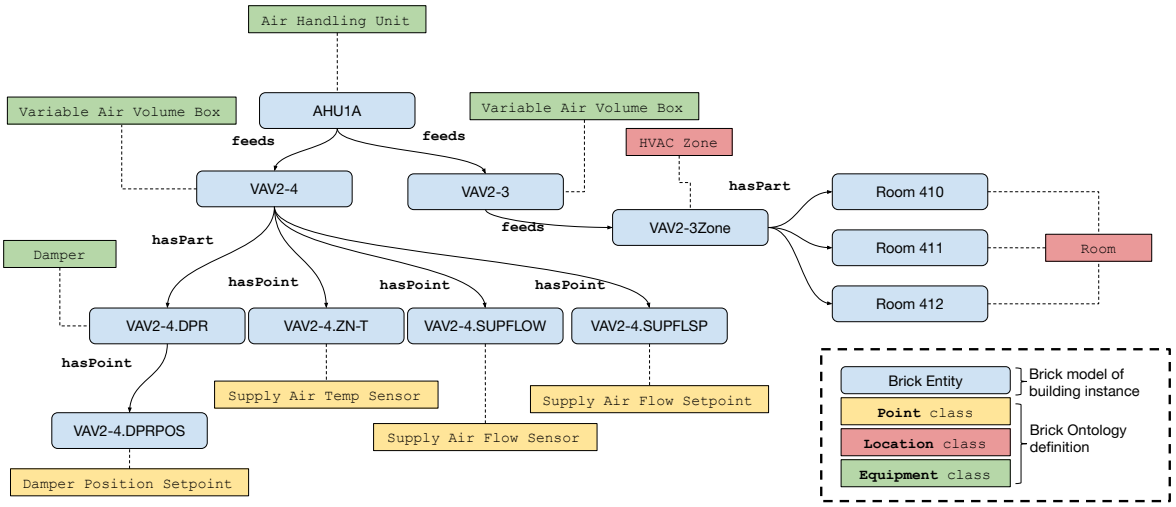


Figure 2.10: Simple, representative Brick model

re-combined to describe different entity types. Users of Haystack can also define custom tags to fill the gaps in the pre-defined dictionary. The resulting flexibility of the model comes at the cost of consistency and interpretability; these issues will be discussed in-depth in Chapter 4.

Value tags associate scalar values with entities. The `curVal` value tag associates the current value of a point with an entity; other value tags capture timezones, engineering units and geo-coordinates. An important kind of value tag is a "ref tag". A ref tag describes a relationship between two Haystack entities. The particular tag name describes the kind of entity that the relationship points to. Haystack's tag dictionary defines several ref tags for common use: `equipRef` associates a Haystack point entity (indicated by the `point` tag) with an equipment entity (indicated by the `equip` tag), `ahuRef` associates a VAV or chiller entity with an AHU, `elecMeterRef` associates a piece of equipment with an electric meter, and `siteRef` associates an entity with a site.

Brick: Semantic Metadata for Buildings

Brick [16, 54], developed as part of this thesis work in collaboration with others, is a semantic metadata ontology describing building entities and the relationships between them for the purpose of enabling data-driven applications. Brick defines a broad set of classes capturing common types of equipment (from HVAC, lighting and electrical subsystems), locations and points. The set of classes are informed by an empirical study of the concepts that are referenced in real building applications [26].

Brick also defines a minimal set of relationships that capture the semantics of how entities relate to one another. Broadly, these relationships encompass *composition* (how equipment

and sytems are made up of other components), *topology* (how equipment are connected within a system) and *telemetry* (how data sources relate to equipment and systems).

A Brick model of a building is a directed labeled graph where the nodes represent entities and the edges represent relationships between entities. Entity types and the relationships between entities use the vocabulary defined by Brick. Figure 2.10 contains an example of a small Brick graph, comprising an AHU and two VAVs. One VAV has an internal damper, and the other VAV feeds an HVAC zone consisting of two rooms. The model also incorporates telemetry associated with each of the equipment.

2.4 Ontologies and Linked Data

Linked data models provide a means of representing entities and their properties and relationships in a much more flexible and extensible manner than traditional entity-relationship data models: this constitutes a directed, labeled graph. This kind of flexibility is crucial for modeling cyberphysical systems, which are heterogeneous in both their content (what kinds of entities are present) and structure (what kinds of relationships exist between entities). A further benefit of linked data models is that their semantics can be formalized through the use of an ontology. In a linked data context, an *ontology* is a set of rules and axioms which programmatically capture what kinds of information can be represented and derived from a given linked data model. The process of deriving new information from the application of rules to a linked data model is commonly referred to as *materialization* or *inference*. This section presents relevant background on the RDF data model for knowledge graphs, the OWL family of ontology languages and their companion technologies which together constitute the "semantic web stack".

RDF Data Model

The Resource Description Framework (RDF) data model represents directed, labeled *multi*graphs. Graph nodes correspond to *resources*; edges are directed and labeled with the name of a *property*. This implements a flexible model in which the value of an entity's property can be another entity.

An RDF graph is defined as a set of *triples*. A triple is a 3-tuple specifying the content of a `<node, edge, node>` segment of a graph. The first element of a triple is the *subject* resource; the third element of a triple is the *object* resource; the second element of a triple is the *predicate* which labels the relationship between the subject and the object from the subject's perspective (hence the directed nature of the graph). A triple thus has the form `<subject, predicate, object>`. Figure 2.11 illustrates a generic triple both in a graphical form and in a typical textual representation. RDF models can be serialized into a variety of formats, including XML, JSON-LD, N-Triples and Turtle. In this thesis, figures containing RDF graphs will either be depicted as a node-edge graph or in the Turtle format.

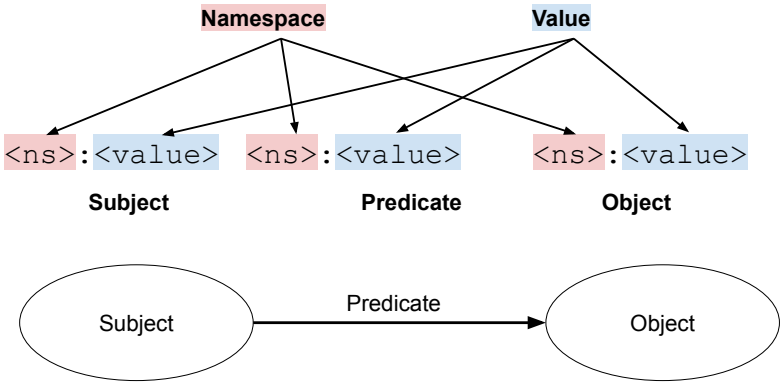


Figure 2.11: Generic RDF *triple*, consisting of a subject and object (nodes in a graph) and a predicate (directed edge from subject to object)

Figure 2.12 shows how the RDF data model can be used to describe a cyberphysical scenario consisting of a thermostat with an attached temperature sensor. Even without an understanding of the `brick:`, `example:` and `rdf:` prefixes, the intent of each of the triples is clear:

- Line 1 declares that a resource `tstat_ABC` has a `type` relationship to the concept of a `Thermostat`, i.e. `tstat_ABC` is an entity that happens to be a thermostat.
- Line 2 (which shares the same subject as line 1) declares that the entity `sen1` is a "point" of the thermostat entity.
- Line 3 mirrors the structure of Line 1 to state that `sen1` is a temperature sensor.

The elements of an RDF graph — the nodes and edges — are either IRIs or literals. An IRI, or Internationalized Resource Identifier, is a Unicode-encoded Uniform Resource Identifier (URI) as defined by RFC 3987 [47]; IRIs may but do not necessarily represent Internet-hosted resources. Literals are atomic typed data such as strings, integers and floating-point numbers. In RDF, IRIs can be used as subjects, predicates and objects but literals can only be used as subjects and objects[2]. Formally, this means that an RDF triple t is a member of the set

$$t \in (\mathcal{I} \times \mathcal{L}) \times \mathcal{I} \times (\mathcal{I} \times \mathcal{L})$$

where $\mathcal{I}$ is the set of all IRIs and $\mathcal{L}$ is the set of all literals.

[2]Many RDF databases and formulations constrain the use of literals to only objects, but this is an artificial restriction. This restricted RDF model follows from an early intuition that introducing the ability to make statements about literals, i.e. with literals in the *subject* position, would encourage bad modeling practices where a literal is incorrectly used to refer to an entity: for example `"Gabe" a foaf:Person` rather than `berkeley:Gabe a foaf:Person`.

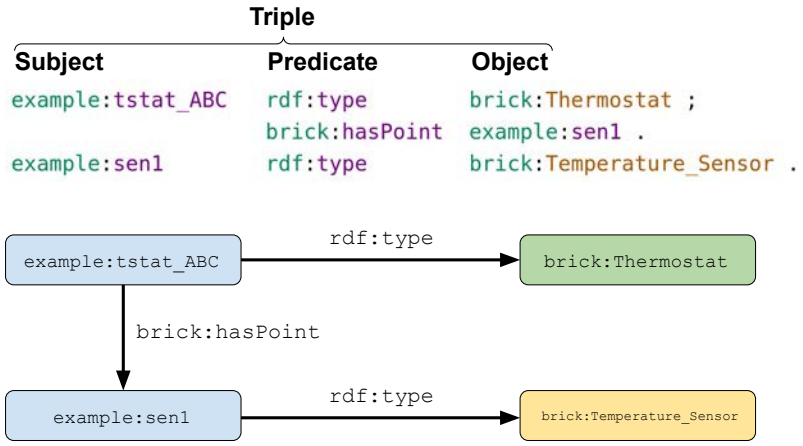


Figure 2.12: Simple RDF graph — depicted both graphically and textually — describing a thermostat with an attached temperature sensor.

IRIs are organized into *namespaces*, which are common prefixes of groups of IRIs. An IRI is a member of a namespace if that namespace is a prefix of the IRI. Typically, the namespace of an IRI is all but the last component of an IRI path or all of the IRI except for the fragment. Because IRIs can be quite lengthy (often in the tens of characters long), IRIs may be abbreviated by factoring out the namespace to a common identifier.

The Brick ontology has a namespace of `https://brickschema.org/schema/Brick#`, with a commonly used prefix of `brick`. This means that the Brick Thermostat concept's full IRI is `https://brickschema.org/schema/Brick#Thermostat`, but has an abbreviated form of `brick:Thermostat`. The mapping from abbreviation to namespace is purely a construct of RDF graph serialization and is used to simplify syntax. In this abbreviated form, the text before the `:` is referred to as the *prefix* and denotes the namespace.

Sometimes it is useful to model complex values; for example, the value of a sensor may have a timestamp, a scalar datum and the engineering units of the datum. Using IRIs and Literals, this could be modeled as the following:

```
1 example:sensor  rdf:type          brick:Sensor ;
2                 brick:hasValue    example:sensorValue1 .
3 example:sensorValue1    brick:hasTimestamp  "2021-04-14T00:00:00Z" .
4 example:sensorValue1    brick:units     "degF" .
5 example:sensorValue1    brick:value     72.3.
```

However, this requires the generation of an IRI (`example:sensorValue1` above) to refer to the aggregate of a set of properties and property values. *Blank nodes* are a special kind of IRI which act as an anonymous aggregate resource that groups the related information. Blank nodes are identified by a prefix of `_` and cannot be referred to directly:

```
example:sensor  rdf:type         brick:Sensor ;
                brick:hasValue   _:sensorValue1 .
_:sensorValue1    brick:hasTimestamp  "2021-04-14T00:00:00Z" .
_:sensorValue1    brick:units     "degF" .
_:sensorValue1    brick:value     72.3.
```

or, in Turtle syntax,

```
example:sensor  rdf:type         brick:Sensor ;
                brick:hasValue   [
                    brick:hasTimestamp  "2021-04-14T00:00:00Z" .
                    brick:units     "degF" .
                    brick:value     72.3.
                ] .
```

The use of Blank nodes is a modeling choice which marks certain resources as only useful as a collection of properties. A sensor entity is named with a full IRI because it needs to be referred to directly; the value of that sensor only makes sense in the context of a set of properties, namely the timestamp, value and units.

Figure 2.13 contains the Turtle representation of the RDF graph depicted in Figure 2.10. In this *Brick model* — an RDF graph which represents a building and its resources — each entity has a type which is given by a Brick class. Brick relationships are the predicates which describe how Brick entities relate to one another.

Linked Data and the Semantic Web

RDF graphs are one example of *linked data*. "Linked data" is the idea that knowledge can be defined in a distributed and decentralized manner: a resource, such as a IRI, can be referred to by many sources which can make arbitrary statements about that resource. Applied to the Internet, linked data can enable a *semantic web*. In a semantic web, IRIs refer to online resources. These resources can be described, typed and related to one another by RDF statements organized into graphs.

Even though cyberphysical systems are not usually exposed as publicly-accessible IRIs, they are naturally described as graphs [54]. The RDF data model is a convenient, standardized data model for encoding these graphs. A significant advantage of the RDF data model is it exists in an ecosystem of useful standards and technologies for querying graphs (SPARQL), formalizing knowledge (OWL) and constraining the forms of those graphs (SHACL).

SPARQL

SPARQL is the W3C-recommended query language for RDF graphs [66]. SPARQL defines how RDF graphs can be queried and updated. SELECT queries are the most common kind of SPARQL operation performed by self-adapting software. A SELECT query consists of a SELECT clause and a WHERE clause. The WHERE clause is composed of a set of *graph*

```
@prefix bldg: <http://example.com/mybuilding#> .
@prefix brick: <https://brickschema.org/schema/Brick#> .

bldg:AHU1A a brick:Air_Handler_Unit ;
    brick:feeds bldg:VAV2-4,
        bldg:VAV2_3 .

bldg:VAV2-3 a brick:Variable_Air_Volume_Box ;
    brick:feeds bldg:VAV2-3Zone .

bldg:Room-410 a brick:Room .
bldg:Room-411 a brick:Room .
bldg:Room-412 a brick:Room .

bldg:VAV2-3Zone a brick:HVAC_Zone ;
    brick:hasPart bldg:Room-410,
        bldg:Room-411,
        bldg:Room-412 .

bldg:VAV2-4 a brick:Variable_Air_Volume_Box ;
    brick:hasPart bldg:VAV2-4.DPR ;
    brick:hasPoint bldg:VAV2-4.SUPFLOW,
        bldg:VAV2-4.SUPFLSP .

bldg:VAV2-4.DPR a brick:Damper ;
    brick:hasPoint bldg:VAV2-4.DPRPOS .

bldg:VAV2-4.DPRPOS a brick:Damper_Position_Setpoint .
bldg:VAV2-4.SUPFLOW a brick:Supply_Air_Flow_Sensor .
bldg:VAV2-4.SUPFLSP a brick:Supply_Air_Flow_Setpoint .
bldg:VAV2-4.ZN_T a brick:Supply_Air_Temperature_Sensor .
```

Figure 2.13: Example Turtle-formatted RDF graph for the Brick model depicted in Figure 2.10

patterns. A basic graph pattern is a triple of `subject`, `path expression`, `object`. Subjects and objects can be IRIs, Literals or *variables*. A path expression describes a route from the subject to the object in terms of IRIs and variables. SPARQL 1.1 defines a family of path operators and expressions [66] which include sequence (`path1 / path2`), transitive closure (`path1*`), alternatives (`path1 | path2`) and optional paths (`path1?`).

The result of evaluating a graph pattern is a relation R (a set of n-ary tuples) whose n attributes are bindings of RDF graph terms to the variables in that clause. The `.` operator between graph patterns is the conjunctive operator. A variable appearing in more than one graph pattern amounts to a join between those two graph patterns. The result of executing a SPARQL query is a relation derived from the natural join of the intermediate relations produced by each of the graph patterns; this relation has a projection operator applied which preserves the variables contained in the SELECT clause. Figure 2.14 contains a SPARQL query which retrieves the name of all VAVs in Figure 2.13 which have air flow sensors and air

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX brick: <https://brickschema.org/schema/Brick#>
SELECT ?vav ?sensor ?setpoint WHERE {
    ?vav    rdf:type/rdfs:subClassOf*   brick:VAV .
    ?vav    brick:hasPoint ?sensor .
    ?vav    brick:hasPoint ?setpoint .
    ?sensor rdf:type brick:Supply_Air_Flow_Sensor .
    ?setpoint rdf:type brick:Supply_Air_Flow_Setpoint .
}
```

Figure 2.14: Simple SPARQL query for the airflow setpoints and sensors associated with VAVs

flow setpoints, along with the names of those two points. The result of executing Figure 2.14 on Figure 2.13 is a single tuple with the bindings:

- `?vav`: `http://example.com/mybuilding#VAV2-4`
- `?sensor`: `http://example.com/mybuilding#VAV2-4.SUPFLOW`
- `?setpoint`: `http://example.com/mybuilding#VAV2-4.SUPFLSP`

Self-adapting software can use graph query languages like SPARQL to retrieve necessary configuration data and other information from an RDF graph representing the semantic metadata for a particular environment. This is closely examined in Chapter 5.

OWL

On their own, RDF graphs are just collections of statements with a well-defined syntax and structure. In order to enable the consistent and interpretable exchange of information, the *semantics* or meaning of the data must be formally specified. An *ontology* is a way of programmatically defining the meaning of data for a particular knowledge domain as a set of rules and axioms. The Web Ontology Language, or OWL, is a language for declaratively expressing those rules and axioms.

The rules and axioms expressible in OWL operate under the *open world assumption* or OWA. Statements that are contained within a graph are true. Under the open world assumption, statements that are not contained in the graph are simply unknown and cannot be asserted as false. Its corollary, the *closed world assumption* or CWA, treats statements not in the graph as false; relational database systems operate under CWA. The OWA makes sense in the context of the semantic web, where the absence of a statement in a particular graph does not mean that another graph has made that statement. However, as explored in Chapter 4, OWA presents challenges for cyberphysical systems when the full extent of the graph can actually be known.

There are two ways in which computable meaning can be attached to the statements in an OWL ontology: OWL 2 DL and OWL 2 Full [74]. OWL 2 DL can compute most of what OWL 2 Full can compute and is dramatically faster and simpler to compute than OWL 2 Full. The "DL" indicates that OWL 2 DL is a description logic, which is a decidable but less expressive subset of first-order logic. Chapter 4 establishes that only OWL 2 DL is required to implement the ontology features required for consistent and extensible semantic cyberphysical metadata. For this reason and the decidability and reduced computational complexity of OWL 2 DL compared to OWL 2 Full, this thesis focuses on the former. Description logics make a distinction between the *TBox* and the *ABox*. The TBox, or "terminological box", is the set of statements that define the ontology; this includes class structures and relationships between concepts. The ABox, or "assertion box", are the ground statements or "facts" that populate the content of the model. There is no formal distinction between TBox and ABox; these terms are used to differentiate between the statements that define the ontology and the statements that define the model instance.

OWL 2 DL can be broken down into several *profiles*, or sublanguages, that define subsets of OWL 2 DL that trade off computational complexity for expressive power in different ways [74]. The ontology features described in Chapter 4 require the OWL 2 RL profile: this is a subset of OWL 2 DL that can be implemented in a rule-based language such as Datalog[3]. Chapters 4 and 7 delve into the OWL 2 RL rules used to express metadata and the systems that evaluate those rules, respectively.

SHACL

The Shapes Constraint Language (SHACL) is at its core a language for validating an RDF graph against a set of conditions [81]. RDF graph validation allows a modeler to express what the graph should and should not contain, depending on other properties of the graph. SHACL constraints can also helpfully emulate some features of CWA.

SHACL constraints are expressed as RDF triples and take the form of a set of *shapes*. A shape is an entity with a *target*, a specification of the IRIs that it applies to, and a set of parameterized *constraints* that describe predicates on that target. There are two kinds of shapes: node shapes and property shapes. Node shapes specify constraints about the nodes of an RDF graph; property shapes specify constraints about the edge of an RDF graph.

Validating an RDF graph against a set of SHACL shapes generates a report which describes whether or not the validation passed and, if not, which constraints were violated by which elements of the target graph. SHACL-AF [80] is an addendum to SHACL that adds reasoning and inference capabilities.

[3] a closed-world relational language

2.5 Summary

The *built environment* spans the buildings, cities, power grids, transportation systems and other networked, physical components of the human experience. These *cyberphysical* components consist of a complex, interconnected array of devices, equipment, sensors and other data sources. Despite the existence of many possible digital representations of this infrastructure, such representations do not capture the salient elements that support self-adapting software. Linked data specification languages such as RDF, OWL, SHACL and SPARQL provide a means of specifying a formal representation of cyberphysical systems that can be reasoned about, validated and queried. These languages form the foundation of the proposed approach to enabling self-adapting software.

Chapter 3

A Vision of Self-Adapting Software

Executing data-oriented cyberphysical software at scale is impractical because of the effort required to customize the operation of each application to each deployment environment. This customization effort is *manual* and *time-consuming* due to the lack of structured metadata about cyberphysical environments, and the lack of an effective method of customizing the behavior of software.

Software needs to be customized because cyberphysical environments are heterogeneous and thus require bespoke treatment. The nature of the customization depends on the needs and complexity of the software. At minimum, the software needs to be configured with the data sources it needs to read and where to read them from. This requires discovery of which data sources are relevant to the application. In current practice, data discovery is performed

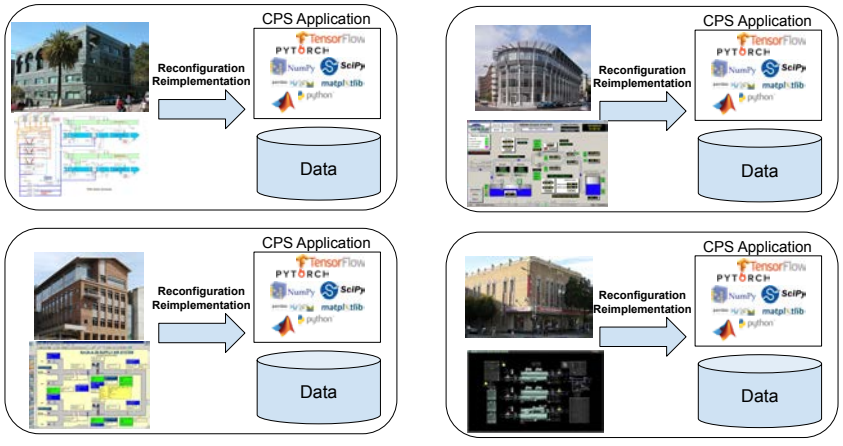


Figure 3.1: Current practice: deploying software to different sites requires re-implementing and re-configuring the software for *each site*

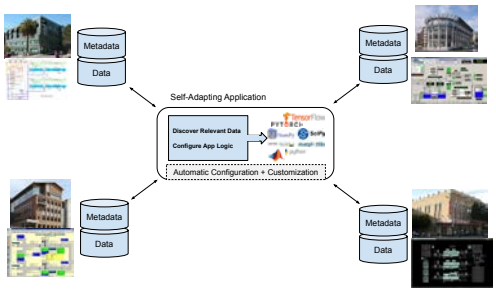


Figure 3.2: Self-adapting software will query a metadata model of its environment in order to automatically configure a single implementation to operate on new sites.

manually[1] through inspection of BMS labels or other unstructured, human-oriented metadata. Configuring more complex software may require additional context about what kinds of subsystems are in the environment and how those subsystems are composed and operated. Discovery of context is more time consuming and more error prone because this metadata is not usually captured in available representations. In most cases, each environment is so unique that configuring software requires a wholly new algorithm or implementation for the software's purpose to be achieved. Figure 3.1 describes the current state-of-the-art: for each building, a new implementation must be developed for each deployment environment. This approach is impractical at scales larger than a handful of environments and is a significant bottleneck to the adoption of data-driven sustainable and energy efficiency practices.

Self-adapting software (Figure 3.2) *is a new paradigm for programming cyberphysical systems which eliminates the configuration and implementation bottleneck to deploying software at large scales.* By exploiting rich, descriptive semantic metadata about each environment, self-adapting software is able to discover enough about its deployment to configure itself automatically. This enables a new regime of "write once, run anywhere" software by making cyberphysical software become substantially easier to write and deploy. Lowering the barrier to entry for adopting cyberphysical software also democratizes access to data-driven sustainable practices and energy efficiency measures that were previously cost-prohibitive to deploy at scale.

Addressing Challenges

Accomplishing this vision requires addressing a family of challenges; each of these is addressed in a chapter of this thesis.

Challenge #1: Describing Heterogeneous Settings: The success of self-adapting software for cyberphysical systems is contingent on the existence of a machine-readable description of the nature and context of the data sources and other resources in a given

[1]such as browsing through web interfaces, executing regular expressions over lists of points, and the like

environment. This description must be able to describe enough of the environment that self-adapting software can perform the necessary discovery and configuration. However, existing metadata representations are ill-suited for modeling the diversity and heterogeneity of cyberphysical systems. This thesis proposes the use of a formalized graph model for metadata that is flexible enough to represent many different kinds of cyberphysical environments. Each cyberphysical environment will be represented by an RDF graph authored in Brick which captures the data sources and their associated context. The semantics and structure of the graph will be informed by a formal OWL-based ontology which defines the concepts, axioms and relationships required to describe cyberphysical environments. Doing this effectively requires new ontology design patterns which enforce interpretability and modeling consistency of the metadata graph while enabling extensibility of the ontology design. Chapter 4 reviews shortcomings of existing metadata representations for cyberphysical systems and identifies fundamental flaws in popular tag-based metadata designs. It then proposes and implements a set of new ontology design patterns and features which advance the applicability of semantic web technologies for complex cyberphysical systems. These novel design patterns are evaluated in the context of Brick, a metadata ontology for data-driven smart buildings.

Challenge #2: Designing an Effective Programming Model: Self-adapting software must have a means of exploiting the semantic metadata representing each environment in order to configure itself. How should such software be expressed? Specifically, how should self-adapting software access the metadata model for an environment and implement their application logic in terms of the content of that model? The metadata model adheres to the standard RDF data model, so it is possible for self-adapting software to use existing query languages and technologies like SPARQL for discovering and accessing the metadata model. However, a query language alone is insufficient to enable self-adapting software. Chapter 5 explores the design of two different programming models that provide first-class support for semantic metadata within a Python-based programming framework. The first model decomposes application logic into set of stages which progressively abstract away heterogeneity between environments, allowing the core application logic to be expressed in an environment-agnostic manner. This provides a modular, batch-style approach to constructing portable software: query and data cleaning and analytics components (or "stages") can be reused or repurposed with minimal reconfiguration. The second model explores an imperative abstraction of SPARQL queries that is more appropriate for interactive and exploratory programming.

Challenge #3: Metadata Management Over Time: Cyberphysical environments inevitably evolve over time. These changes may be in response to a variety of stimuli, including decay (parts of the environment breaking), repairs and retrofits (replacing or augmenting parts of the environment), Semantic metadata models must be kept up-to-date with their cyberphysical counterparts in order for self-adapting software to execute correctly and continuously. This thesis proposes a metadata management system that can detect changes in the environment and use those changes to bootstrap edits to the metadata model. The system can also handle changes to the semantics of the model itself which may occur as the underlying ontology evolves and expands. Addressing these challenges requires the devel-

opment of new semantic data integration techniques. Prior work on database migrations provides mechanical techniques for how changes can be incorporated into the model, but does not ensure that these changes are semantically valid. The data integration literature establishes techniques for extracting structured data from heterogeneous information sources but does not address how different extracted data can be merged into a single, semantically valid model. Chapter 6 develops and formalizes a system that addresses the shortcomings of existing techniques which arise in their application to cyberphysical environments and metadata.

Challenge #4: Metadata Management at Scale: A final challenge with enabling and supporting self-adapting software is how to design and implement the platforms that store, query and manipulate the semantic metadata. Existing platforms for semantic-web data are largely designed for batch processing on large graphs (on the order of millions or billions of nodes and edges); most are positioned as knowledge bases and information references rather than platforms for large volumes of interactive applications. Most research in RDF databases has focused on efficient storage and query processing as a result. However, the ontology features that that make semantic metadata suitable for self-adapting software require support for performant reasoning and inference. Furthermore, existing platforms do not implement features for integrating semantic metadata with historical telemetry or interfaces for cyberphysical control. Chapter 7 presents the architectures and implementations of two real-world systems that address these gaps in RDF database design. The first, Mortar, is an analytics platform with an integrated timeseries database and RDF database that provides a declarative API for programming and executing self-adapting applications using the staged execution model. This platform is augmented by `reasonable`, a performant OWL-RL reasoner that augments semantic metadata models with inferred information and implements the required ontology features.

Architecture

Figure 3.3 outlines how this vision of self-adapting software can be realized. Cyberphysical systems — including but not limited to buildings, cities, water treatment and smart grids — produce data by means of an increasing numbers of sensors and other digital sources, and are represented by a diverse family of structured and unstructured metadata sources. This thesis proposes a semantic and structural lifting of these sources into a *formal graph-based model* as typified by Brick. This model contains descriptive metadata which captures the composition and structure of cyberphysical systems in an environment as well as the data sources and command points which track the behavior of the environment over time.

The metadata model for each environment can be bootstrapped and maintained through a *metadata integration* process which extracts and combines semantic metadata from available digital representations. A self-adapting software platform manages the content of the metadata model over time, including the evaluation of inference rules and validation of the model with respect to logical axioms defined by a formal ontology. This ensures that

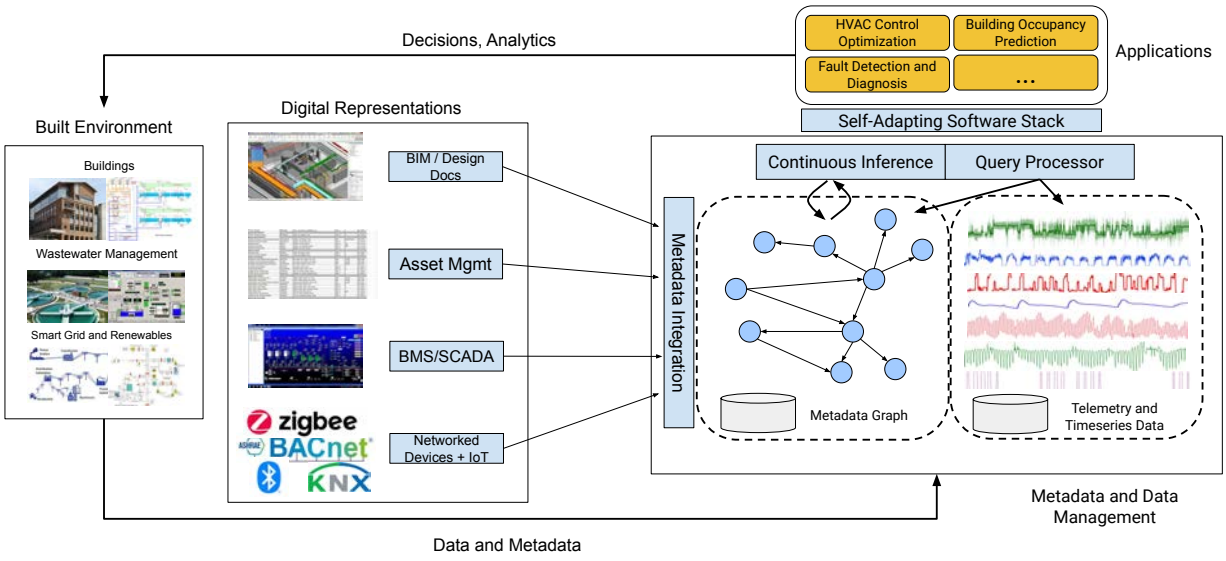


Figure 3.3: A high level vision of self-adapting software interacting with the built environment

self-adapting software can access a valid and semantic representation that is the union of what is explicitly provided by the environment and what can be inferred about the environment given that information. Finally, the platform provides access to historical timeseries data about the environment which is contextualized by the metadata model. Self-adapting software is expressed by means of a *programming model* which uses the context captured and exposed by the platform to configure the operation of an application to perform in a given environment. This presents an approach to authoring and configuring cyberphysical software which enables its deployment and execution in hundreds or thousands of different environments without having to change a single line of code.

Chapter 4

Managing Heterogeneity with Semantic Metadata

> We are tied down to a language
> which makes up in obscurity
> what it lacks in style.
>
> *Tom Stoppard*, Rosencrantz and Guildenstern are Dead

A fundamental challenge to enabling self-adapting software is dealing with the intrinsic heterogeneity of the cyberphysical systems being controlled and analyzed. When every site, building and system differs both in the nature of its construction and in how it is described, there is often little choice but to undertake the manual, time-consuming and error-prone process of rewriting and reconfiguring software for each intended deployment. Prior work establishes that rich, descriptive metadata is crucial to reducing and even automating this effort [16, 15, 84, 25, 54]. Self-adapting software queries metadata in order to configure its operation to the particulars of a given deployment. This includes the discovery of data sources and their context as well as the determination of which algorithms are appropriate.

Doing this effectively requires a careful balance of potentially conflicting design principles. An effective metadata representation for self-adapting software must be *general* enough to represent many different kinds of cyberphysical systems while capturing enough *specific* detail to support the complex requirements of software. The representation should model an appropriate level of abstraction that permits automated reasoning about the composition and function of cyberphysical systems. This chapter explores the design and use of a semantic metadata ontology for cyberphysical systems which effectively manages heterogeneity in order to support self-adapting applications.

Metadata Design Principles

Established metadata representations and standards (Chapter 2) do not capture the information required to support real-world cyberphysical applications [26]. To address these shortcomings, the Brick ontology, developed in 2016, established three design goals to guide the design of effective building metadata for such applications [16]:

- **Completeness**: Can the metadata representation define the concepts, types and entities that exist in buildings and are required by canonical energy-, operations- and management-related applications?
- **Expressiveness**: Can the metadata representation define the important relationships between entities that exist in buildings and are required by canonical energy-, operations- and management-related applications?
- **Usability**: Can the metadata be presented in a form that supports the development of applications through integration with existing data analysis frameworks and tools?

These three design goals inform an empirical approach to the design of a metadata representation. The set of equipment, location and point concepts defined in Brick are sourced from the BMS labels for six large college campus office and laboratory buildings [16]. The set of Brick relationships are sourced from a systematic study of over 90 building applications [26] Together, these two sources make Brick both *complete* and *expressive*. A representative set of building analytics applications, drawn from the literature, was implemented using queries against Brick models for the six reference buildings; the success of this initial study indicates that Brick is also *usable*.

This chapter argues that these principles, although successful, are not sufficient for managing heterogeneity at large scale. In particular, a metadata representation must also promote

- **Interpretability**: The intended meaning of the metadata in the model should be unambiguous and defined programmatically. If there is more than one interpretation of a part of the metadata model, software will not be able to reliably identify and contextualize the data it needs to operate.
- **Consistency**: The structure and composition of metadata models for cyberphysical systems should depend more on the level of detail and completeness necessary for applications, rather than on the opinions and philosophies of the modeler. Similar cyberphysical systems should be described in a similar manner. Two individuals describing the same cyberphysical system should produce similar, or at least compatible, metadata models.
- **Extensibility**: The metadata representation should support the structured extension of the concepts and relationships defined within in a consistent and interpretable manner

The rest of this chapter will review the issues and design limitations of existing metadata representations for cyberphysical systems in the context of the design goals outlined above, in particular, linked data ontologies and tag-based metadata as typified by Project Haystack. A systematic study of the consistency, interpretability and extensibility issues of the Haystack metadata representation is performed. A review of the limitations and issues incumbent in the Brick ontology is also conducted. The resulting discussion explores how these issues are due to a lack of *formal axioms*. This motivates the design and implementation of *Brick+*, a formalization and extension of Brick that combines the "tag-based" benefits of Haystack with the "class-based" benefits of Brick.[1]

4.1 Limitations of Existing Metadata Representations

Despite the large number of metadata representations available in the IoT and smart building domains, they largely do not address the needs of self-adapting software.

Consumer-facing IoT Platforms

IoT software platforms define APIs and libraries that directly support executing software on cyberphysical systems. The metadata aspects of these platforms focus on service discovery and descriptions of device functionality that can be accessed via APIs. Descriptions of device functionality typically involve structured documents containing well-known service names, with implementation and access details available as standardized parameters. This is seen in designs like Ninja [36], Universal Plug-and-Play, and mDNS [31].

Another approach, commonly referred to as "the web of things", uses linked data and other web standards to describe and provide access to device functionality [160]. The most mature effort is the ontology produced by the W3C Web of Things working group [155], which standardizes the representation of *Thing Descriptions*. Thing Descriptions represent the set of supported interactions for a device using a small vocabulary that captures the structure of data that is input and output to the device. Because these descriptions are built on RDF they can — and are expected to — leverage external ontologies for semantic descriptions of what the interactions actually are.

However, a fundamental limitation of these digital representations is that they leave out, or fail to standardize, contextual information about where devices are, how they are installed and how they interact with their environment and each other. Service discovery frameworks are designed to enable access to remote services, regardless of location; the focus is on what functionality those services provide. Likewise, consumer IoT platforms largely target small deployments in typically residential settings: devices with straightforward functionality in relatively homogeneous settings do not have a need for complex contextual metadata. As a

[1]The distinction between Brick+ and Brick is done solely to differentiate what work has been done as a part of this thesis. Brick+ has been merged into Brick versions 1.1 and 1.2.

result, these standards and platforms have a limited ability to support the richer metadata for the heterogeneous settings typical of large buildings and complex cyberphysical systems.

Building Information Modeling

Building Information Modeling (BIM) technologies — including gbXML [60], IFC [1] and Revit [11] — are designed to enable the exchange of information between different teams during the design and construction of a building. As a result, the data model must be flexible and extensible enough to capture precise definitions of geometry, mechanical, electrical and plumbing infrastructure [12]. The Industry Foundation Classes standard contains hundreds of concepts and properties describing furniture, HVAC, lighting and other kinds of physical equipment, in addition to the segments, fittings and connectors constituting the distribution systems for different substances in the building [1].

The resulting complexity of these models makes them difficult to maintain and use for other stages of the building lifecycle, which require different kinds of details to be captured [139]. In particular, the flexibility provided by BIM data models presents challenges for the *consistent* and *interpretable* exchange of BIM data because the extensions made "in the field" by the teams designing and constructing buildings do not extend the BIM in the same way. This lack of interoperability has real costs: the National Institute of Standards and Technology published a report in 2004 estimating the cost of inadequate interoperability within the property, construction and facility management industry to be $15.8 billion per year [57].

The interoperability issues with BIM accentuate the need for a data model to be extensible in a consistent manner, so that the resulting representations are easily and automatically interpretable. This begins with defining a data model that is complete and expressive enough that the necessary concepts can be defined.

Upper Ontologies for IoT

Issues of structural interoperability — the ability of two or more systems to exchange information — and *semantic* interoperability — the ability of two or more systems to exchange information with a consistent and interpretable meaning — have long been the subject of study in linked data and semantic web communities. The RDF data model, with its many serialization formats such as Turtle [18], RDF/XML [17] and more recently JSON-LD [134], enables a structural interoperability between data sources: triples are triples, and they can be easily stored and queried.

Ontology design, in particular the design of *upper ontologies*, intends to solve issues of semantic interoperability. Upper ontologies attempt to define a common family of generic concepts and axioms that are common to many (if not all) domains, and can therefore enable semantic interoperability. Domain-specific ontologies can "anchor" their own concepts and properties to those provided by an upper ontology. Ontologies based on the same upper ontology can then interpret each other's contents in terms of that upper ontology.

Of course, the issue is that there is no single universal upper ontology. Sowa's ontology defines logical and philosophical foundations of knowledge (e.g. what differentiates objects, concepts and symbols) [133]. The Basic Fundamental Ontology defines notions of function, role and disposition to inform a coherent perspective on biomedical-oriented ontologies [10]. The General Formal Ontology defines categories of concepts based on space-time (permanence and duration), material structures (physical things that take up space), processes and occurrences [72]. As is clear from this small sampling of upper ontologies, they deal in high-level and philosophical concepts.

Ironically, the intended generality of upper ontologies raises more interoperability issues than are purportedly solved. Consider a family of ontologies that have been developed for IoT and cyberphysical systems: SSN/SOSA [62], SAREF [37], and the SAREF family of ontologies [116]. Each of these ontologies provide practical, formal descriptions of elements of cyberphysical systems. SSN/SOSA describes sensors, actuators and the observations and actuations made by those entities. SAREF describes "smart appliance" devices, their properties and purposes, and the tasks and actions they take. Several SAREF extensions define common building equipment and components (SAREF4BLDG) and system topologies and connections between systems (SAREF4SYST). However, even though the concepts defined in these ontologies cover many aspects of cyberphysical systems and are general enough to model many different kinds of cyberphysical systems, the actual vocabulary provided is neither specific nor extensive enough to satisfy the requirements of self-adapting applications.

The reason for this is simple: when describing a real-world cyberphysical system, a modeler wants to be able to use the domain-specific terms familiar to them. For example, the SAREF `Device` class may be appropriate for AHUs, VAVs, dampers, fans, meters, pumps, valves and chillers, but without extra properties, these specific kinds of equipment are not distinguishable. Modelers can extend the SAREF ontology by subclassing the `Device` class, but there is no guarantee that different modelers will extend SAREF in the same way to model the same concepts. This motivates the need for an ontology to support *structured extensibility*, and to define enough common classes that extensions can be avoided for common cases.

Tags and Folksonomies

On the other side of the spectrum of axiomatic descriptions of knowledge are so-called "folksonomies". Folksonomies encompass any kind of user-generated metadata, but the most common form of this is the use of *tags*, which are simple words and terms that are associated with documents or other entities. Tags and folksonomies typify common established practice for annotating and describing data. Tags allow users to "describe and organize content with any vocabulary they choose" [91]. Tags exist in a flat namespace with no organization or associated definitions. The meaning of a tag is ambiguous without additional context, which may be intimated by other tags, or only evident after observing the entity that is the subject of the tag.

The main benefit of tags are their flexibility and low barrier to entry [91]; there is no need to learn relatively esoteric logic languages and familiarize oneself with the philosophical underpinnings of an upper ontology. The social bookmarking site `del.icio.us` and the photo sharing site Flickr were largely successful due to their adoption of user-generated tags to describe the content described by users. More recent examples of successful tag-based products include Instagram and Twitter, which use "hashtags" as a form of tagging. After tagging an entity such as a post or photo with a tag, users can find other content with the same tag. This content may not use the tag in the same way, but the intent is to share content, not to communicate its meaning in an unambiguous manner.

Project Haystack [118] adopts a similar folksonomic approach to metadata in buildings. The equipment, points and other entities in a Haystack model of a building are described with sets of tags. Haystack defines a dictionary of common tags which are available to users, but also allows users to define their own tags. The result is a data model that is flexible and extensible but also does not enable the consistent, interpretable and interoperable exchange of building metadata.

Relational Schemas

Although cyberphysical metadata is fundamentally graph-oriented, it is possible to express most features of the metadata in a relational form. However, the heterogeneous and interconnected nature of cyberphysical metadata makes the relational model a poor fit.

Relational tables can represent names and properties of the entities in the environment. More than one table may be required to represent different types of entities: the properties of a temperature sensor entity will be different than the properties of a pump equipment entity. Variability in which properties are available for each entity may result in wide and sparse tables because the schema must represent *all* possible properties for an entity, even if a given entity does not have those properties. Foreign keys, one-to-many and many-to-many tables can all be used to represent the relationships between entities.

While it is possible to author a relational schema for a particular environment, developing an *efficient* schema requires prior knowledge of the kinds of entities in the environment, their properties, and the relationships. Generalizing the relational schema to other environments requires denormalization of the schema. Denormalized schemas like entity-attribute-value tables are essentially a representation of RDF's subject-predicate-object triple. Recursive queries are required in order for denormalized schemas to support the required query features for traversing the graph structure of the metadata; this includes path queries and transitive closures. For these reasons, a single relational schema would be a poor fit to model heterogeneous and complex cyberphysical systems. Developing a unique relational schema for each cyberphysical environment is only a partial solution, because software would still have to be rewritten for each environment.

4.2 Modeling Issues for Tag-Based Metadata

The tag-based Haystack metadata model has several intrinsic issues that fundamentally limit the semantic interoperability of the data it describes: designing for flexibility at the cost of consistency, a lack of a formal specification, and inconsistent and ill-defined modeling practices.

Balancing Composability and Consistency

The design of Haystack prioritizes familiarity, composability and flexibility over consistency and interpretability. This is one of the key benefits of a tag-based metadata scheme [91]. However, this flexibility hampers the ability of an arbitrary consumer of tags to consistently interpret how a specific combination of tags is intended. In tag-based metadata schemes, increased composability comes at the cost of lower consistency. Specifically, without *formal rules* defining the meaning of groups of tags and how tags may be composed, the interpretation of a *tag set* is dependent upon idiom, convention and other "common knowledge" of the group or individual who chose the tag set. As a result, the tag set chosen by one individual to describe an entity may have multiple possible subtly different meanings, or no meaning at all, to other individuals.

Haystack provides a dictionary of tags with community-provided definitions; these definitions are intended to make it clear when a modeler should choose one tag over the other. However, the tag definitions often include caveats and describe idioms, which complicate tag usage. For example, the Haystack definition of the `air` tag points out that the use of the tag implies a dry bulb temperature measurement unless otherwise specified:

> "Point associated with the measurement or control of air. In regards to `wetBulb`, points with the `air` tag are associated with dry bulb.[122]"

The root of the semantic ambiguity is the fact that tags can be used in different contexts. Even if a tag has a specific definition (such as `air`), the use of a tag does not communicate how the tag's concept relates to the other tags associated with a given entity. For example, the `heat` and `oil` tag together do not indicate if oil is being heated, or if hot oil is doing the heating of some other substance. Haystack identifies this kind of issue as a "semantic conflict":

> "Another consideration is semantic conflicts. Many of the primary entity tags carry very specific semantics. For example the `site` tag by its presence means the data models a geographic site. So we cannot reuse the `site` tag to mean something associated with a site; which is why use the camel case tag `siteMeter` to mean the main meter associated with a site.[119]"

To mitigate this issue, Haystack introduces compound tags (called "camel case tags" above), which are concatenations of existing tags into new atomic tags with specific and distinct meanings from the tags from which they are composed. This reduces the composability of the tagging scheme, as tags now only have well-defined meanings in specific contexts and cannot be arbitrarily combined.

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

Table 4.1: An enumeration of the intended use and context of tags relating to heating and cooling, as given by the Haystack documentation. Note the differences in diction across compound tags, and how some compound tags could be assembled from more atomic tags. Some tags are used both for equipment and for points when equipment is modeled as a single point (such as VFDs, Fans, Coils)

As an illustrative example, consider the `hotWaterHeat` compound tag: Haystack defines this tag as an annotation on an air handling unit that has a hot water-based heating capability. Unintuitively, this annotation *does not* use the `heat`, `hot`, `water` *or* `air` tags. In this way, the need to bring consistency to tags comes impedes their use as a flexible and permissive annotation feature: the `hot`, `water`, `heat` and `air` tags applied to an air handling unit may refer to many different aspects of its functionality.

Table 4.1 contains the results of a survey of defined Haystack tags that contain the word "heat" or "cool". Each of the tags in the table is defined in the Haystack dictionary, and each have a unique and thus nonfungible definition. The first three columns of the table

capture the contexts in which each tag has meaning:

- *Describes* ***equip***: if this column is checked, then the tag describes equipment, often by what function it performs. For example, the `gasHeat` tag indicates that the related equipment consumes gas to perform some heating process.
- *Describes* ***point***: if this column is checked, then the tag describes the context of a data stream (point). For example, the `reheat` tag describes data about the reheating capability of a VAV.
- *Describes mechanism*: if this columns is checked, then the tag communicates *how* a certain process or function is performed. For example, the `gasHeat` tag describes the nature of the heating process.

The last six columns of the table indicate if the tag is specifically for use with only one kind of equipment. The resulting tagging scheme is often unintuitive: the `heat` tag would seem to relate to a heating process, but it is in fact intended only for heating coils that occur in AHU and VAV equipment. Further, the tag definitions are simple text: there are no formal or standardized rules for communicating how they are intended be used or not used.

These issues are compounded by the fact that the Haystack query model does not support string matching operations, meaning that tags can only be matched by equality. This makes it difficult to use generic tags as a way of searching for relevant or similar entities [91], for instance, a query for all entities that have a tag containing the word `"heat"`.

Most of the ambiguities exposed above can be tracked to confusion as to what aspect or process of an entity a tag relates to. To address this issue, tags must either have meaning independent of context, which is the direction that most tag-based metadata systems take but also provides the least consistency, or a way of communicating the intended context[2]. Because the Haystack data model is so simple — unnested key-value documents with a unique identifier — there is no mechanism to easily communicate the intended use of a tag (recall the dilemma "is `oil` being `heat`ed, or is it doing the `heat`ing?"). This forces Haystack into the other model in which tags must have concrete meanings that require no additional context to interpret correctly; however, this increases the size of the tag dictionary, complicates usage, and limits the composability of the tags in the dictionary. The rest of this chapter will explore how composability and consistency can be balanced through the use of a formal data model.

Formalizing Tag Composition

Haystack lacks rules to formalize how tags may or may not be composed. This raises two issues for semantic interoperability. First, arbitrary combinations of tags may not have a consistent interpretation, which hampers Haystack's ability to provide an effective and

[2]This is called *reification* and it will be discussed in-depth in the next section.

standardized description of data. Ultimately, this makes well-known concepts harder to communicate consistently, and makes it difficult to figure out what the proper descriptions are. Second, the use of tags to communicate the *type* of an entity can result in a class organization that is not *well-formed*.

These issues are examined through an OWL-like treatment of classes and entities: all entities of the same type (i.e. belonging to the same *class*) are members of the same set. A *subclass* is a set that is a subset of its superclass; the entities in the subclass *must* belong to the superclass, but not all entities in the superclass are necessarily members of the subclass. A set of classes and the subclass relationships between them form a directed graph called a *class organization*. A class organization is *well-formed* if it meets two conditions:

1. Acyclic: no class should be a superclass of itself.
2. Consistent semantics: if one class is a subclass of another, then it should be a *more specific* concept than its superclass

A well-formed class organization is essential for the creation of consistent metadata models because it facilitates the automated discovery of classes. In order to discover what classes are available and to determine which is the most appropriate for a given entity, a user can browse the class organization from the topmost superclasses (equipment, locations, sensors, etc) and follow the subclass links to find more specific versions of those classes. Without a well-formed class organization, the traversal cannot take advantage of the expected "general to specific" organization in the class organization. A well-formed class organization is also extensible: users can create new, more specific classes that are subclasses of existing and more general superclasses. Even in the absence of a textual definition for this new class, the subclass relationship provides an immediate contextual scoping for how the class is meant to be used; this follows from the second condition for a well-formed class organization.

In the Haystack metadata model, the type or class of an entity is indicated by its tag set — the *marker tags* are associated with the entity. Formally, the tag set of an entity e is given by $T(e)$. A class C is defined as a set of tags given by $T(C)$. A class C_b is a subclass of C_a if $T(C_a) \subseteq T(C_b)$. An entity e is an instance of class C if $T(e) \subseteq T(C)$. If an entity e is an instance of class C_b and C_b is a subclass of C_a, then e is necessarily an instance of both C_b and C_a.

This formalism reveals how the use of tag sets to communicate the type of an entity is insufficient for defining a well-formed class hierarchy in two ways: tags imply the existence of classes that have no consistent or discernible meaning, and the resulting hierarchy may not have consistent semantics.

To the first point, consider the set of tags `sensor`, `temp`, `discharge` and `air`. There are

$$15 = \sum_{n=1}^{4} \binom{4}{n}$$

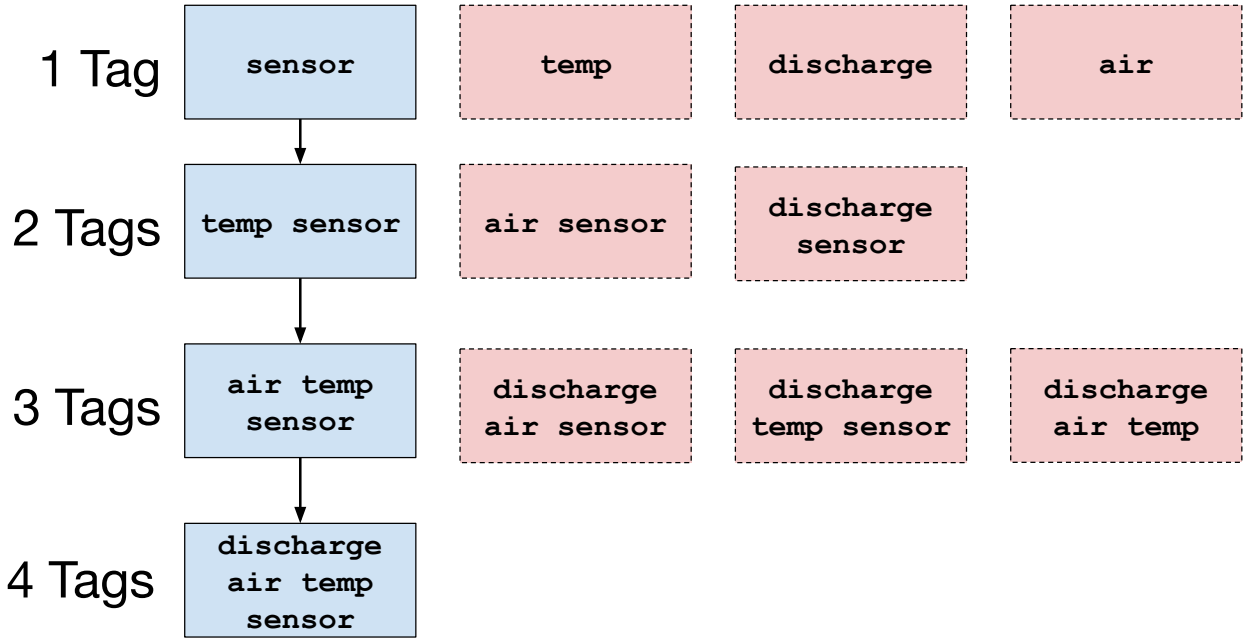


Figure 4.1: The set of valid (blue + solid outline) and invalid (red + dashed outline) tagsets for a set of four tags. The class hierarchy is established from top to bottom;subclass relationships are indicated by arrows

combinations of tags possible from that dictionary. Figure 4.1 contains 12 of these. However, of these 15 tags, only 4 of them have an interpretation that makes sense in the building HVAC domain (along the left side).

To the second point, it is possible to construct two class definitions C_i and C_j such that $T(C_i) \subseteq T(C_j)$ but the definition of C_i is disjoint from C_j. Consider two concepts: `Air Flow Setpoint` (the desired cubic feet per minute of air flow) and `Max Air Flow Setpoint` (the maximum allowed air flow setpoint). In Haystack, `Air Flow Setpoint` would be identified by the `air`, `flow` and `sp` tags, and `Max Air Flow Setpoint` would be identified by the `air`, `flow`, `sp` and `max` tags. Although the tags of `Air Flow Setpoint` are a subset of the `Max Air Flow Setpoint` class, `Air Flow Setpoint` is *not* a superclass of `Max Air Flow Setpoint`: the former is a setpoint, but the latter is actually a parameter governing the selection of setpoints and therefore belongs to a distinct subhierarchy.

To circumvent this issue, rules for defining valid tagsets for subclass relationships must be defined in terms of which tags can be added to a given tag set to produce a valid subclass relationship. Defining a concept requires knowing which tags *cannot* be added; without a clear set of rules for validation, a user may use the `max` and `min` tags to indicate the upper/lower bounds of deadband-based control, which is inconsistent with the intended usage of these tags. Haystack defines a few rules for extending existing concepts with new tags. Two examples from many:

1. AHU entities can be further characterized by their heating method, given by one of `gasHeat`, `hotWaterHeat`, `steamHeat` or `gasHeat`.

2. Water meter entities can be further differentiated by the tags `domestic`, `chilled`, `condenser`, `hot`, `makeup`, `blowdown` and `condensate`.

In most cases, however, knowledge of which tags may be appropriate to add to an entity largely depends upon domain familiarity and informal conventions. For example, Haystack point entities (sensors, setpoints and commands) often have a "what" tag (e.g. `air`), a "measurement" tag (e.g. `flow`) and a "where" tag (e.g. `discharge`). However, this is not a hard and fast rule, and many of the tag sets in Haystack's documentation break with this convention. Consequently, there is no clear notion of how concepts can be meaningfully extended or generalized, which limits the extensibility of Haystack.

Inconsistent Modeling Practices

Due to the lack of formal tag composition mechanisms and a well-formed class organization, there is substantial variance in how users of Haystack model the same concept. These inconsistent modeling practices limit the semantic interoperability of the Haystack model.

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

> "Pumps may optionally be defined as either an `equip` or a `point`. If the pump is a [variable frequency drive] then it is recommended to make it an `equip` level entity. However if the pump is modeled [in the BMS as] a simple on/off point as a component within a large piece of equipment such as a boiler then it is modeled as just a `point`.[120]"

However, not all modelers in Haystack choose to make this simplification, especially when there are more points that may relate to the same equipment entity. To query a Haystack model in a consistent way, a user must write complex predicates that take into account the family of possible modeling choices.

Impact of Tags on Metadata Design

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

— presents issues for adoption as an industrial standard and basis for automated analysis and reasoning.

The next 2 sections show that the tradeoff between composability and consistency is tied to the choice to use tags for annotation as well as definition. With an explicit and formal class hierarchy it is possible to design a system that exhibits the composability of simple tags, while retaining the consistency and extensibility of an ontology.

4.3 Ontology Design for Consistent Metadata

This chapter introduces the design and implementation of Brick+, a mostly[3] backwards-compatible edition of the Brick ontology that is built on formal composition rules to achieve desirable properties of tag-based metadata while also improving upon the original Brick design [16, 15]. The heterogeneity of real-world deployments — including systems composed in unusual ways or containing novel or legacy equipment — and the innovation in the building industry mean that extensions to the Brick ontology are a continuous necessity. Experiences with the original Brick design revealed that simply providing a class organization, even with textual descriptions, is not sufficient to ensure consistent extensions of the ontology.

Brick+ improves on the original Brick design through the incorporation of the following features:

- Semantic Class Lattice: classes in Brick+ are not just named sets like in many OWL-based ontologies. Instead, their definitions are qualified by a number of composable semantic properties. The result is a *lattice* of classes that provides more structure to Brick extensions than a simple hierarchy.

- Compatibility between tags and classes: Brick+ incorporates Haystack-style tags as definition-free annotations on Brick entities. Entities inherit tags from the classes they instantiate; the implementation also supports the inference of a Brick class from a collection of tags. This can be done using both an *exact* and an *approximate* matching algorithm.

- Validation and tooling support: Brick+ defines constraints that enable the automatic validation of Brick models, and contains additional annotations that inform external tooling that assists in the development and validation of Brick models.

The implementation of these features requires careful appliation of two different modeling languages — OWL RL and SHACL — and the invention of several design patterns. Brick+ has been fully implemented and adopted as the authoritative Brick implementation at time of writing[4]. The intuition and formal design of each of these features is presented below, followed by an overview of their implementation using semantic web modeling languages.

[3] A few mechanical and easily automatable changes must be made to existing Brick models.

[4] Brick+ features have been released in Brick versions 1.1 and 1.2.

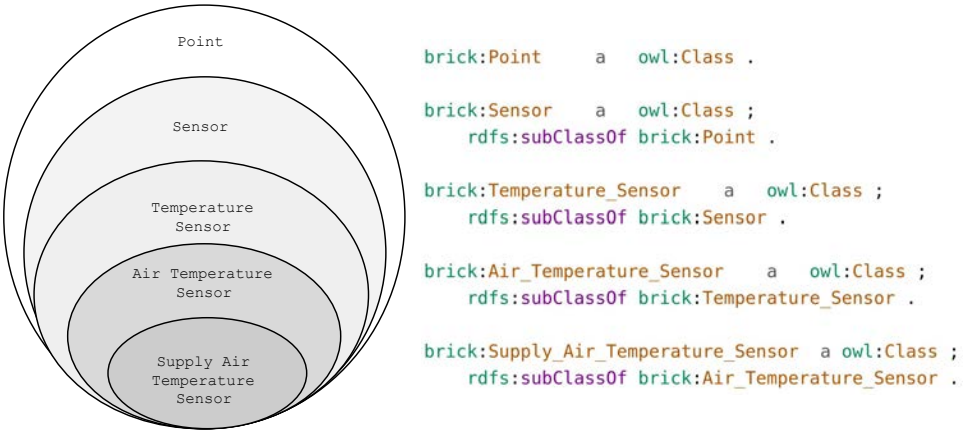


Figure 4.2: An OWL-based ontology (right) encodes classes as named sets (left); a subclass relationship between two classes means that all the members/instances of the subclass are also members/instances of the superclass.

Semantic Class Lattice

OWL-based ontologies are typically built from classes related to each other through a subclass relationship (Figure 4.2) — `rdfs:subClassOf`. Classes are named sets, but may have other properties associated with them. For example, the `skos:definition` property from the Simple Knowledge Organization System ontology is a common choice for relating a class declaration to a text-based human-readable definition; alternatively, the `rdfs:seeAlso` property might point an interested user at a citation or other related resource.

However, these additional properties do not capture the semantics of the class in a machine-interpretable manner. Further, the subclass relationship only indicates members of one class are a subset of the members of another class. It does *not* capture what distinguishes the members of the subclass from the members of the superclass. Without the ability to capture this information, it is difficult to ensure that extensions to the class organization will be done in a compatible manner. Modelers may choose a different class name for the same concept, and thus ruin any interoperability between models produced using different extensions. Even though a formalized and well-formed class organization addresses the ambiguities and inconsistencies inherent to the Haystack design, it is not enough to guarantee semantic interoperability as an ontology is extended.

A *semantic class lattice* addresses this issue by removing the requirement that extensions to the class hierarchy must use the same name for the same concept. This is accomplished by augmenting the subclass relationship with additional semantic information which captures the distinguishing characteristics between two classes connected by a subclass relationship. This semantic information is *composable* — different combinations of semantic properties

indicate different concepts. The composition rules determining which semantic properties have semantic meaning is implicitly defined by the class organization. This means that the set of valid compositions of semantic properties are exactly those that have been explicitly equated to a member of the class organization. In this design, a class is defined as a unique set of semantic properties; two classes are equivalent if they have the same set of associated semantic properties. Under a well-defined set of semantic properties, extensions to the class organization can be defined as novel combinations of semantic properties: a subset relationship between two sets of semantic properties implies a subclass relationship.

Formally, a class lattice is based on a set of semantic properties $\mathcal{S}$. A semantic property is a tuple (p_i, v_i) of a property name $p_i \in \mathcal{P}$ (where $\mathcal{P}$ is the set of all property names) and a value $v_i \in \mathcal{V}(p_i)$ from the domain of that property. Each class c_j in the class organization $\mathcal{C}$ is equivalent to a unique set of semantic properties: $c_j \equiv s_j \in \mathcal{S}^n$, where $\mathcal{S}^n$ is a set of n semantic property tuples drawn from the set of semantic properties $\mathcal{S}$. Because each c_j has a unique s_j, this implies that if $s_j = s_k$, then $c_j = c_k$.

The key assumption in the class lattice design is that any two modelers will reach for the same set of semantic properties to describe the same concept. At first glance, this approach is reminiscent of the tag-based metadata model described in §4.2; however, the semantic property design improves on the tag approach in two ways.

1. The use of a structured tuple (p_i, v_i) instead of a simple tag makes it possible to more precisely define the semantics of the property name p_i. In the Brick+ implementation, property names have specific meanings that make their intended usage clear.

2. The property name contextualizes the intended meaning of the property value v_i. The tag-based model cannot capture the relationship between two tags, and Haystack in particular lacks any enforcement or specification of the domain of possible values for all "value tags". In contrast, the key-value nature of semantic properties provides a richer mechanism for capturing the intent of a given value.

These two improvements make it possible for a modeler to choose a *consistent* and *interpretable* set of semantic properties with which to define a new class. §4.4 describes in detail a few of the semantic properties defined in Brick+ and demonstrate how the class lattice structure above can be implemented using the OWL and SHACL ontology definition languages.

Tag/Class Compatibility

Brick+ also embeds a tag-based metadata model which supports the annotation of entities using Haystack-style tags. The tag-based metadata model improves upon the Haystack model in two significant ways.

First, Brick+ attaches a *unique* set of tags to each class. This provides a "dictionary" of which tag sets have clear, well-known definitions. By removing the need to have tags act as freely-composable annotations (a role now served by the semantic properties described

above), tags can be used as simple annotations. Instances of Brick+ classes inherit the appropriate tags, allowing the tags to be used as a discovery mechanism for both classes and entities.

Second, Brick+ supports the *inference* of classes from a tag set; that is, given an entity's tag set, Brick+ can automatically derive an appropriate Brick+ class for the entity. This inference can be done in both an exact (i.e. which Brick+ class exactly matches the given tag set) and an approximate (i.e. which is the *most likely* class for the given tag set) manner. In this way, the Brick+ class organization determines which sets of tags have a valid and interpretable meaning.

The exact matching inference filters out the entities that do not have an interpretable set of tags. Crucially, the exact matching inference does not rely solely upon the subset relationship between two sets of tags. As discussed in §4.2, the subset relationship can lead to situations where the same set of tags indicates two disjoint classes, resulting in an uninterpretable model. Even though the Brick+ class organization defines the set of valid tag sets, it is still possible for a subset of a tag set to imply an incompatible class. Brick+ circumvents this issue by including the *cardinality* of the tag set as an inference requirement.

Consider the same two classes and concepts that illustrated this issue in the Haystack data model: `Air Flow Setpoint` (the desired cubic feet per minute of air flow) and `Max Air Flow Setpoint` (the maximum allowed air flow setpoint). In Brick, `Air Flow Setpoint` would be identified by the `air`, `flow` and `setpoint` tags, and `Max Air Flow Setpoint` would be identified by the `air`, `flow`, `setpoint` and `max` tags[5]. In order to infer a `Air Flow Setpoint` classification for an entity with the `air`, `flow` and `setpoint` tags, the entity would also need to have *exactly three associated tags*. These two conditions (posessing the required tags and possessing the required *number* of tags) are sufficient to guarantee that shared tag sets do not result in the inference of incorrect or inappropriate classes.

The approximate matching inference method is helpful for automating the production of Brick+ models from other metadata representations such as Haystack. The implementation of approximate tag matching and its use in producing Brick+ models will be discussed in depth in Chapter 5.

Brick+ Model Validation

The formal structures in Brick+ also enable the validation of Brick+ models. There are two kinds of validation supported by Brick: *correctness* validation, and *idiomatic* validation.

Correctness validation ensures that the construction of a Brick+ model is logically sound and does not imply any internally inconsistent information. Importantly, "correctness" in this sense does not capture whether or not the Brick+ model is "up-to-date" with a building (that is, whether the Brick+ model a correct representation of the building); §6.3 explores this

[5]Brick+ defines its own set of tags to avoid unnecessary abbreviations like `sp` for "setpoint". Most Brick+ tags have an analogous version in Haystack

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

Figure 4.3: An example of a logical violation in an instance of a Brick+ model.

subject in more depth. Instead, this validation ensures that Brick+ is being used correctly for the statements that are in the model.

Consider the set of RDF statements in Figure 4.3. Lines 1-6 are a partial implementation of the Brick+ ontology. Lines 1-3 specify that any subject of the `brick:isPointOf` property is implied to be an instance of the `brick:Point` class. Lines 5-6 state that the set of instances of `brick:Point` is disjoint with the set of instances of `brick:Equipment`; that is, no entity can be both a Point and an Equipment.

Lines 8-10 define the Brick+ model and contain the violating statements. Lines 8 and 9 define two pieces of equipment. Line 10 introduces the logical violation: the use of the `brick:isPointOf` property implies that `building:vav1` is a member of `brick:Point` which conflicts with the statement on line 9 that `building:vav1` is a member of the disjoint class `brick:Equipment`. It is not possible to determine which statement is erroneous without external information about how the model was intended; however, a generated list of the violating statements can help in the correction of a model. In this example, the two violating statements are line 9 (`building:vav1` cannot be a `brick:Equipment` if it is to be a `brick:Point`) and line 10 (which implies `building:vav1` to be a `brick:Point`).

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

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

- all VAVs in an instance should refer to an upstream AHU and a downstream HVAC

```
bsh:HasPointShape   a   sh:NodeShape ;
    sh:targetObjectsOf brick:hasPoint ;
    sh:class    brick:Point ;
    sh:message "Objects of the hasPoint relationship should be Points" ;
.
bsh:IsPointOfShape   a   sh:NodeShape ;
    sh:targetSubjectsOf brick:isPointOf ;
    sh:class    brick:Point ;
    sh:message "Subjects of the isPointOf relationship should be Points" ;
.
```

Figure 4.4: SHACL node shapes validating use of the Brick+ `hasPoint` and `isPointOf` relationships

zone

- all VAVs of a particular make and model should have five associated monitoring and control points
- all temperature sensors should be reporting in Celsius

Validation for Brick+ is implemented in SHACL [81] (Shapes Constraint Language), a W3C specification for describing constraints on RDF graphs. Recall that SHACL describes a graph consisting of *shapes*: shapes have a set of constraints and rules, which describe tests for existing triples or the conditions for generation of new triples, and a *target* which specifies parts of a data graph are subject to the constraints[6]. SHACL defines two kinds of shapes: node shapes and property shapes. These define constraints for nodes and edges (respectively) in an RDF graph.

Correctness Validation

Brick+ shapes focus on the proper usage of relationships. Because Brick+ does not specify any minimum level of information, the only way to incorrectly use a Brick+ class is to connect to it with an inappropriate relationship. For each of the relationships in Table 4.2, Brick+ defines a node shape which targets the subjects or objects of the relationship. A shape targets the subjects of the relationship if there are restrictions on its *domain*, and targets the objects of the relationship if there are restrictions on the *range*. The shape then specifies the intended type of the target node. Figure 4.4 contains the SHACL shape for the `hasPoint` relationship. Because correct use of the `hasPoint` relationship only requires that the object is a `Point`, the shape itself is very simple: `sh:targetObjectsOf` identifies that the shape applies to all nodes that are objects of triples whose predicate is `hasPoint`; `sh:class` constrains the type of those nodes to be Brick's `Point` class.

[6] §2.4 provides more background on SHACL

```
bsh:IsPartOfShape   a   sh:NodeShape ;
    sh:targetSubjectsOf brick:isPartOf ;
    sh:xone (
        [
            a   sh:NodeShape ;
            sh:class    brick:Equipment ;
            sh:property [
                sh:path     brick:isPartOf ;
                sh:class    brick:Equipment ;
            ] ;
        ]
        [
            a   sh:NodeShape ;
            sh:class    brick:Location ;
            sh:property [
                sh:path     brick:isPartOf ;
                sh:class    brick:Location ;
            ] ;
        ]
    ) ;
.
```

Figure 4.5: A SHACL shape enforcing the two possible uses of the `isPartOf` relationship.

For other relationships, the usage and implementation is more complex. The `isPartOf` relationship can be used in two ways to indicate that equipment can be composed of other equipment, or that locations can be composed of other locations. A model which indicates that some locations are composed of equipment (or vice versa) is incorrect. Figure 4.5 implements the SHACL shape for this relationship. `sh:xone` specifies a list of constraints, of which *exactly one* must be met in order for the target node to validate. The two constraints in the list have the same structure: they each specify the expected type of the subject *and* object of the relationship. The former is handled in the same way as the simpler shape in Figure 4.4, but the second constraint encoded as a property shape.

The library of correctness shapes for each of the Brick+ relationships is included in the Brick+ distribution where it can be easily accessed by tooling and applications. However, the correctness shapes constrain very little about the contents of a Brick+ model — for example, an empty Brick+ model is valid, but is not useful for executing any sort of data analytics. In order to specify requirements on the desired content in a Brick+ model modelers should use non-normative idiomatic validation shapes.

Idiomatic Validation

Idiomatic validation is also based on SHACL node shapes, but differs from the above in its more liberal use of other SHACL features — offering more expressive requirements — and the focus on targets qualified by class rather than relationship. Shapes for idiomatic validation are simply all shapes that are not used for correctness. Presented here are several

```
bsh:BadVAVShape   a   sh:NodeShape ;
  sh:target   brick:VAV
  sh:property [
    sh:path brick:hasPoint ;
    sh:class brick:Supply_Air_Temperature_Sensor ;
    sh:minCount 1;
    sh:maxCount 1;
  ], [
    sh:path brick:hasPoint ;
    sh:class brick:Supply_Air_Temperature_Setpoint ;
    sh:minCount 1;
    sh:maxCount 1;
  ]
.
```

Figure 4.6: An *erroneous* SHACL shape for defining two required points for VAVs

illustrative examples of common kinds of idiomatic validation shapes, and how they may be used.

Equipment Properties and Relationships: Node shapes can express what Brick+ metadata *must* be attached to entities of a certain type. For example, in a particular building it may be the case that all variable air volume box controllers have a standard set of points exposed in the BMS. By encoding these expectations in an idiomatic validation shape, tooling can ensure that the Brick+ model actually describes the points and relationships that are known to exist. Implementing these shapes requires using *qualified* SHACL constraints — simply using `sh:path` in property shapes to encode the expected type of related points results in unintuitive validation errors. Figure 4.6 contains an example of how not to do this: when two different `sh:path`-based property shapes are associated with a node shape, then fulfilling one means violating the other because the same entity which is the object of the relationship cannot be an instance of two (disjoint) classes. Figure 4.7 shows the corrected node shape, which uses `sh:qualifiedValueShape` to indicate that the property shape only applies to those entities that match the node class pointed to by `sh:qualifiedValueShape`. `sh:qualifiedMinCount` and `sh:qualifiedMaxCount` are analogous to the classic `sh:minCount` and `sh:maxCount` constraints, but only apply in the same context as the `sh:qualifiedValueShape`.

Site-wide Requirements: In a similar manner as the above, idiomatic validation can specify mandatory enumerations of equipment and other expected properties of a particular site. For example, a modeler may use metadata from external data sources to author a shape targeting an instance of a `brick:Building` which has exactly 10 VAVs, two AHUs, four floors, and 37 rooms. This is possible to model in SHACL using the modeling approach above: these more complex constraint structures are necessary because Brick+ only defines a small number of relationships which may be used in different contexts.

Non-validating use of shapes: Shapes can also inform other use cases beyond val-

```
bsh:GoodVAVShape   a   sh:NodeShape ;
  sh:target   brick:VAV
  sh:property [
    sh:path brick:hasPoint ;
    sh:qualifiedValueShape [
      sh:class brick:Supply_Air_Temperature_Sensor ;
    ] ;
    sh:qualifiedMinCount 1 ;
    sh:qualifiedMaxCount 1 ;
  ], [
    sh:path brick:hasPoint ;
    sh:qualifiedValueShape [
      sh:class brick:Supply_Air_Temperature_Setpoint ;
    ] ;
    sh:qualifiedMinCount 1 ;
    sh:qualifiedMaxCount 1 ;
  ]
.
```

Figure 4.7: The correct SHACL shape for defining two required points for VAVs.

idation. First, because shapes can be queried just like any other RDF graph, they serve as a point of reference for bootstrapping an understanding of what data may be available in a given Brick+ model. This can be used to populate documentation pages or other visualizations. Second, shapes can be used as templates for populating Brick+ models: the required property shapes of a node shape inform the information that needs to be provided by the user, and node shape itself contains the relevant metadata necessary to automatically instantiate a Brick+ entity of the right type and with the required properties.

4.4 Brick+ Formal Implementation

The features of Brick+ that make it suitable for providing consistent metadata are implemented in two different ontology languages: OWL-RL and SHACL. This section covers the implementation of Brick+ and explains how it incorporates novel ontology features into a traditional ontology design. The result is a practical ontology which enables the consistent modeling of heterogeneous systems.

Brick+ Overview

Brick+ defines a family of OWL classes and properties using the OWL ontology language. Recall that a class represents a named set; entities that are instances of a class are members of that set. Brick+ classes categorize and define the behavior of the physical, virtual and logical entities found in buildings. They are implemented as instances of `owl:Class` and are organized into a class structure using the `rdfs:subClassOf` predicate. Brick+ defines three

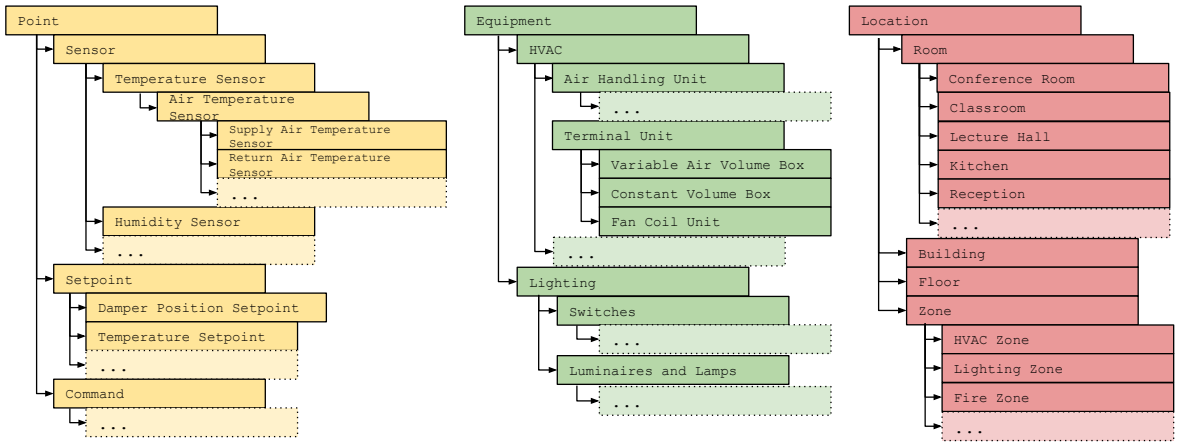


Figure 4.8: The three main class organizations for Brick+ entities

```
brick:Temperature_Sensor a owl:Class ;
    rdfs:label "Temperature Sensor" ;
    rdfs:subClassOf [ owl:intersectionOf ( _:has_Point _:has_Sensor _:has_Temperature ) ],
        brick:Sensor ;
    owl:equivalentClass [ owl:intersectionOf ( [ a owl:Restriction ;
                          owl:hasValue brick:Temperature ;
                          owl:onProperty brick:measures ] ) ] ;
    skos:definition "Measures temperature: the physical property of matter that"
        " quantitatively expresses the common notions of hot and cold"@en ;
    brick:hasAssociatedTag tag:Point,
        tag:Sensor,
        tag:Temperature .
```

Figure 4.9: The Brick+ definition of a temperature sensor.

(disjoint) types of entities: Locations, Equipment and Points. Location classes encompass physical locations such as buildings, floors and rooms as well as logical spaces such as sites (zero or more buildings) and zones (groups of spaces related to different building subsystems). Equipment classes organize the physical assets in a building that are metered, monitored and controlled as part of various building subsystems. Point classes define any source or sink of data in a building; importantly, this is not limited to the data that is accessed through a BMS, but rather any kind of digital data. Together, these three categories of classes cover the majority of relevant concepts for data-driven building applications [54, 16, 15]. Figure 4.8 illustrates a subset of the Brick+ class structure: broader and more generic classes are closer to the top of the organization, and more specific classes are closer to the bottom.

Each Brick+ class definition (e.g. Figure 4.9) has 6 components:

Relationship	Definition	Domain	Range	Inverse	Transitive?
hasLocation	Subject is physically located in the object	*	Location	isLocationOf	yes
feeds	Subject conveys some media to the object in some sequential process	Equipment Equipment	Equipment Location	isFedBy	no
hasPoint	Subject has a monitoring, sensing or control point given by the object	*	Point	isPointOf	no
hasPart	Subject is composed – logically or physically – in part by the object	Equipment Location	Equipment Location	isPartOf	yes

Table 4.2: Brick+ relationships for associating entities with each other

- A unique URI in the Brick+ name space (line 1 of Figure 4.9)
- A human-readable label identified by `rdfs:label` (line 2 of Figure 4.9)
- A human-readable definition identified by `skos:definition` (lines 8-9 of Figure 4.9)
- One or more parent classes identified by `rdfs:subClassOf` (line 3-4 of Figure 4.9)
- Any associated tags, identified by `brick:hasAssociatedTag` (line 10-12 of Figure 4.9)
- Semantic properties which define the class. These use a variety of property names, as explored in the next section (line 5-7 of Figure 4.9)

Several of these are simply annotations on the class definition. `rdfs:label`, `skos:definition` and `brick:hasAssociatedTag` can be queried by tooling and applications, but do not affect or contribute to the formal implementation of Brick. The other properties, discussed below, are interpreted by a *reasoner* to insert inferred information into the graph.

Brick+ relationships connect entities together to form the model representing a particular building. Brick+ defines a small, orthogonal set of relationships — implemented as OWL object properties — which capture composition, topology, telemetry and other associations between entities that are required for authoring data-driven applications. These relationships, carried over from the original Brick+ release, are listed in Table 4.2.

However, the class organization and relationship definitions alone are not enough to guarantee the consistent extensibility of the Brick+ ontology. To address these issues, Brick+ includes formal definitions of semantic properties and tags; these enable the structured extension of Brick+ and allow new and richer kinds of metadata to be associated with Brick+ entities.

Brick+ Tag Implementation

In Brick, tags are entities which are instances of the `brick:Tag` class. Brick+ tags have no associated definition, and thus are not designed to provide any additional semantic context to classes or entities. Tags are associated with Brick+ classes via the `brick:hasAssociatedTag`

```
1  brick:Temperature_Sensor    a   owl:Class ;
2      rdfs:subClassOf [ # beginning of the OWL intersection class definition
3          owl:intersectionOf  ( # the list of tag classes
4              [ a owl:Restriction ;  # restriction class for "Temperature" tag
5                owl:hasValue tag:Temperature ;
6                owl:onProperty  brick:hasTag ]
7              [ a owl:Restriction ;  # restriction class for "Sensor" tag
8                owl:hasValue tag:Sensor ;
9                owl:onProperty  brick:hasTag ]
10             [ a owl:Restriction ;  # restriction class for "Point" tag
11               owl:hasValue tag:Point ;
12               owl:onProperty  brick:hasTag ]
13         )
14     ], brick:Sensor .
```

Figure 4.10: Part of the `brick:Temperature_Sensor` class definition from Figure 4.9, showing the unoptimized full implementation of the OWL intersection class

relationship, and are associated with Brick+ entities via the `brick:hasTag` relationship. These two relationships differentiate if a tag is intended as part of a definition (for classes) or to facilitate discovery (for entities). Having two different relationships also makes it possible for a reasoner to differentiate between the two use cases of tags.

Brick+ uses a reasoner to automatically add the appropriate tags to all instances of a Brick+ class (referred to as class→tag inference). To implement this, each Brick+ class definition has a `rdfs:subClassOf` relationship to a unique OWL intersection class which encodes the required tags for that class. An OWL intersection class is defined by the conjunction of membership in other classes: if an entity is a member of each of the indicated classes, it is inferred to be a member of the intersection class. In the Brick+ tag implementation, the constituent classes to the OWL intersection class are OWL restriction classes which individually encode the tag that is to be inherited. OWL restriction classes are defined by a set of conditions; an entity is inferred to be an instance of a restriction class if it meets all of the conditions.

Three restriction class definitions can be seen in Figure 4.10 on lines 4-6, 7-9 and 10-12. Each of the classes is characterized by a single condition which is that the entity has a given tag as the value of `brick:hasTag`. By making a Brick+ class a *subclass* of the OWL intersection class, any instance of the Brick+ class is also inferred to be a member of the intersection class, and thus also a member of each of the constituent classes. The inheritance of the OWL restriction classes enables the reasoner to attach the appropriate tags to the Brick+ entity.

One optimization that Brick+ includes in the implementation is the re-use of tag restriction classes. In many ontologies, the OWL restriction classes and OWL intersection classes are anonymous classes called blank nodes; these are typically used and defined in the same place in the file (such as in Figure 4.10). This results in a large number of duplicate restriction

```
1  bsh:Temperature_Sensor_TagShape a sh:NodeShape ;
2      sh:rule [ a sh:TripleRule ;
3              sh:condition _:has_Point_condition,
4                  _:has_Sensor_condition,
5                  _:has_Temperature_condition,
6                  _:has_exactly_3_tags_condition ;
7              sh:object brick:Temperature_Sensor ;
8              sh:predicate rdf:type ;
9              sh:subject sh:this ] ;
10     sh:targetSubjectsOf brick:hasTag .
```

Figure 4.11: SHACL-AF implementation of tag→class inference for the Brick+ `Temperature_Sensor` class. Figure 9.2 in the Appendix contains the full expanded shape.

classes: each class that shares a tag will have its own definition of a restriction class encoding the inference of that tag. A large number of restriction classes can severely impact the performance of the reasoner. Brick+ ameliorates this performance issue by defining a single restriction class for each tag, which drastically reduces the number of restriction classes that must be processed by the reasoner. For clarity, these are often named `_:has_<tag name>`[7]; examples of these can be found on line 3 of Figure 4.9.

Class→tag inference is possible to implement in OWL without encountering the ambiguous classification problems exhibited by Haystack because it only adds tags using unambiguous class declarations, rather than using tags to infer a class. However, it is necessary to incorporate a closed-world model in order to perform tag→class inference. A closed-world model allows the reasoner to check for which tags and properties are *not* associated with a given entity and reason about the cardinality of certain properties. In particular, Brick+ uses SHACL's advanced features (SHACL-AF) to implement tag→class inference. SHACL-AF provides reasoning capabilities that are based on the closed-world assumption [80].

The SHACL shapes that implement this feature are similar in structure to the OWL implementation for class→tag inference. Brick+ defines one SHACL shape for each class. Each SHACL shape has a `sh:rule` clause and targets subjects of `brick:hasTag`. SHACL rules, identified by the `sh:rule` predicate, consist of a set of conditions and a single triple that is added to the model if all the conditions are met. For tag→class inference, each rule requires that the required tags are associated to the entity and that the *number* of associated tags is equal to the number of associated tags. Together, these conditions are sufficient for a rule to fire only if an entity has exactly the required tags, which removes the ambiguous classification error exhibited by Haystack. Figure 4.11 contains the SHACL implementation of tag→class inference for the `brick:Temperature_Sensor` class. The blank node optimization from class→tag inference is also applied here.

[7]The _ prefix indicates a blank node

Substances and Quantities

To support richer semantic definitions of classes, Brick+ defines a set of *substances* and *quantities* which can be used to capture the behavior and purpose of a point or equipment. A *substance* is any physical medium that can be observed or manipulated, including air, water, light, electricity and gas. Brick+ defines several more specific kinds of substances to clarify the phase of matter or the purpose within a larger subsystem. For example, the definition of water has several more specific flavors including ice and water vapor (phases of matter), but also potable water, supply water, return water and other subsystem-specific uses of water.

In Brick, substances are instances of the `brick:Substance` class rather than being classes themselves. This simplifies their usage by removing the need to create an instance of a substance that can be related to a Brick+ class or entity. Even though substances are instances rather than classes, it is still helpful to organize them. Brick+ uses the Simple Knowledge Organization System's `skos:broader` and `skos:narrower` properties to relate substances as being more general or more specific than each other. These properties are not interpreted by a reasoner; they only provide organization to the substances to facilitate discovery.

A *quantity* represents a quantifiable property of a substance. The set of quantities defined in Brick+ is based on QUDT's QuantityKind class [75], and includes quantities such as thermodynamic temperature, active power, volume and volumetric flow. By linking Brick's quantities to those defined by QUDT, Brick+ can also embed suggestions for units of measure that may be associated with points. For example, any Brick+ point that is associated with the `brick:Temperature` quantity (representing thermodynamic temperature) can be automatically inferred to support units of Celsius, Fahrenheit or Kelvin. Like substances, quantities are instances of a root class (`brick:Quantity`) rather than being classes themselves.

Even though substances and quantities may share the same names as tags (e.g. `brick:Air`, the substance, and `tag:Air` the tag), they are placed into different namespaces in order to differentiate which terms have significant semantic meaning and which are simply annotations.

Semantic Properties

Brick+ defines a family of semantic properties that augment the `rdfs:subClassOf` relationship with additional metadata about the behavior and purpose of entities. Recall that semantic properties are a tuple (p_i, v_i) where p_i is a property name and v_i is the property value. Many semantic properties have a value that is drawn from the dictionary of substances and quantities defined by Brick. The property name qualifies the nature of the relationship between the class definition and the substance or quantity. Table 4.3 defines a few of the property names defined in Brick+ as well as their ranges (the type of the property value) and domains (the type of the entity that has the semantic property).

Property Name	Definition	Domain	Range
measures	Identifies the quantity *or* substance that is measured by a point	`brick:Sensor`	`brick:Substance` `brick:Quantity`
regulates	Identifies the quantity *or* substance that is regulated by an equipment	`brick:Equipment`	`brick:Substance` `brick:Quantity`
hasInputSubstance	Identifies a substance that flows into an equipment	`brick:Equipment`	`brick:Substance`
hasOutputSubstance	Identifies a substance that flows out of an equipment	`brick:Equipment`	`brick:Substance`

Table 4.3: A set of semantic property name definitions

Each Brick+ class is bound to a unique set of semantic properties. This is implemented as an OWL intersection class which models the conjunction of an entity having each of the required semantic properties; each of the constituent classes is an OWL restriction class. Due to this construction, each Brick+ class has a *superset* of the semantic properties of its parent. The semantic properties are assigned such that classes with $n + 1$ semantic properties are subclasses of classes with an n-sized subset of those semantic properties. The `rdfs:subClassOf` relationships between classes make this relationship explicit. Figure 4.12 depicts the resulting lattice for a subset of the `brick:Point` class organization.

Consider the example of the `brick:Temperature_Sensor` class depicted in Figure 4.9. This class is characterized by one semantic property: a `brick:Temperature_Sensor` *measures* the temperature quantity of some yet-unspecified substance. Lines 5-7 of Figure 4.9 contain the definition of this single OWL restriction class and the encapsulating OWL intersection class.

Importantly, and unlike tags, semantic properties are *interchangeable* with the class definition. For this reason, the implementation relates the OWL intersection class to the Brick+ class using the `owl:equivalentClass` predicate. This informs an OWL reasoner that it can infer the semantic properties for an entity from the class definition as well as infer the class from a set of semantic properties. The semantic properties for a class are inherited all entities which are instances of that class. This allows semantic properties to be used not just for structuring the extensibility of Brick's class organization, but also for discovering entities using richer annotations than simple tags.

Entity Properties

Classes provide helpful organization for groups of entities, but there are many potential uses of a Brick+ model which require additional information about individual entities. To model this metadata, Brick+ introduced a new *entity properties* component to the ontology. Entity properties are similar to semantic properties in that they are a tuple (p_i, v_i) where p_i is the property name and v_i is the property value, but differ in two important ways:

1. The values of entity properties are structured objects, rather than single values

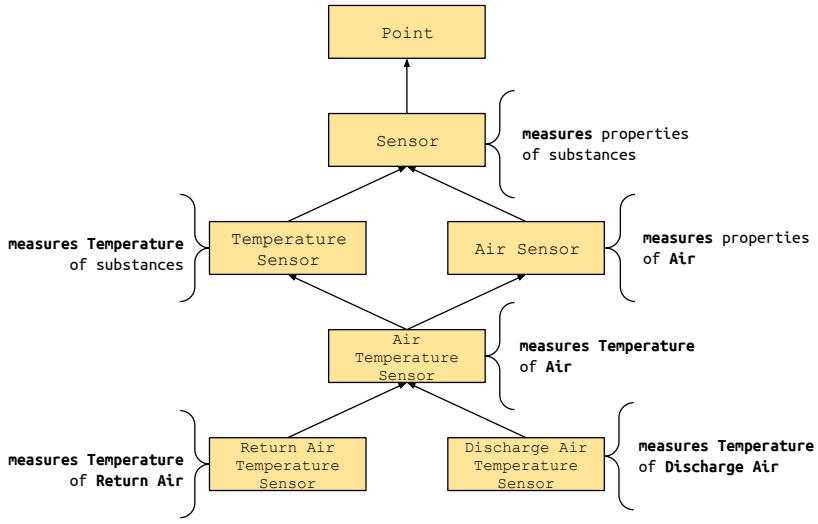


Figure 4.12: A subset of the Brick+ class lattice for sensors, showing the semantic properties which characterize each class

2. Entity properties are applied directly to entities, rather than inferred or inherited from class definitions

The values of entity properties are structured objects modeled in RDF. The specific structure of the object is governed by a SHACL shape which is indicated by the `rdfs:range` of the entity property name. The advantage of having a structured object as the value of an entity property is the value can be *reified* with additional metadata. This allows the definitions of entity values to be extended or further reified as the Brick+ ontology evolves. The object itself is modeled as a URI (often but not necessarily a blank node) and the reifing metadata is modeled as RDF predicates and objects of that URI.

Reifying a value means to attach additional descriptions and metadata to that value that are not captured by the value itself. For example, consider modeling the floor area of a room. The floor area itself is a simple scalar quantity, but the model can be made more helpful and complete by annotating the quantity with the engineering units and whether or not the quantity represents net or gross floor area. Other annotations may be the time at which the observation or measurement was made or an estimation of its accuracy or certainty.

Separating the definitions of the entity property value (the SHACL shape) and the property name makes it possible to reuse the same kind of value in different contexts. The semantic meaning of the value is determined by the property name. The semantics of the property name can be further refined by creating new *subproperties* (indicated with `rdfs:subPropertyOf`), which inherit definitions from any parent properties.

Most SHACL shapes defining the entity property values follow one of two design patterns. The first pattern is for modeling scalar values, such as floor area, room volume, or the

```
1  brick:area a brick:EntityProperty ;
2      rdfs:domain brick:Location ;
3      rdfs:range brick:AreaShape ;
4      skos:definition "Entity has 2-dimensional area" .
5
6  brick:AreaShape a sh:NodeShape ;
7      sh:property [ a sh:PropertyShape ;
8              sh:datatype xsd:float ;
9              sh:minCount 1 ;
10             sh:path brick:value ],
11         [ a sh:PropertyShape ;
12             sh:in ( unit:FT2 unit:M2 ) ;
13             sh:minCount 1 ;
14             sh:path brick:hasUnit ] .
15
16 :x  a   brick:Floor ;
17     brick:area [
18         brick:value 100^^xsd::float ;
19         brick:hasUnit unit:FT2 ;
20     ] .
```

Figure 4.13

nominal rated voltage of a motor: in these cases, the entity property must capture the value, (identified with a `brick:value` property), the units of measure (identified with a `brick:hasUnit` property) of the property, and the expected datatype (e.g. string, integer, positive integer, floating point number). The second pattern is for modeling enumerated values. Here, the value of `brick:value` can only have one of a fixed number of values; the SHACL shape contains the enumeration and definition of these values.

Not all entity properties adhere to these idioms. For example, the `brick:aggregate` entity property, which defines the aggregation method and window of the data associated with a point, defines its own `brick:aggregationFunction` and `brick:aggregationInterval` properties to characterize the value.

Figure 4.13 contains the definition of the `brick:area` entity property name, the `brick:AreaShape` SHACL shape defining the structure of the value, and an example use of the property to model floor area. Note that the entity property definition on lines 1-4 indicates that the property is only to be used with Brick+ location entities. The shape defintiion on lines 6-14 enumerates the possible units (square feet and square meters) and indicates that floor area should be a floating point number.

Entity properties address several long-standing asks from the Brick community. Over the past 5 years of Brick development, there have been many ad-hoc attempts to model static and scalar metadata that is associated with Brick entities. However, until the entity property design above, these independently-developed models were not consistent in their form and definition. As Brick continues to evolve, the family of entity property definitions is likely to grow to cover additional metadata use cases.

Conclusion

In order to provide a consistent metadata model that can describe the heterogeneous settings typical of cyberphysical systems, it is not enough to simply define a dictionary of tags or terms which can annotate data streams. Brick adopts a formal, graph-based model which not only provides a rich and extensible dictionary of concepts, but provides a means for instances of concepts to be related to one another. This chapter demonstrated new ontology design techniques which provide facilities for structured extension of the ontology, ensuring that additions can be done in a consistent and interpretable manner. Brick also provides an alternative and usable implementation of tagging for both entities and concepts, and incorporates a new *entity properties* feature which associates reiified metadata with instances of Brick classes. Together, these features enable the robust modeling of complex cyberphysical systems, and provide a framework for implementing self-adapting software.

Chapter 5

Expressing Self-Adapting Software

> Write once, run everywhere.
>
> *Sun Microsystems*

Self-adapting software leverages structured and semantically-rich descriptions of its deployment in order to automatically configure its operation to that environment. The prior chapter demonstrated how graph-based metadata models, such as Brick, can capture the necessary information to enable self-adapting software; however, the challenge remains how to actually write self-adapting software. How does software express what the requirements are for its operation, and how its structure and execution should react to different configurations of its environment? What are the supporting systems, databases and programming runtimes that enable the efficient execution self-adapting software? How can the potential complexity of self-adapting software be automated or otherwise abstracted away from the programmer?

Two models for expressing and executing self-adapting software are examined. First, the chapter details a *staged* execution model for self-adapting software. Software describes the data and metadata requirements for its operation in an *application manifest*. To determine if and how software can execute in a given deployment, the software's manifest is evaluated on the metadata model for that deployment. The results of this evaluation provide the application with the information it needs to determine if it can execute and, if it can execute, how to configure itself and adapt its logic to the deployment. While straightforward to implement, the programming model outsources much of the complexity of self-adaptation logic to the user. The chapter then contrasts the staged programming model with an *interactive* approach that abstracts away much of the bookkeeping associated with writing self-adaptation logic.

Definition 5.0.1 (Environment)**.** *An* environment *is the cyberphysical context in which a piece of software is running. This includes, but is not limited to, the networked resources available to the software, the structure and topology of the cyberphysical subsystems affecting*

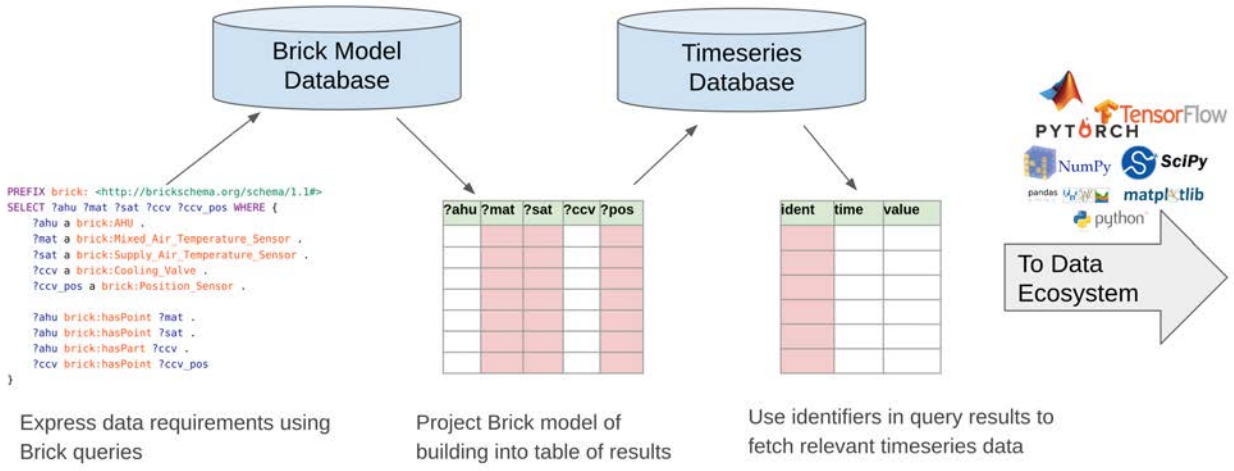


Figure 5.1: Logical workflow for application configuration and execution

the environment (such as HVAC, lighting and electrical subsystems in buildings), and any historical telemetry about the environment.

5.1 Using Metadata for Configuration

The kinds of rich semantic metadata detailed in §4 can inform the execution of self-adapting software in several ways. An effective programming model for self-adapting software must:

- allow software to access the metadata representation of an environment to determine if the environment is appropriate and if there is sufficient metadata for the application to operate
- allow software to adapt its execution with respect to the available metadata about an environment

Metadata-Driven Software Portability

Definition 5.1.1 (Portability)**.** *Portability is the quality of a piece of software that describes the extent to which the software can adapt the execution of its functionality to a given environment without human interaction or configuration. Generally speaking, the more portable a piece of software is, the more environments it may execute in. Non-portable software may only run on 1 or a small number of environments; this is characteristic of most data-oriented cyberphysical software deployed today.*

Self-adapting software queries the metadata model of an environment in order to understand how the software should adapt its behavior. The first task that these queries

accomplish is establishing whether or not the environment is appropriate for the software. This predicate, which is specific to the logic of the application, is evaluated by querying the environment for the kind and structure of the subsystems that are required by the software

Cyberphysical systems are heterogeneous and are characterized by unique, one-off designs. However, while the exact instantiations of these subsystems differ significantly from building to building, most subsystems of a particular type will contain similar kinds of equipment organized into similar topologies. For example, recall from Chapter 2 that conditioning the air in buildings may be performed through air-based HVAC systems (usually with a complex AHU-based system in larger buildings, and simpler RTU-based systems in smaller buildings), or through water-based radiant systems. In each of these systems, a substance (water or air) can be traced through different equipment which regulate or condition different properties of that substance until the substance reaches a point where it interacts with the human-occupied parts of the environment. The functional relationships between these components can be captured in a general manner (Chapter 4).

Self-adapting software will query the environment to identify which kinds of subsystems are installed, and use the results to determine how to adjust the operation. Query results can drive an algorithmic change — such as what kind of model is trained or which fault conditions are tested — but may also lead the application to terminate early due to a lack of available metadata or the presence of a subsystem which is incompatible with the application's intent.

Once the software knows the structure of the environment it is interacting with, it can then query the metadata model for application- and subsystem-specific configuration. For data-driven applications, this includes retrieving the names, units and other salient metadata for any points and other data streams required for operation. This is complemented with contextual information about the data streams, such as which equipment or location they are associated with.

Figure 5.1 illustrates a logical execution model for self-adapting applications in a Brick context. The queries describing the application's semantic and data requirements are evaluated on a Brick model database, which stores the metadata representation of the current environment. The answers to these queries give the application the foreign keys or other identifiers necessary to retrieve historical and live telemetry for any relevant data sources. This bootstrapping process can then forward the resulting metadata and data to the rest of the application.

There are fundamental limits to the portability of an application. The most portable software is an application that requires no metadata about its environment. However, most useful applications require certain assumptions to hold for the application's logic to be appropriate. For example, an application that is detecting broken fans in a building will likely need to know what kinds of data are available for the fans in the building. If a building has no fans, then the application is not appropriate for the building. Further, an application's ability to operate in a certain environment also depends on the metadata available about the environment. An incomplete or out of date metadata model of an environment will also impact how well, or if, software can run in that environment. Techniques and systems for maintaining and managing metadata models over time are described in Chapter 6.

Another crucial component to application portability is the existence of common APIs for accessing the metadata model and telemetry for a building. Past work such as sMAP [40] and recent commercial platforms address how to extract data from cyberphysical environments. Chapter 7 discusses APIs and platforms that support self-adapting software.

SPARQL Queries for Application Configuration

The SPARQL query language is a W3C standard for querying and manipulating RDF graphs, and is an essential tool for accessing the contents of metadata models. Self-adapting software leverages SPARQL features and ontology structure (using the features described in Chapter 4) to account for the *both* the heterogeneity of cyberphysical systems in the environment *and* the heterogeneity in the graph structure describing that environment. This allows effectively-written software to remain somewhat agnostic to the content and structure of the metadata model, which increases portability. SPARQL has a large number of features, but only a subset of them are necessary to support self-adapting software. The overview of SPARQL presented here will focus on the major features of SPARQL 1.1 [66] used in this context.

A SPARQL query has two primary components: the `SELECT` clause which indicates which variable bindings to return from the query execution, and the `WHERE` clause which specifies the predicates on and relationships between variables. A variable is identified with a `?` prefix, e.g. `?x` and `?ahu`. The name of a variable has no semantic significance, though for clarity variables often retain names similar to what they will be bound to.

The `WHERE` clause consists of a set of graph patterns, which are made up of triple patterns and other operators. A *triple pattern* has a similar structure to an RDF triple. It is composed of:

- a *subject*: a variable or URI
- a *predicate*: a variable, URI or *property path* [66]
- a *object*: a variable, URI or literal

Triple patterns describe the parts of a graph that must be matched for the query predicate to be true. The flexibility of this matching is what allows SPARQL queries to execute successfully over heterogeneous graphs. Property paths are a key feature of SPARQL which facilitate this flexibility. A property path is a complex pattern of RDF predicates and query variables which can match transitive closures (0 or more of the same predicate in sequence), optional predicates, alternative predicates, explicit sequences of predicates, and negations of the above. Triple patterns are joined together into a graph pattern by the "`.`" conjunction operator. Many `WHERE` clauses consist of a single graph pattern; however, it is also possible to union multiple graph patterns (where "`.`" is a logical "and", the union is a logical "or") to express more complex query predicates. The programming models explored below differ primarily in how and when SPARQL queries are expressed and executed.

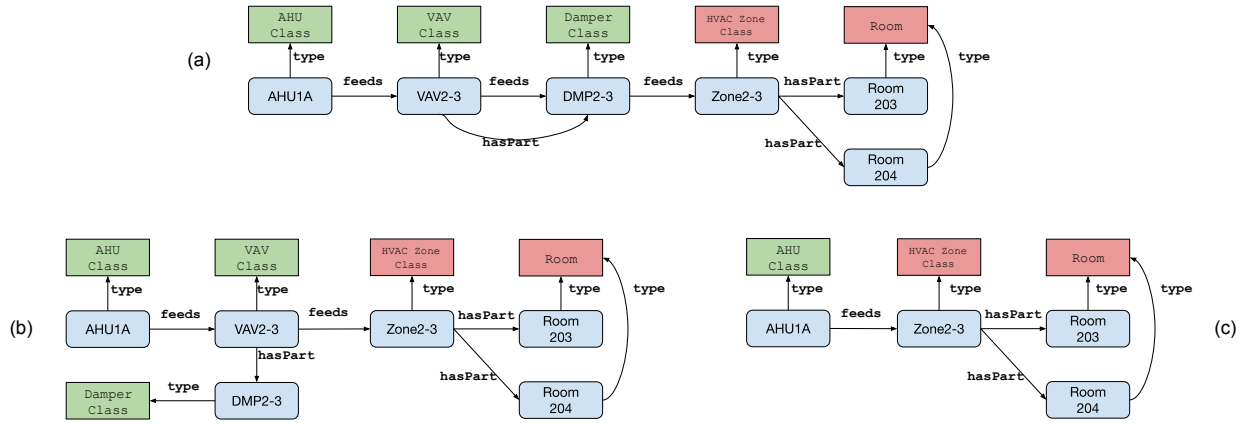


Figure 5.2: Three different Brick models describing the same physical system. It is possible to author a single SPARQL query which will retrieve the names of the AHU and downstream rooms, independent of the specific structure of the graph.

Handling Heterogeneous Graph Structures

To be portable, self-adapting software must be able to find the metadata it requires in many different graphs. Even for the same cyberphysical system, differences in the metadata model graph may arise for several reasons. Primary among these is the availability of information: not all cyberphysical systems have a complete description available that can inform the creation of the metadata model. Another reason graph structures may differ is the effort required to make more complete models, or differences in opinion for how the model should be constructed. The combination of property paths and the small and expressive set of predicates supported by a cyberphysical ontology like Brick allow SPARQL queries to generalize to many different graph structures. Brick's design goals of consistency and interpretability facilitate this task in three concrete ways.

First, the small number of predicates defined by the ontology mean there is a limited number of ways in which components can be related to one another. Because Brick's predicates have orthogonal semantics, it is usually obvious which predicate should or would be used to relate a given pair of entities. For example, entities representing a sequence of equipment in a subsystem will be related to one another using the `brick:feeds` relationship.

Second, the class organization makes it easier for modelers to choose consistent types for the entities they are modeling. Brick currently defines over 700 different classes, each with a human-readable definition which can clarify the meaning of the class. The hierarchical nature of the class organization also makes it possible to discover entities based on more generic types. This removes the need to know the exact type associated with an entity in order to find it with a SPARQL query. For example, an entity with a type of `brick:Discharge_Air_Temperature_Sensor` is also an

```
1 SELECT ?ahu ?room WHERE {
2     ?ahu    a   brick:AHU .
3     ?ahu    brick:feeds+    ?zone .
4     ?zone   brick:hasPart   ?room
5 }
```

Figure 5.3: A SPARQL query which finds the AHU and two rooms in each of the graphs in Figure 5.2

.

instance of `brick:Air_Temperature_Sensor`, `brick:Temperature_Sensor`, `brick:Sensor` and `brick:Point`.

Third, the semantic properties that contextualize class definitions can also be used to discover relevant entities in terms of their *behavior* and *purpose*, rather than the name of their type. For example, it may be more natural for a query author to ask the metadata model for "all sensors which measure air temperature and are downstream of a particular VAV" rather than "all discharge air temperature sensors downstream of a particular VAV".

These design properties of cyberphysical ontologies such as Brick ensure consistency and compatibility in modeling choices, and provide multiple ways of finding the same information. Property paths in SPARQL queries help address other variances in how the metadata model is expressed. This is best illustrated with an example. Figure 5.2 depicts three different Brick models describing the same basic HVAC system consisting of an AHU, VAV containing a damper and an HVAC zone containing two rooms. Model (a) depicts a damper as explicitly in the flow of air from the VAV to the zone. Model (b) models the damper as part of the VAV, but does not place it in the air flow sequence. Model (c) omits the VAV and damper entirely.

The query in Figure 5.3 successfully finds the AHU and room entities in each of the graphs in Figure 5.2. The query uses the `brick:feeds` relationship to match an "upstream/downstream" relationship between the AHU and the rooms to which it supplies air. To account for different levels of detail with relation to the equipment that are modeled in that air supply, the query uses a transitive closure on the `brick:feeds` relationship to match any arbitrary sequence of those edges (line 3).

5.2 Programming Models for Self-Adapting Software

This section describes and reviews two proposed programming models for self-adapting software, termed *staged* and *interactive*. The two approaches both use SPARQL queries to access the metadata model describing a particular environment, but differ in how the SPARQL queries are expressed, when they are executed, and how the results are incorporated into the execution and adaptation of software. The programming models are discussed in the context

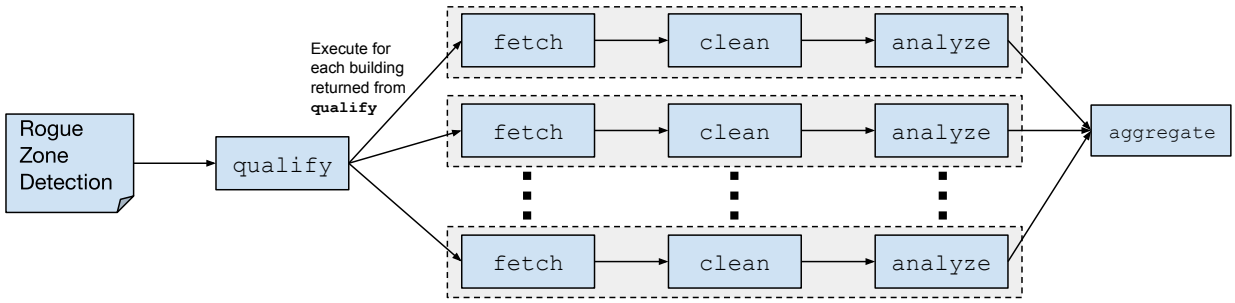


Figure 5.4: Architecture of an application written in the staged programming model

of analytics for buildings, driven by metadata expressed using the Brick ontology; however, the techniques and methods do generalize to other cyberphysical settings and ontologies.

Staged Programming Model for Large-Scale Analysis

Many implementations of cyberphysical analytics are not portable due to hard-coded point names — a characteristic that is addressed with expressive metadata models — but also tightly-coupled phases of operation which make assumptions about the nature and availability of data. The staged programming model is so-named because it decomposes the structure of a self-adapting application into discrete phases of operation which can individually enforce key assumptions for other parts of the application. Earlier phases can handle unit conversion, perform data cleaning, patch missing data and ensure that required data exists. Decoupling data preparation from the analysis simplifies the implementation of the latter. The modular application structure also has the benefit of providing re-usable and re-mixable components which can reduce development effort and assist in reproducibility.

The staged programming model decomposes analytics applications into five phases: `qualify`, `fetch`, `clean`, `analyze` and `aggregate`. These stages are executed by an application runtime, a library abstracting away access to the metadata and data stores. The structure of the backing analytics platform is described in Chapter 7. Figure 5.4 depicts the flow of execution of a generic self-adapting application expressed in the staged programming model.

The explanations of each of the stages below will follow a running example of a "rogue zone detection application". This application examines pairs of air flow sensors and setpoints for each HVAC zone in a building and identifies zones where the measured air flow is significantly above or below the setpoint for an extended range of time.

`Qualify` Stage

The `qualify` component of a self-adapting application defines the metadata and data requirements of an application in a declarative manner. The evaluation of the `qualify` component

```
# identifies pairs of sensor/setpoint  to be compared
SELECT ?sensor ?setpoint ?equip WHERE {
    ?setpoint   a   brick:Air_Flow_Setpoint .
    ?sensor     a   brick:Air_Flow_Sensor .
    ?sensor     brick:isPointOf ?equip .
    ?setpoint   brick:isPointOf ?equip
}

# (optional) identifies which zones might be effected
SELECT ?equip ?zone ?room WHERE {
    ?equip      a   brick:Terminal_Unit .
    ?zone       a   brick:HVAC_Zone .
    ?equip      brick:feeds+    ?zone .
    ?zone       brick:hasPart   ?room
}
```

Figure 5.5: Two SPARQL queries used in the `qualify` step of a rogue zone detection application.

gates the further execution of the application; if the specified conditions are not met, then the application terminates its execution. One valuable property of the `qualify` component is it can be used to filter down a large set of environments to just the set on which an application can execute; this is instrumental to the implementation of the Mortar building analytics platform described in Chapter 7. The set of environments that fulfill the `qualify` stage's predicates are the *execution set* of the application.

The `qualify` stage of an application consists of a set of SPARQL queries which are evaluated against a collection of environments. The output of this stage is a table of how many results each query returned for each environment. The application uses this information to make a decision about whether or not to qualify each environment. A simple predicate would be to qualify an environment only if all queries returned at least 1 result. More complex predicates could inform the creation of a decision tree which uses the successful execution of some queries to inform the choice of which other queries should return results in order to qualify the environment.

Figure 5.5 contains two SPARQL queries which would be used in the `qualify` step of the running example of a rogue zone detection application. The first query guarantees the existence of the sensor and setpoint data streams which are required by the application. The `?equip` variable allows the correct data streams to be compared with one another. The second query, which could have been incorporated into the first, is listed separately because it is *optional*. The results of the second query provide valuable context for which zones and rooms would be affected by the fault condition and thus considered one of the "rogue zones". However, this useful context is not strictly necessary for the application to run, so making this metadata a supplementary part of the application ensures that the application is more permissive in the set of environments it can execute on.

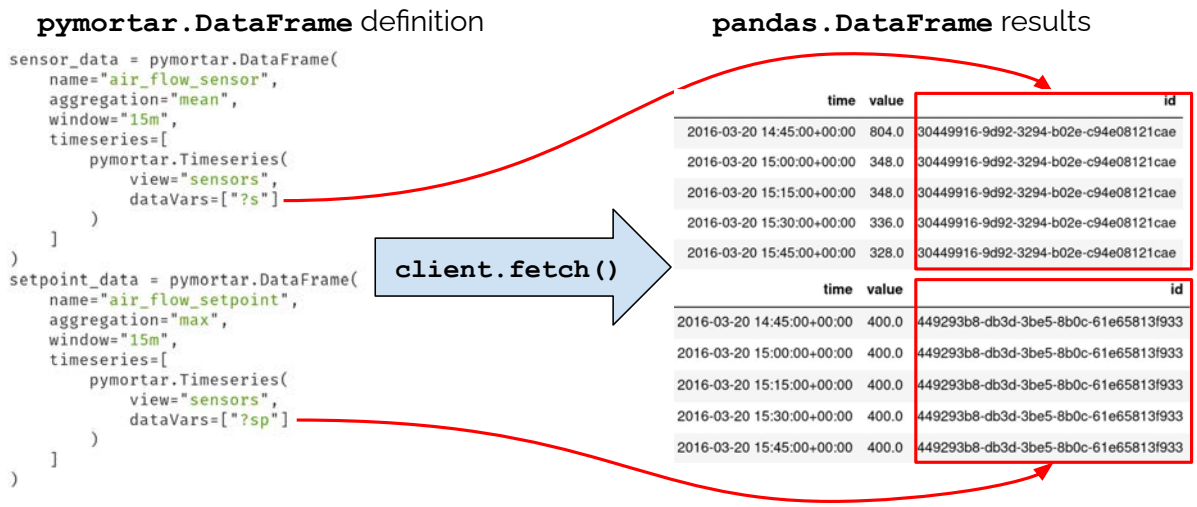


Figure 5.6: Dataframe definition in Python (left) and the resulting dataframes (right)

`Fetch` Stage

The `fetch` component of a self-adapting application describes and retrieves the data required for the application to run. The component is expressed as a declarative specification of a dataset. Specifically, this consists of:

- **Views**: A *view* contains metadata requested by the application. It is defined as a named SPARQL query, the evaluation of which on a given environment produces a table of the results. A `fetch` component can contain many view definitions.
- **Dataframes**: a *dataframe* is a time-indexed table of data requested by the application. The data stream included in a dataframe are determined by the variable bindings from one or more *views*; the author of the dataframe definition chooses which variables correspond to interesting data. The dataframe definition also specifies how the data for each variable is to be aggregated, including the method and window of aggregation. A `fetch` component can contain many dataframe definitions.
- **Time range**: this defines the temporal range of data to be retrieved. A `fetch` component contains exactly one time range.

An application contains a single `fetch` component that is executed on each of the environments in the execution set. Executing the `fetch` component means evaluating the view and dataframe definitions on the metadata model and associated databases for each environment. The output of this evaluation is an object providing access to the now-populated views and dataframes described by the application. The representation of views and dataframes depends on the host language of the programming environment. In the Mortar platform [52,

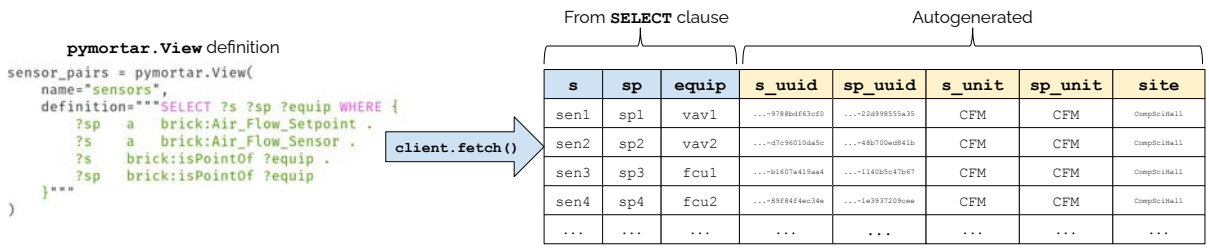


Figure 5.7: View definition in Python (left) and resulting table (right)

53], views are represented as in-memory SQL tables; this facilitates additional grouping, filtering and other operations on the metadata. Dataframes are represented as in-memory Pandas dataframes, which facilitates manipulation, filtering and computation.

Figure 5.7 contains the definition of a view for the rogue zone detection application. The table containing the results of the evaluated view receives several auto-generated columns which contain metadata about the data associated with the point entities that are returned in the query. Each row of the table corresponds to one set of bindings to the variables in the `SELECT` clause.

Figure 5.6 contains the definition of two dataframes for the rogue zone detection application: one containing the data for the air flow sensors (aggregated using a 15-minute mean) and another containing the data for the air flow setpoints (aggregated using a 15-minute max). Data semantics inform the choice to use two dataframes: retrieving the *min* or *max* setpoint in a given window ensures that the data reflects real setpoint values; however for the sensor it is more important to use an aggregation that captures characteristic rather than extreme readings. Evaluating the two dataframe definitions produces two objects that contain the timeseries for each of the entities bound to the respective variables for that dataframe; for example, the `id` column of the "air_flow_sensor" dataframe will contain identifiers for each of the air flow sensors found in the view.

`Clean` Stage

The `clean` component of a self-adapting application executes on the output of the `fetch` component. This component of the application fulfills a vital role in ensuring application portability that is not addressed by the ontology structure and flexible SPARQL queries: data normalization. This includes, but is not limited to, aligning data to a common timeframe (such as on the hour) and filling in data holes by extrapolating historical data. One common operation is dropping ranges of data which contain invalid or missing values so that each set of data streams that will be computed on together has a shared set of timestamps. By making this data cleaning and normalization step a distinct stage of execution, it is possible for other application authors to borrow or repurpose data cleaning code from existing applications. This provides an explicit mechanism for reusing analytics and data cleaning techniques.

```
# 'data' contains a result object from the 'fetch' component
# this pd.Series is True when both dataframes have data for a given timestamp
not_null = (data['air_flow_sensor'].value.notna()) & (data['air_flow_setpoint'].value.notna())
data['air_flow_sensor'] = data['air_flow_sensor'][not_null]
data['air_flow_setpoint'] = data['air_flow_setpoint'][not_null]
```

Figure 5.8: A simple `clean` stage implementation in Python which filters out periods of time where either dataframe is missing a value

```
sensor_df = data["air_flow_sensor"]
setpoint_df = data["air_flow_setpoint"]
# get all the equipment we will run the analysis for. Equipment relates sensors and setpoints
equipment = [r[0] for r in data.query("select distinct equip from airflow_points")]
# loop through data columns
for idx, equip in enumerate(equipment):
    # for each equipment, pull the UUID for the sensor and setpoint
    res = data.query("""SELECT room, s_uuid, sp_uuid, equip
                        FROM sensors
                        LEFT JOIN rooms ON rooms.equip = sensors.equip
                        WHERE equip = $1;""", equip)
    if len(res) == 0:
        continue
    sensor_col = res[0][1]
    setpoint_col = res[0][2]
    rooms = set([row[0] for row in res])
    # create the dataframe for this pair of sensor and setpoint
    df = pd.DataFrame([sensor_df[sensor_col], setpoint_df[setpoint_col]]).T
    df.columns = ['airflow','setpoint']
    bad = (df.airflow + 10) < df.setpoint # by 10 cfm
    if len(df[bad]) == 0: continue
    # use this to group up ranges where the sensor reads below the setpoint
    df['below_setpoint_group'] = bad.astype(int).diff().ne(0).cumsum()
    groups = df[bad].groupby('below_setpoint_group')

    for group in groups:
        print(f"{equipment} for rooms {rooms} had low airflow from {grp[0]} to {grp[-1]}")
```

Figure 5.9: A `analyze` component implementation in Python for the rogue zone application which finds periods where the measured airflow is lower than the setpoint.

`Analyze` and `Aggregate` Stages

The `analyze` component contains the actual application logic. It operates on the normalized dataset produced by the `clean` component. Like `fetch` and `clean`, this component is run individually for each environment in the application's execution set. This component can terminate the application, or it may output a result object; the results of each individual `analyze` component execution are sent to the `aggregate` stage which can execute logic over

the results for all environments. Cross-environment comparisons can be done by returning analysis results from the `analyze` component, or simply by passing data through `analyze` without any computation and letting `aggregate` handle computation across all environment data.

Figure 5.9 contains a sample implementation of the `analyze` component for the rogue zone application. The implementation joins views (lines 8-11) to identify which rooms are part of possible "rogue zones". The output of the component is a textual report of when and for how long terminal units and rooms had low air flow.

The staged programming model enables self-adapting software by providing an application's execution with enough information to decide which code paths to run (the `qualify` and `fetch` steps) and enabling data normalization (the `clean` step) to simplify core application logic. By modularizing the implementation, the staged programming model also makes it easier for new developers to "fork" or build off of existing stages that already accomplish the desired tasks. For example, the `qualify`, `fetch` and `clean` stages for the running example above could be repurposed for a different algorithmic approach for detecting rogue zones (offering an opportunity for rich comparisons over many different kinds of environments), or could be used in a completely different application such as one that detects the presence of nighttime air flow setbacks.

Interactive Programming Model for Exploratory Analysis

Software written using the staged programming model reacts to the details of its environment at the beginning of its execution. This is effective when the application task is well-defined and certain assumptions can be made about the availability of metadata, but these are not always known. Furthermore, software written using the staged programming model can be complex due to the need to join and filter the metadata and data available, even during the analysis itself (such as Figure 5.9). Addressing these issues requires the development of a different programming model for self-adapting software. The description of the interactive programming model below serves to illustrate how such a model might behave and how it may be implemented. A fuller exploration, implementation and evaluation of this programming model is a subject of future work.

The interactive programming model differs from the staged programming model in two important ways:

- Interactive vs batched execution: a program written in the staged paradigm is executes in batch over a collection of environments (multiple buildings). This is possible because the application must declare all queries and computation up front before it can be executed. In contrast, the interactive paradigm does not need to pre-declare any requirements for execution, and allows a programmer to incrementally add constraints and explore metadata and data during development time.

- Implicit vs explicit queries: the staged programming model explicitly includes SPARQL queries as part of its configuration (`qualify`) and execution (`fetch`). SPARQL queries

```
base = Graph()
 # finds all AHUs in the building
ahus = base.by_class("brick:AHU")
# creates a new relation with the AHU and the heating cmd point instance
heat_cmd = ahus.join_point("brick:Heating_Valve_Command")
# creates a new relation with the AHU and the cooling cmd point instance
cool_cmd = ahus.join_point("brick:Cooling_Valve_Command")
# joins on the 'ahu' key of both relations
simultaneous = (heat_cmd > 0) & (cool_cmd > 0)
# returns data between the two timestamps
print(simultaneous.between("2021-01-01", "2021-01-30"))
```

Figure 5.10: A simple program written using the interactive model that finds regions of time when AHUs are both heating and cooling.

```
SELECT ?ahu ?heat ?cool WHERE {
    ?ahu    a   brick:AHU .
    ?ahu    brick:hasPart*/brick:hasPoint   ?heat .
    ?ahu    brick:hasPart*/brick:hasPoint   ?cool .
    ?heat   a   brick:Heating_Valve_Command .
    ?cool   a   brick:Cooling_Valve_Command .
}
```

Figure 5.11: The SPARQL query implied by the execution of the program in Figure 5.10

which account for many variations in the graph structure can be complex, difficult to write and even harder to reason about. The interactive paradigm explores the use of application code to incrementally build up data structures which could have been populated by SPARQL queries but are not.

A key concept in the interactive programming model is the *relation*, an unordered multiset of tuples. A program creates relations through evaluating queries against the metadata graph and by transforming or combining other relations. The metadata graph itself is a special kind of relation that contains all tuples in the model. The programming model supports a collection of operations on relations. Relations are immutable; all operations produce a new derived relation rather than mutating the operand.

- Filter by class: a new relation B can be produced from a relation A by retaining tuples which contain at least one value that is an instance of the provided class. This operation is ontology-aware, so more general class names can be provided than appear in the original relation.
- Augment by relationship: a new relation B can be produced from a relation A by

joining with all entities that are related to a tuple in A through the indicated relationship(s).

- Fetch data by class: a new relation B can be produced that contains the timeseries data relating to entities contained in tuples of A. Data retrieval is potentially expensive and can be postponed until downstream operations materialize which data streams and segments of time are required

- Dataframe operations: relations support common dataframe operations, such as those provided by Pandas. The data types of the operands and arguments are used to disambiguate which elements of the tuples are used in the operation

The relations produced by these operators preserve the elements of the tuples used in their operation, so that downstream joins on those keys are possible.

Figure 5.10 contains an example program written in the proposed interactive programming model, which identifies regions of time when both the heating and cooling valve are active for each AHU equipment in a building. This program illustrates how relevant components of the graph can be extracted using simple operators, without having to know details of the model's expression. For example, lines 5 and 7 use a `join_point` operator, which implicitly executes a transitive closure to find all Point instances that may be related to the AHU key. Figure 5.11 reveals the full SPARQL query that is implied by the operation of this program.

5.3 Evaluation of Staged Programming Model

Evaluation of a self-adapting programming model should incorporate a measure how self-adapting the software is, and how much extra logic and code is required to express that self-adaptivity. To this end, the following evaluation focuses on two concrete metrics:

- Conciseness: how many lines of code does it take to express the self-adapting aspects of an application?

- Portability: how many different environments can the application operate in?

The evaluation is performed in the context of representative building applications which use the Brick ontology as the metadata model of each environment (building). The set of applications is informed by the literature and by knowledge of common analyses performed on buildings. These applications are implemented using the staged programming model on the Mortar [52] platform and dataset, which are described in detail in Chapter 7.

Portable Application Suite

Descriptions follow of the 12 building analytics applications implemented in the staged programming model. While the application suite provides a diverse set of functionality, there

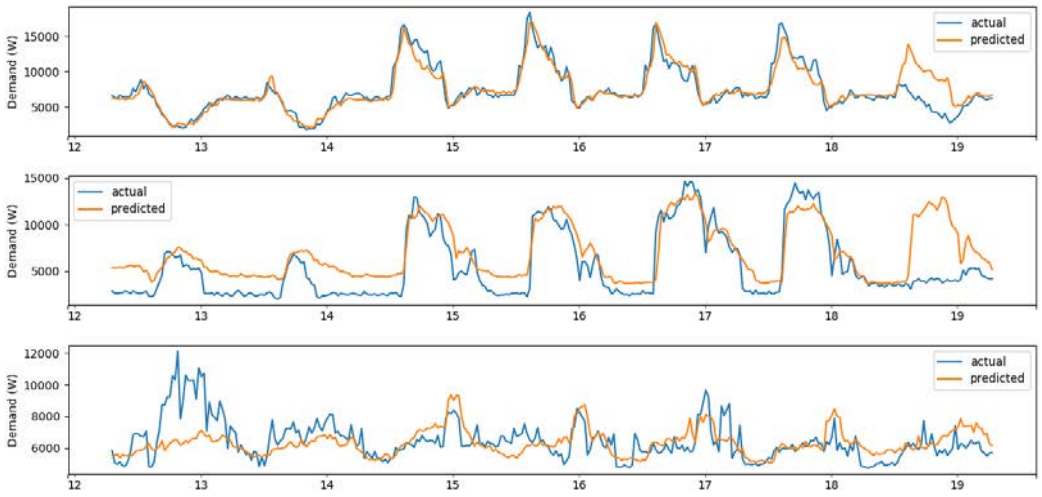


Figure 5.12: **Energy Baseline Calculation and Baseline Deviation Applications**: Predicted baseline using [92] plotted against actual building energy consumption.

is some overlap in the implementations: some applications were able to re-use `fetch` and `clean` stages from other applications.

Energy Baseline Calculation

This application is a re-implementation of an existing open-source package (LBNL-baseline[1]) which implements a baseline calculation algorithm [92]. The application requires access to data representing the energy consumption of the building, and can optionally make use of outside temperature data when available. The portable implementation has value beyond the original package because it lowers the effort required to execute the application on new sites; in conjunction with the data set described in Chapter 7, the single implementation can compare the accuracy of the predicted baseline in a variety of building environments including different climates, constructions and building uses. Figure 5.12 contains several plots of the predicted vs actual measured baseline on three different sites: a public works yard (top), senior center (middle) and fire station (bottom).

Baseline Deviation

The baseline deviation application compares measured energy consumption with a predicted baseline in order to identify periods of abnormally high or low consumption. This analysis is a simple fault detection that can act as a signal to conduct a more detailed investigation. The implementation borrows most stages from the energy baselining calculation above; only the `analyze` component was altered to compare the computed baseline with historical data.

[1] `https://github.com/LBNL-ETA/loadshape`

Energy Usage Intensity

Energy Usage Intensity (EUI) is a metric of building performance which is calculated by dividing the yearly energy consumption of a building by the total floor area of that building. Buildings with a low EUI are considered to have better energy performance. The EUI calculation application uses Brick queries to discover what kinds of energy consumption data are available about the building. Some buildings may have a single power meter for the whole building, which then needs to be integrated to compute consumed energy over time, but others may have multiple submeters that must be added together, or may expose pre-aggregated measurements (e.g. peak hourly power consumption) that require special treatment. The floor area of a building may need to be similarly computed from available data if it is not directly included in the Brick model.

The application logic retrieves a year of meter data, which can optionally be the year prior to the time of the application's execution or the previous full calendar year. Depending on the nature of the data — indicated by the Brick class of the Point entities retrieved and any attached Entity Properties — the application calculates the total energy consumption and total floor area in order to compute the EUI.

HVAC Energy Disaggregation

This application estimates the energy consumption of HVAC equipment in a building by correlating changes in electrical demand with changes in equipment state. Without the use of a metadata model, this application would be time-consuming to write for many different sites because it would involve enumerating all HVAC equipment, finding which BMS points relate to changes in their operating state, and determining which of the available electrical meters measures the demand of the equipment in question. Using the staged programming model, the application is able to discover all available data that relate to the operating state of equipment by querying the Brick model for all instances of the `brick:Status` class that are associated with (`brick:isPointOf`) some HVAC equipment (`brick:HVAC_Equipment`).

Using the above information, the application operates by measuring the largest change in energy or power consumption (depending on which is available) in a small window of time surrounding when major state changes occur. Examples of major state changes include a thermostat calling for heating or cooling and a fan, pump or compressor turning on or off. With enough historical data, this simple approach is able to estimate the per-state demand for a given piece of equipment. This is helpful for implementing models used for predicting the energy consumption of a building or subsystem based on projected equipment behavior.

Thermal Model Identification

This application trains a zone-level thermal model for predicting temperature based on zone- and room-level temperature sensors, outside air temperature, cloud coverage, HVAC equipment state data and other available data. The model does not have a minimum or mandated set of data streams that it requires to operate; rather, it attempts to train a

data-driven model on whatever thermal-related data it is able to discover by querying the Brick model. An advanced version of the application could choose specific featurization of the data based on the Brick classes of the point entities it discovers. While the particular model in this application is a basic linear regression, the general design pattern of training a data-driven model based on any available and semantically relevant data is a powerful one.

Rogue Zone Detection

This fault-detection application detects "rogue zones" which are thermal zones whose thermal behavior is consistently outside the intended control target. Because the calculation is slightly different for temperature-based and airflow-based rogue zones, there are two separate implementations for the detection of these faults; however, the `qualify`, `fetch` and `clean` steps are all very similar.

A temperature-based rogue zone exhibits temperature which is consistently outside of the provided setpoint temperature band. These zones can cause an increase in the energy use of AHUs, which must be actively cooling or heating a rogue zone when they may otherwise be on standby. Rogue zones can be caused by thermal loads in the space which are larger than the HVAC system was designed for, incorrect setpoints, or even broken sensors and equipment [27]. If enough zones in the building qualify as "rogue", it can indicate that the AHU needs to raise or lower the supply temperature of the air being delivered. An airflow-based rogue zone is one whose measured airflow is consistently below the airflow setpoint. Both versions of the application query the Brick model to find pairs of sensors and setpoints that relate to each zone, but can optionally pull in additional metadata (such as the names of the zones and rooms affected, or which AHU setpoints to change) that can provide more helpful output to the user of the application.

Simultaneous Heating/Cooling AHUs

Simultaneous heating and cooling is one of the most common kinds of faults in air handling units [128, 41]). Recall that AHUs have a heating coil valve and a cooling coil valve that condition the air distributed throughout a building. The AHU opens and closes the valves in order to adjust the degree of heating or cooling. Having both valves open at the same time is usually a waste of energy.

This fault-detection application detects simultaneous heating and cooling in AHUs using the rule-based fault diagnosis algorithms described in [128, 41]. Only a single Brick query is required to fetch the points corresponding to the valve positions for all AHUs in a building. The analysis itself consists of looking for periods of time during which both valve positions are greater than 0 and returning a report to the user. Figure 5.13 demonstrates the output of the application for one AHU during a week in the summer of 2017.

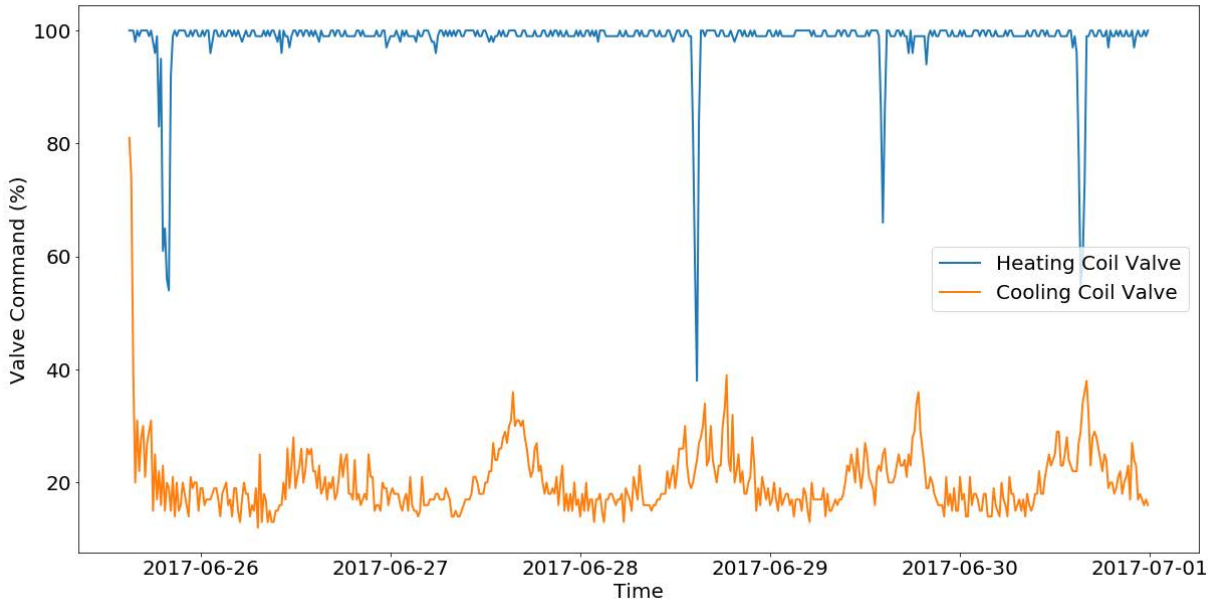


Figure 5.13: Heating coil and cooling coil valve commands over time for an AHU in a building demonstrating simultaneous heating and cooling.

Stuck Damper Detection

This is a simple and common fault-detection application which identifies dampers whose position has not changed in weeks or months. The implementation automates the task of finding damper position status (or sensor) streams and identifies the HVAC zones and rooms affected by possibly broken dampers. Similar to other applications above, the implementation can query the Brick model for other auxiliary information which can augment the capabilities of the application. The stuck damper application searches for zones with discharge airflow sensors to identify cases where the damper appears to work correctly, but the amount of air supplied to the zone does not change with its position. This can indicate a faulty airflow sensor or a broken linkage between the actuator and the physical damper.

Obscured Lighting Detection

This application implements a basic fault detection procedure for lighting systems in buildings. It queries a Brick model for available data about lighting status (both on/off state and brightness) and measured luminance for each lighting zone in a building. It uses the available data to construct a model which correlates the digital state of the lighting system and the measured luminance over time. Deviations from this model can be used to identify broken or obscured fixtures and luminaires. By leveraging the Brick class organization,

Category	Application	Brick LOC	App LOC	# sites	% coverage
Measurement, Verification & Baselining	Baseline Calculation	3	120	33	37%
	EUI Calculation	10	100	7	8%
	HVAC Energy Disaggregation	14	124	14	16%
	Thermal Model Identification	17	339	17	19%
Fault Detection & Diagnosis	Rogue Zone Temperature	15	104	56	62%
	Rogue Zone Airflow	7	98	3	3%
	Baseline Deviation	3	204	14	16%
	Stuck Damper Detection	8	91	30	33%
	Simultaneous Heat/Cool	5	125	54	60%
	Obscured Lighting Detection	11	100	2	2%
Advanced Sensing	Virtual Coil Meter	14	150	60	67%
	Chilled Water Loop Virtual Meter	17	160	15	17%

Table 5.1: Applications: Brick LOC and App LOC indicate the lines of code needed to define the Brick queries and application logic, respectively. "% coverage" is what proportion of the testbed's buildings qualified for that application; the corresponding number of buildings is in the "# sites" column.

the application can remain agnostic to the exact make and model of the lighting systems deployed in a building.

Virtual Coil Meter

This application implements the algorithm described in [124], which describes how the amount of heat energy used by a heating or cooling coil can be estimated by performing a calculation over upstream and downstream air temperature sensors, air flow sensors and the position of the valve in the coil. The implementation bootstraps itself by using Brick queries to discover the relevant families of points for each heating and cooling coil in a building. The result is a "virtual meter" which can be used as a data input to fault detection applications or predictive models.

Chilled Water Loop Virtual Meter

This application leverages the `brick:feeds` relationship to identify all equipment on a chilled water loop and sums the electrical consumption of any component found on that loop. Assembling this collection of equipment and related building points without a Brick model involves a high degree of manual effort that does not carry over when porting the application to other buildings.

Application Portability Evaluation

To evaluate the efficacy of the staged programming model, implementations of the above applications were executed on a testbed of 90 buildings. Each application is implemented in

Python. The testbed, which is described in Chapter 7, contains a Brick model and between 6 months and 6 years of telemetry for each building.

Recall that there are two dimensions to the evaluation of the programming model: conciseness and portability. Measuring conciseness consists of counting the number of lines of code to implement each application; this number is split into configuration (the number of lines of code to express the SPARQL queries in the `qualify` step) and execution (the rest of the application).

Measuring portability consists of counting the number of buildings in the testbed on which each application successfully ran without any changes; this proportion is labeled as *coverage*. Due to differences in the availability of metadata and the subsystems present in each building, not every application can achieve 100% coverage.

The result of the evaluation are summarized in Table 5.1. Applications with simpler metadata requirements (such as Rogue Zone Temperature and Basline Calculation) tend to have higher coverage; this is because they rely on entities that are more likely to be included in a Brick model. Likewise, applications which are very flexible in the kinds of metadata that they can leverage — such as Virtual Coil Meter — can run on many buildings, but will likely perform better (more accurate and robust predictions, for example) on buildings which have more metadata available.

On the other hand, applications like Rogue Zone Airflow and Obscured Lighting Detection have very low coverage. This is because Rogue Zone Airflow requires setpoint information, which is often not available in buildings, and Obscured Lighting Detection requires lighting metadata. Because lighting systems are often separately installed from the BMS, which usually manages HVAC systems in the kinds of buildings contained in the testbed, it can be difficult to include metadata about the lighting system in the Brick model.

Future Evaluations

The ideal self-adapting programming model is one which minimizes the overhead of developing a self-adapting application. "Overhead" incorporates both the *development time* of the application — a self-adapting application should ideally not take dramatically longer to develop than a non-self-adapting version — as well as the *cognitive load* of the programming model. The semantic web technology underlying the metadata model, while powerful, is mostly unfamiliar to most users and developers. An effective self-adapting programming model should, where possible, reduce the need for users to be experts in first order logic, semantic web technologies, and other "implementation details" of how the metadata is expressed. Studying the usability and efficacy of self-adapting programming models will involve developing user studies which consult not just programmers but also building and facilities managers.

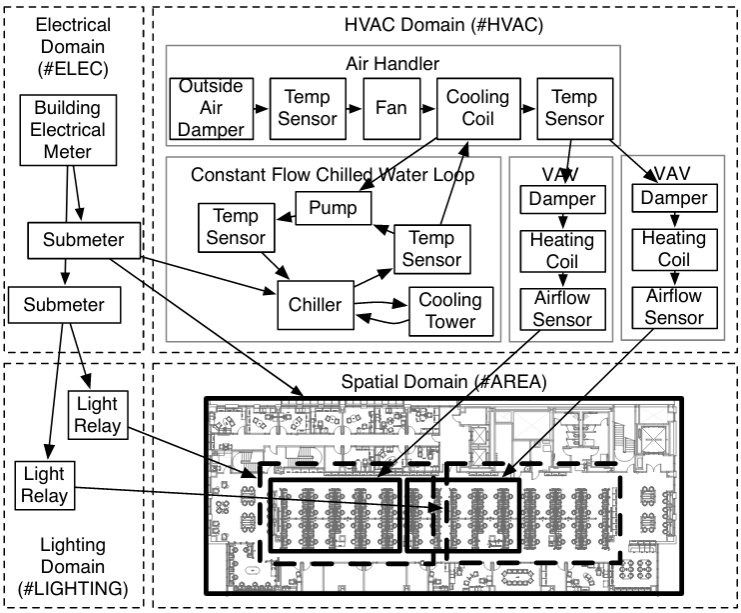


Figure 5.14: A BAS [84] representation of a building's electrical, lighting, spatial and HVAC subsystems

5.4 Alternative Programming Models

Other models for programming portable applications are both present in the sensor networking literature and have emerged in the years since Brick was first published in 2016.

Macroprogramming [61, 106] is a family of programming models enabling high-level programs to be distributed and executed over networks of potentially heterogeneous sensor nodes. Most macroprogramming models incorporate a limited number of system primitives which are implementable on a variety of sensor platforms. These system primitives can be composed together into novel programs. The intellectual overlap between macroprogramming and self-adapting software is in the use of declarative expressions to specify what data and computation is required. A platform then carries out the fulfillment of this specification; TinyDB [89] is a prime example of this approach. One significant difference between macroprogramming and self-adapting software is that macroprogramming enforces a separation between the specification and implementation of the program. Users of a macroprogramming system can remain agnostic to how the system primitives are implemented and executed; in contrast, the self-adapting programs above still require the user to write procedural code that deals with the differences between deployment environments.

Earlier models for portable software for buildings was developed in Building Application Stack (BAS) [84], Building Operating System Services (BOSS) [39] and BuildingDepot [147]. BAS established a novel portable programming model for analytics and control applications in buildings. BOSS established a suite of supporting services providing OS-like abstractions

for executing multiple applications in a building. BAS and BOSS both leverage a non-formal graph-based representation of mechanical and logical building entities (Figure 5.14). Rather than storing individual data sources in the graph like Brick, the nodes in the BAS and BOSS graphs are instantiatable objects that expose system-agnostic APIs for reading data and writing control commands. Applications discover resources by executing queries against the graph; the only recognized relationship is analogous to `brick:feeds`. Entities are organized into several broad categories such as lighting, VAV and AHUs. The concept of using expressive graph queries to find resources is a powerful one that ultimately helped to inspire the Brick work. The lack of a formal model in the BAS graph representation makes it difficult to represent different kinds of equipment and subsystems; however, the simplified application programming interface may be an effective substitute for the models explored above.

BuildingDepot proposes a template-based approach to portable programming. In this model, the metadata representation of the building is manually adapted to the application, instead of the other way around. An application defines a "template" of the data sources, relationships and entities required for its operation. The programmer specifies which elements of the building correspond to which elements of the template. In this way, an application can still be written once and executed on many buildings, but the burden of porting the application is shifted to the building modeler rather than the application developer.

Ontology-based data access (OBDA) [30] is a related line of work that employs formal ontologies to describe the semantics of data stored in other (usually relational) databases. The goal of OBDA is to faciilitate data access in terms of the semantics of the domain being modeled, rather than the schema of how the data is represented. Importantly, a single domain ontology can be mapped onto multiple schemas [156], allowing queries against the ontology to be *portable* to many different database instances.

EnergonQL [69] is a recent exploration into an alternative query language for portable building appliations. It defines a simple SQL-like language that is automatically translated into queries against the Brick ontology at query time. The intent of the design is to allow application queries to be expressed independent of the actual graph queries, which may be complex. However, the approach embeds several assumptions about the structure of a Brick model that limit the portability of queries written in EnergonQL: for example, EnergonQL appears to only support certain common classes of equipment and uses hardcoded associations of certain types of points to certain kinds of equipment in order to generate queries.

Conclusion

This chapter explored the key ideas, features and benefits of two programming models for self-adapting software. The staged programming model implements a "configure then execute" approach. Queries against a graph-based metadata model such as Brick allow the application to make decisions about which code paths and configuration decisions to make. The alternative interactive programming model presents an object-oriented approach to building

self-adapting software that implicitly performs queries against the Brick model during execution. This model, unlike the staged model, does not require users to have an understanding of the structure and contents of the metadata graph before the program is executed.

The staged programming model was evaluated by implementing 12 analytics applications that were executed on a set of 90 buildings. 9 of the 12 applications were able to run on at least 15% of the buildings without any changes to application code or configuration. This demonstrates that self-adapting software brings real benefits and can be practically implemented.

Chapter 6

Metadata Management for Self-Adapting Software

> Another flaw in the human character is that everybody wants to build, and nobody wants to do maintenance.
>
> *Kurt Vonnegut Jr.*

A critical challenge in realizing the vision of self-adapting software is how to maintain the semantic metadata model over time. Cyberphysical software relies on a correct and up-to-date representation of the environment. However, cyberphysical environments are constantly in a state of churn: physical equipment breaks or gets replaced, subsystems get updated and physical spaces get remodeled. Effective management of semantic metadata requires detecting changes in the cyberphysical environment and incorporating those changes into the metadata model which is available to self-adapting applications. This chapter focuses on methods and systems for producing and maintaining the semantic metadata model for an environment over time.

Two aspects of the metadata model will change over time: its *content* and its *semantics*. The content of the metadata model changes in response to the evolution of the environment it models with respect to the scope of the model. For a metadata representation like Brick, salient environmental changes include repairs, retrofits and remodels that affect the representation of equipment, controllers, networks and physical architecture.

The semantics of the metadata model are driven by evolution of the domain — for example, due to technological advancements — or evolution of the application — for example, if the ontology developers want to support new kinds of applications. While certain development practices such as semantic versioning[1] can foster "backwards compatibility" between

[1] `https://semver.org/`

versions of an ontology, these simply place subjective constraints on how the metadata representation can be changed and do not provide a mechanism for using older or newer versions. Since its release in 2016, Brick has seen two major releases that changed the definition and names of classes and reorganized the hierarchy [16, 51].

This chapter examines why and how changes in metadata models and representations occur over time in cyberphysical settings. It then formalizes and develops a technique for handling changes in model content. This technique is implemented in a system, *Shepherding*, which handles changes in metadata model content over time and is demonstrated in the context of Brick models for evolving buildings. The chapter concludes with a brief discussion of how evolution in semantics may be handled.

6.1 Prior Work on Metadata Management

Managing changes in metadata content and semantics incorporates components of data integration and schema evolution.

Data Integration and Record Linkage

Data integration is the process of consolidating heterogeneous data from many sources into a single, "application-facing" representation. Architectures and techniques for data integration differ in the coupling and mapping between data sources and the destination schema. Recent work in data integration has favored data warehouses. Here, heterogeneous data sources are queried indirectly through a "virtual database" which translates queries over a central schema to the schemas of individual data sources. The wrapper-mediator architecture is one example of this approach [56, 152]. Queries are executed against a *mediator* which has knowledge of many data sources which may contain data relevant to the query. The mediator forwards queries to *wrappers* — logical processes which interface directly with the data sources and rewrite the query to execute on the underlying schema.

However, in cyberphysical environments, different representations are often owned by different stakeholders. This means that they are not consolidated into a single storage repository nor are they readily available for consumption by automated tooling. In many cases, the source data have an unstructured form that does not facilitate querying — recall the heterogeneous and unstructured labels detailed in Chapter 2. Extract-transform-load (ETL) pipelines are one mechanism for translating existing data sources into a more desirable form [144]. ETL pipelines *extract* data from existing data sources which may be local or remote, *transform* the data into the necessary structure and *load* the processed data into a unified database.

There are several classic data integration issues which are relevant to managing cyberphysical metadata. First, the semantics of different metadata representations must be handled. Different data sources are built with differing perspectives on a domain. Integration of data sources must take into account the different meanings and organizations of the data

they contain. Often, the data semantics are implicit in the data schema and are not captured in any formal manner. Systems like Mastro [30] and [87] describe source data using ontologies in order to inform the data integration process with the semantics of the data being integrated. Establishing the mapping between many metadata sources and the destination model can be time-consuming. Systems like [38] and [95] reduce the effort with a "pay-as-you-go" approach to schema mapping that focuses on only the data required by particular applications.

Another issue is that of data representation. Different data sources may represent the same items using different syntax, data structures and schemas. In particular, many sources of cyberphysical metadata are characterized by a *lack* of well-defined semantics. Lines of work on schema mapping [48] and schema matching [22] deal with the mechanics of transforming data in one form to another. This includes aggregations, disaggregations, relabelings of data and reorganizations of the schemas. An effective solution for integration of cyberphysical metadata should handle both structured and unstructured sources of metadata.

The final major issue is how to determine when two or more data sources refer to the same object or entity under different names or with different descriptions. This is classically referred to as *record linkage* or entity disambiguation [153]. Record linkage may be accomplished through a variety of classic techniques such as string similarity or matching entities in different data sources by shared attributes. Newer techniques explore leveraging large amounts of complex and related data [45] or reasoning through noisy or uncertain data [46].

Schema Evolution

The body of work on **schema evolution** focuses on how to better manage and automate the process by rewriting queries to execute on new schemas and transforming source schemas to fit a new schema. Many frameworks like Python's Django [42] and Ruby's ActiveRecord [13] provide tools for schema evolution, but focus on the migration of data and do not provide any mechanism for database administrators (DBAs) to reason about the semantic impact of the schema migration. Systems like [35], [23] and [21] provide greater degrees of automation through high-level schema modification operators that assist DBAs in evaluating the impact of a schema modification as well as performing the data transformation itself. Other approaches take the additional step of automatically transforming applications to match changing schemas [32].

Schema evolution research focuses on the *structural* aspects of schemas: the attributes, their names and datatypes, membership in tables, and integrity constraints between them. This approach ignores the issue of *semantic* integrity: how do the meaning and context of attributes change over time and how do these changes affect the interpretation and usage of the data in a database?

[107] calls out this difference in the context of **ontology evolution**. Because an ontology is an "explicit specification of a conceptualization of a [knowledge] domain"[107], changes in the domain, conceptualization and explicit specification may all incur changes in the ontology. Existing approaches for ontology evolution [108, 136, 109] focus on managing

changes in the ontology itself and do not address how to interpret those changes when the ontology describes attributes in an external table.

6.2 Extracting Semantic Metadata

One of the most time-consuming and error-prone aspects of managing semantic metadata is creating the representation that can be accessed by software. This section focuses on how metadata models such as Brick can be bootstrapped from existing digital representations of the environment. For a family of common metadata representations — recall BuildingSync, Haystack, gbXML and Modelica/CDL from Chapter 2 — a *wrapper* is developed which is able to produce or infer Brick metadata from the underlying source. Wrappers are a component of a metadata integration architecture, which is able to combine the individual Brick models produced by each driver into a unified representation.

BuildingSync

BuildingSync is an XML-based schema for reporting energy-related properties of buildings and their subsystems and coarse-grained relationships between them. The BuildingSync wrapper produces Brick metadata from a BuildingSync document by mapping combinations of XML elements and attributes to Brick class definitions. The correspondence between BuildingSync and Brick is expressed as a mapping from an XPath expression to a Brick class. For example, the BuildingSync `auc:Chiller` element aligns with the Brick `brick:Chiller` class. If the `auc:Chiller` element contains a `auc:ChillerType` property with the value "Absorption", then the wrapper can infer the more specific Brick class of `brick:Absorption_Chiller`.

BuildingSync does not explicitly contain topological information about building systems which could inform inference of Brick relationships. However, the limited number of equipment types recognized by BuildingSync means it is feasible to infer some relationships based on the structure of the BuildingSync document and the types of the entities. A child relationship between one XML element and another usually denotes a compositional (`brick:isPartOf`) relationship between the child and parent. One example of this is the nested relationship between the `auc:Site` and `auc:Building` BuildingSync elements. Other internal attributes such as `auc:LinkedSystemID` can associate components of a subsystem with each other. At time of writing, the wrapper defines 27 direct mappings, primarily for locations and equipment types.

There are a few challenges that must be addressed by the BuildingSync wrapper. BuildingSync documents often represent *collective* properties of building systems and equipment — e.g. the number of absorption chillers, not how the individual chillers are connected — which limits the number of Brick relationships that can be derived. Also, by modeling systems rather than components, BuildingSync models often lack descriptive labels for equipment and points.

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

Figure 6.1: Original Haystack entity from the Carytown reference model

Project Haystack

The structure of a Haystack model has a straightforward mapping to Brick: each Haystack entity corresponds to one or more Brick entities. The generic links between Haystack entities (called `ref`s in Haystack parlance) can be expressed with Brick relationships. However, because the semantics of a Haystack model are not well-defined, there is no unambiguous and exhaustive mapping of Haystack metadata to Brick. As a result, the types of Haystack entities and relationships between them must be inferred. This is accomplished by using the tags associated with Brick concepts to infer the most likely Brick class for a given set of Haystack tags.

The inference engine operates on a JSON export of a Haystack model. First, the engine applies some preprocessing by filtering out tags that do not contribute to the definition of the entity, including data historian configuration (`hisEnd`, `hisSize`, `hisStart`), current readings (`curVal`) and display names (`disMacro`, `navName`). Figure 6.1 shows an example of a "cleaned" Haystack entity containing only the marker and `Ref` tags from the Carytown reference model.

Next, the engine transforms the Haystack entity into an RDF representation that can be understood by the inference engine. The engine translates each of the marker tags into their canonical Brick form: for example, Haystack's `sp` becomes `Setpoint`, `cmd` becomes `Command` and `temp` becomes `Temperature`. The engine creates a Brick entity identified by the label given by the Haystack `id` field, and associates each of the Brick tags with that entity using the `brick:hasTag` relationship. Figure 6.2 contains the output of this stage executed against the entity in Figure 6.1.

At this stage, the engine assumes a one-to-one mapping between a Haystack entity and a Brick entity. This is usually valid for equipment entities which possess the `equip` tag, but Haystack point entities (with the `point` tag) may implicitly refer to equipment that is not modeled elsewhere. Figure 6.1 is an example of a Haystack point entity that refers to an outside air damper that is not explicitly modeled in the Haystack model. The last stage of the inference engine performs the "splitting" of a Haystack entity into an equipment and point. This proceeds as follows:

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

Figure 6.2: Intermediate RDF representation of the Haystack entity; Haystack software-specific tags (e.g. `cur`, `tz`) are dropped.

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

Figure 6.3: Brick inference engine splits the entity into two components: the explicit point and the implicit outside damper equipment.

First, the inference engine attempts to classify an entity as equipment. The engine temporarily replaces all point-related tags from an entity – `Point`, `Command`, `Setpoint`, `Sensor` – with the `Equipment` tag, and finds Brick classes with the smallest tag sets that maximize the intersection with the entity's tags. This corresponds to the *most generic Brick class*. In the running example, the inference engine would transform the entity in Figure 6.2 to the tags `Damper`, `Outside` and `Equipment`. There are 12 Brick classes with the `Damper` tag, but only one class with both the `Damper` and `Outside` tags; thus, the minimal Brick class with the maximal tag intersection is `Outside Air Damper`. If the inference engine cannot find a class with a non-negligible overlap (such as the `Equipment` tag), then the entity is not equipment.

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

Figure 6.3 contains the two inferred entities output by this methodology. In the case where a Haystack entity is split into an equipment and a point, the Brick inference engine associates the two entities with the `brick:isPointOf` relationship (line 10 of Figure 6.3). Additionally, the inference engine translates Haystack's `Ref` tags into Brick relationships using the simple lookup-table based methodology established in [15]. The inference engine applies these stages to each entity in a Haystack model; the union of the produced entities

and relationships constitutes the inferred Brick model.

Modelica/CDL

Modelica and CDL models consist of a set of connected *objects*. This linked structure clearly identifies entities and the relationships between them, which closely resembles the structure of a Brick model. Modelica objects have classes, which supports the development of a mapping between Modelica classes and Brick classes. For example, every instance of the Modelica class `Buildings.Fluid.Sensor.Temperature` can be translated into a `brick:Temperature_Sensor` entity.

The wrapper takes as input a JSON export of a Modelica/CDL model [98]. The wrapper treats each instance of a Modelica model in the document as a Brick entity, and assigns a Brick class to entities whose class is defined in the Modelica Buildings Library [150]. To infer relationships between these entities, the wrapper examines the ports for each Modelica instance; these are connected by `connect` statements to other instances of Modelica models. These links between objects in a Modelica model can inform the choice of sequential (`brick:feeds`) and compositional (`brick:hasPart`) relationships between their corresponding Brick entities.

There are a few challenges in producing Brick metadata from a Modelica/CDL model. First, because Modelica is a general modeling language, it is possible for models to describe buildings using classes unknown to the wrapper. The wrapper establishes mappings for many of the common classes in the Modelica Buildings Library [150], but there is no guarantee that a Modelica model will use these classes. Second, due to Modelica's model encapsulation, contextual properties, i.e., how the objects relate to a larger system, need to be inferred. For example, for an instance of the class `Buildings.Fluid.Sensor.Temperature`, where it is located (e.g. exhaust air, return air, entering water, leaving water) need to be inferred from the system that contains the sensor.

gbXML

Wrappers for BIM can produce Brick metadata about individual components, but inferring contextual relationships between those components is more difficult. Because BIM models focus on the geometry and physical connections of spaces and equipment, the representations may lack or obscure the contextual information needed during the operations and maintenance stages of the building. For example, although several versions of the IFC standard enumerate possible physical and contextual properties of fans, these details may not be included in an IFC model or must be inferred by traversing the connections between other elements in the model. Previous work also indicates that little Brick metadata can be inferred from IFC models [85]. This is due in part to the complex and generic schema of IFC in which related pieces of information are often separated by many intermediate objects [44]. In contrast, gbXML models contain more explicit contextual information such as

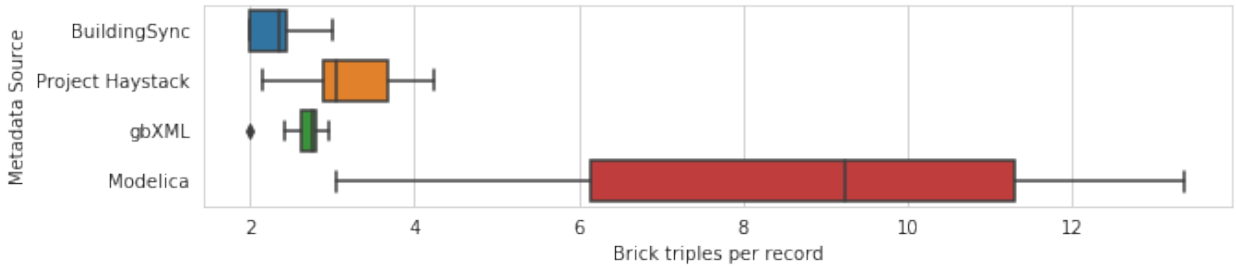


Figure 6.4: The distribution of the number of triples inferred per entity for each wrapper.

the `<AirLoop>` element, which groups related HVAC equipment together. This permits the inference of more metadata such as sequential and compositional relationships.

A significant challenge for the production of Brick metadata from a BIM model is variability in how BIM models are expressed. BIM standards are designed to be flexible and extensible, which can result in a lack of consistency and thus interoperability [49, 99]. Representing BIM models using semantic web technology will allow the wrapper to more easily validate and inspect the BIM metadata, which may result in a more complete Brick metadata model [115].

Evaluation of Wrapper Performance

To characterize the behavior and performance of the wrappers, a set of publicly available metadata models from a variety of metadata representations was assembled. The models do not cover the same set of buildings, but the population still enables empirical measurement of the availability of Brick metadata in those representations.

Figure 6.4 and Figure 6.5 illustrate the number of Brick triples inferred per entity and the total number of triples inferred by each wrapper, respectively. The reason for these distributions is due to fundamental differences between the metadata representations and Brick, and implementation details of the wrappers.

The amount of Brick metadata obtained from a BuildingSync model is limited compared to what can be inferred from gbXML, Modelica or Project Haystack models (Figure 6.5). This is due to a difference in scope: BuildingSync describes properties and performance characteristics of building systems, rather than the individual components and relationships found in other metadata sources. As a result, a BuildingSync model may be a better *export* target from a unified Brick model.

Within this population of sites, Modelica models contain the most Brick metadata per entity, but do not contain as many entities as Haystack models. The BuildingSync and gbXML wrappers only produce a few triples per entity: usually the type of the entity and a topological reference to a few other entities. Haystack models contain more entities and more Brick metadata per entity than metadata sources for energy audits and BIM. This is

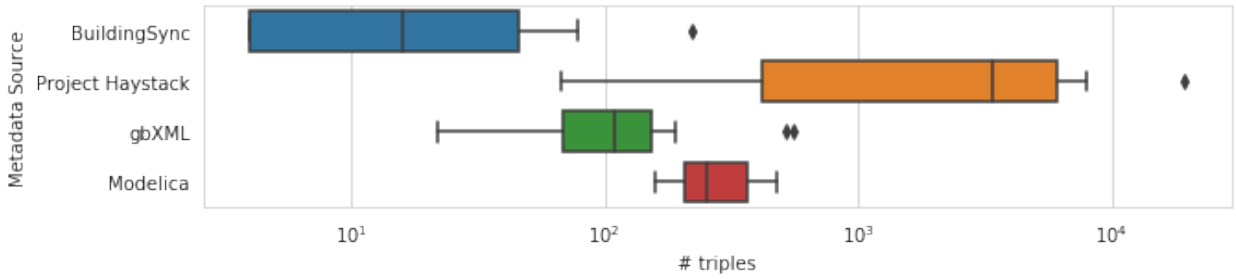


Figure 6.5: The distribution of the *total* number of triples inferred by each wrapper. Note the log-scale on the X axis

an intuitive result because the Haystack metadata representation is the most similar to Brick in terms of its intended use cases, and thus presents much of the same metadata (if in a less structured form).

6.3 Metadata Management Over the Building Lifecycle

Effective management of a metadata model which enables self-adapting applications requires keeping the metadata model "in sync" with the state of the cyberphysical environment in which applications will be deployed. This section examines the evolution of a building over time as a case study of handling changes in a cyberphysical environment while enabling self-adapting software.

In the building context, updates to the metadata model are informed by observing and deriving updates from other digital representations of the same environment. Many different digital representations of a building are typically produced over the course of its lifecycle. These representations contain — in some form — the metadata required to support different operational stages and treatments of the building, from initial planning and design, to construction and commissioning, through operations, audits, retrofits and repairs.

Figure 6.6 describes at a high level the different metadata representations that may be created and leveraged over the lifecycle of a building. The design phase of a building, conducted using BIM, may produce IFC or gbXML models that are used during the construction phase of the building. During construction, this metadata may be used in conjunction with CDL descriptions of the building's sequence of operations to configure the BMS. Unstructured BMS metadata may be captured in a Brick or Haystack model to facilitate data analysis and to perform predictive maintenance. Other metadata sources such as BuildingSync may be used to conduct energy audits before and after retrofits and repairs, which themselves may rely upon CDL, IFC or gbXML representations of the building's control loops and assets.

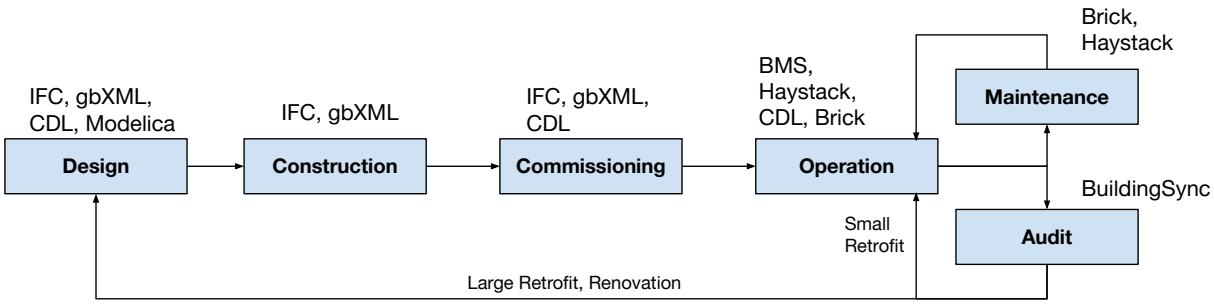


Figure 6.6: Different tools for different stages: Many different metadata standards and technologies are applied over the course of a building's lifecycle, but are relatively siloed and thus non-interoperable.

Different representations communicate different perspectives on the same building: the metadata required to support the construction of a building is different than the metadata required to manage schedules and control sequences. Therefore, it is possible to detect changes in many different aspects of the environment by observing many different metadata representations. There are several desirable properties of this approach:

- It is possible to bootstrap the creation of the metadata model from existing digital representations, lowering the initial cost of building out the model, which can be a non-trivial investment [27].
- Because the metadata model's intent is to enable self-adapting software, it does not need to enable other kinds of use cases that are traditionally served by other technologies. This preserves any existing investments and makes the proposed system easier to adopt.
- Many other digital representations are "dropped" or left unmaintained after the operational stage of the building for which they were created is over [114]. Providing a use for these representations — enabling self-adapting software — incentivizes their maintenance.

Working with several metadata representations poses challenges for effective management. Differences in the semantics, structure and syntax of models generally result in a lack of *interoperability* [158] between them. This limits the extent to which metadata from one stage of a building's lifecycle can contribute to the metadata for another stage. There have been previous attempts to increase information sharing and reuse between stakeholders though a shared knowledge base [83] or by centralizing all data in a BIM model [158]. [139] demonstrates the semi-automated configuration of a building automation system by exporting BACnet objects from the BIM, enabling the exchange of information across the design, construction and operational stages of a building. However, because representations such

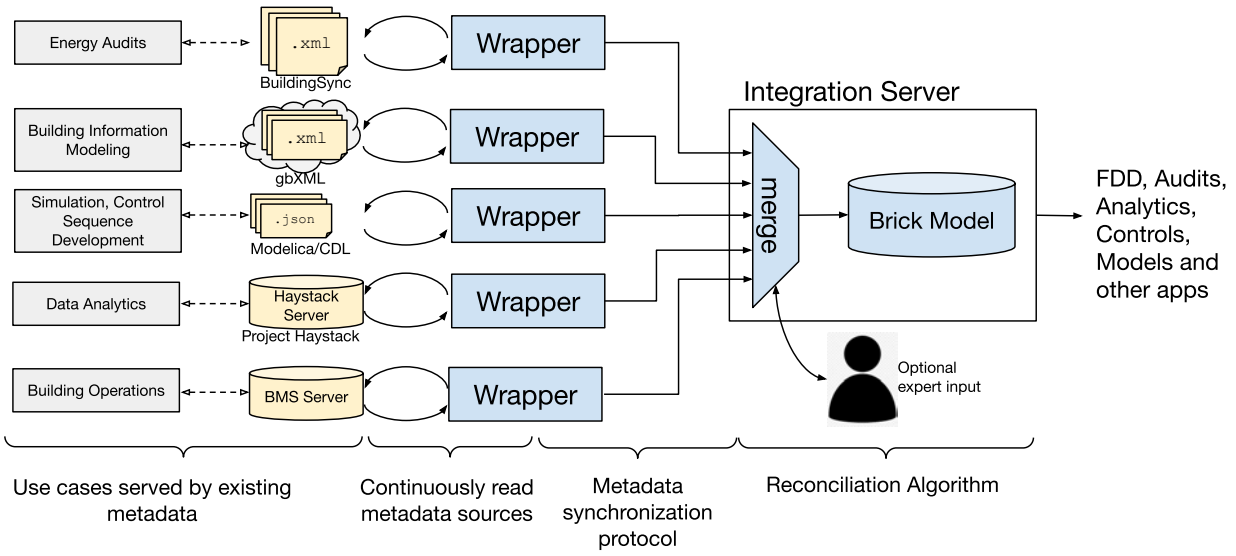


Figure 6.7: Overview of the proposed approach: *wrappers* interface directly with existing metadata sources stored in local file systems, or accessed via file shares or networked services. Wrappers continuously publish inferred Brick metadata to a central server, which produces a unified model.

as BIM are not appropriate for self-adapting software, the approach developed below deals with how to lift existing metadata representations into a Brick model.

Specifically, this section develops a protocol and algorithm for performing *continuous integration* of heterogeneous metadata representations into a single metadata model which can be used for self-adapting applications. The protocol and algorithm are incorporated into a functional system which has been evaluated on several real metadata sources for two representative buildings.

Architecture for Continuous Metadata Integration

The proposed system adapts existing data integration architectures and techniques to the problem of lifting heterogeneous metadata sources about buildings into an ontology-defined structure that serves as a canonical representation.

Figure 6.7 represents the architecture of the ETL data integration system. On the far left of the figure are existing metadata representations of the building; these may exist in a structured or unstructured form and support a variety of legacy applications. A **wrapper**, described above, is a software process that interfaces with one or more of the existing metadata representations and produces a Brick model of the metadata contained within. For structured or standardized metadata sources such as BuildingSync and gbXML, the wrapper may perform a direct translation of the source's concepts and structures to Brick. For

less structured and more ad-hoc sources such as Haystack and BMS labels, a wrapper may require a statistical or heuristic-based approach to inferring Brick metadata. It is the job of the wrapper to continually monitor the content of the underlying metadata source and to produce up-to-date Brick models in a timely manner. The architecture places no other requirements for how Brick metadata must be produced or inferred.

Wrappers report inferred Brick metadata to an integration server by means of a **synchronization protocol**. The protocol implements a simplified `git`-style source control system which ensures that the integration server has the most up-to-date Brick metadata from each metadata source. The protocol operates in a push-based manner: wrappers report updated metadata when it is available. A push-based mechanism is preferable because the Brick inference process implemented by the wrapper may require human input that is not available on-demand.

The **integration server** is a logically centralized process that assembles Brick metadata from a collection of wrappers and integrates them into a unified Brick model which is made available for applications. Because different metadata sources are created at different stages of the building and by independent stakeholders, the Brick metadata produced by the wrappers is likely to contain disagreements and inconsistencies, or may simply be out of date. To address this issue, the integration server incorporates a novel **reconciliation algorithm**—analogous to the "merge" operation in `git`—that attempts to resolve the differences between the metadata reported by the wrappers.

The architecture decouples the tasks of inferring Brick metadata from a particular source and integrating multiple sources of Brick metadata into a unified model. This establishes a common platform for Brick metadata inference research and allows existing and future methods to be used together. For example, inference methods such as [27] and [82] that operate on ad-hoc metadata representations can be adapted to the protocol.

Formalism of Brick Metadata Integration

The following definitions formalize the behavior of the wrappers, synchronization protocol and reconciliation algorithm described above.

A Brick model is a directed graph in which nodes are "entities" (physical, virtual, logical things and concepts) and edges are relationships between entities. Brick models are handily expressed in the RDF data model, which defines a graph as a set of *triples*: 3-tuples of `subject` (node), `predicate` (edge), `object` (node).

A metadata source S_i corresponds to a set of metadata models m_i^t indexed by a unique timestamp t. A wrapper W_i is a function which produces a set of entities for a particular metadata model:

$$W_i(m_i^t) \to \{e_1^t, e_2^t, \ldots, e_n^t\} \tag{6.1}$$

where e_j^t is described by a set of fields called a *record*. Each record includes a set of triples describing the entity, given by $T(e_j^t)$. Together, the triples produced by a wrapper constitute a Brick model G_i^t for a given metadata source and timestamp. The content of the Brick model

```
{"id": "RTU-1",
 "raw": {
   "content": "<auc:Delivery ID=\"RTU-1\">
    <auc:DeliveryType>
      <auc:CentralAirDistribution>
        <auc:AirDeliveryType>Central fan</auc:AirDeliveryType>
        <auc:FanBased>
          <auc:CoolingSupplyAirTemperature>73</auc:CoolingSupplyAirTemperature>
        </auc:FanBased>
      </auc:CentralAirDistribution>
    </auc:DeliveryType>",
    "encoding": "XML"},
  "source": "BuildingSyncWrapper",
  "source_version": "2.1.0",
  "timestamp": "2020-07-16T20:02:50",
  "protocol_version": "1.0.0",
  "triples": [["http://example.com/building#RTU-1",
               "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
               "https://brickschema.org/schema/Brick#Rooftop_Unit"]]}
```


Figure 6.8: Example record published by the BuildingSync wrapper, showing the original metadata (`raw`) and the inferred Brick metadata (`triples`).

representing metadata from source i at time t is given by

$$G_i^t = \bigcup_{j=1}^{n} T(e_j^t) \tag{6.2}$$

The task of the reconciliation algorithm is to combine the set of Brick models produced by each wrapper into a single unified Brick model. The input to the reconciliation algorithm is the set of entities E from the *latest* metadata model for each of m sources

$$E = \bigcup_{i=1}^{m} W_i(m_i^{t_{\max}}) \tag{6.3}$$

where $t_{\max}$ is the timestamp of the most recent model for metadata source i.

Metadata Synchronization Protocol

Wrappers update the integration server with the latest Brick metadata from a specific source via the metadata synchronization protocol. The metadata synchronization protocol decouples the method of inferring or producing Brick metadata from how that metadata becomes integrated into the authoritative model. This allows the proposed system to incorporate new metadata sources and novel methods of inferring Brick metadata.

As part of the protocol, each wrapper presents the inferred Brick metadata according to a metadata *profile*. The profile is a structured representation of the Brick metadata

produced by the wrapper at a given time and is expressed as a set of HTTP resources. The root resource (`/`) holds a list of entity ids with associated Brick metadata for the current metadata model, and a timestamp representing the version of that metadata model. The record resources (`/record/<id>`) hold the *record* associated with a given entity id. A record contains the following fields:

- `id`: a name or other identifier for the entity, as given by the metadata source
- `raw`: identifies the encoding (e.g. JSON, XML) and content of the original metadata that defined this entity. May contain additional metadata not expressed in Brick
- `source`: identifies the metadata source
- `timestamp`: denotes the time at which the metadata source was read to produce the current metadata model
- `triples`: a list of RDF triples defining the Brick metadata for the entity

The `timestamp` field constitutes the version of the metadata model and is updated only when the model or wrapper changes.

Figure 6.8 contains an example of a BuildingSync record for an HVAC delivery system named "RTU-1". Note that the original XML element contains additional metadata not conveyed in the produced Brick triples.

The content of the profile is synchronized with the integration server over the protocol. The protocol operates over HTTP and consists of two request-response actions: `check` and `sync`.

A `check` is an HTTP GET which asks the server for the latest known version of metadata from a particular source, and the number of records at that version. The server responds with the version as a timestamp and an integer representing the number of records. By comparing this information with the latest local version and corresponding number of records, the wrapper can determine if the server has a complete copy of the most recent Brick metadata from the wrapper. If the server timestamp is older than the local timestamp, or the number of server records at the latest timestamp is less than the number of local records, the wrapper performs a `sync`.

A `sync` is an HTTP POST of the list of records for the most recent version of the metadata model to the integration server, These records must contain the same version timestamp, which allows the set of records to span more than one HTTP POST while still being associated with the same version of the metadata model.

The server saves all records in a local database. When the server performs the reconciliation algorithm to produce a unified metadata model, it by default only considers the records corresponding to the most recent timestamp (version) per source. By extension, the server can also produce a unified metadata model for any point in the Brick model's history. This allows applications to access the history of changes in a building, but through the interface of

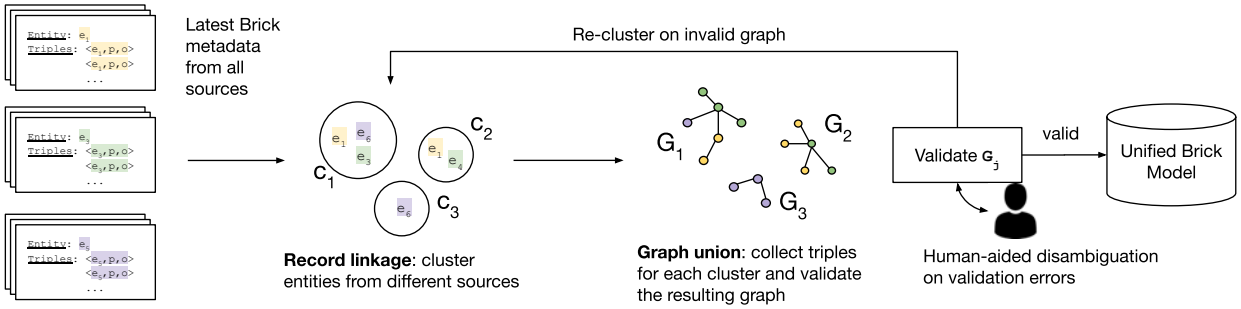


Figure 6.9: The phases of the reconciliation algorithm. The latest Brick metadata (far left) is stored by the integration server.

a standardized, unified representation rather than an ad-hoc collection of diverse metadata sources.

The `triples` field can contain arbitrary Brick metadata to be relayed to the server. Typically this involves type information (`vav1` is a `brick:VAV`), system composition information (`vav1` is downstream of `ahu1`), telemetry association (`vav1` has temperature setpoint `temp_sp1`) and location information (`tstat1` is in `Room 410`). The triples may also define extensions to the Brick ontology, such as to describe additional properties of an unusual device or point. The union of all triples in all `triples` fields for all records constitutes the Brick model for the wrapper ($G_i^{t_{\max}}$ above).

Reconciliation of Multiple Brick Models

In order to produce a unified Brick model usable by applications, the Brick models produced by each wrapper must be merged together. This requires *reconciling* the differences between each of the Brick models: entities may be named differently and have different associated types and metadata. Reconciliation must be performed continuously in order to account for changes in the underlying metadata sources arising from evolving representations and environments. This subsection presents a reconciliation algorithm that extends existing record linkage techniques to graph-based semantic metadata and enforces the logical and semantic validity of the resulting unified model.

Figure 6.9 illustrates the major phases of the reconciliation algorithm. First the most recent Brick models for each source are loaded from the integration server, having been deposited there by the wrappers. Then, the algorithm finds clusters of entities such that all entities in the cluster correspond to the same logical, virtual or physical *instance* (the "record linkage" stage in Figure 6.9). For each cluster, the algorithm produces the graph which is the union of all Brick metadata associated with the entities in that cluster and validates that the new graph is logically and semantically sound (the "graph union" stage in Figure 6.9).. Formally, each cluster of entities c_j has an associated metadata graph G_j

which is the union of all the associated triples for entities in that cluster:

$$G_j = \bigcup_{e_i \in c_j} T(e_i) \tag{6.4}$$

The clustering is successful and the algorithm terminates if the metadata graph G_j satisfies a set of constraints and checks.

The algorithmic processes for each stage are described in detail below.

First Phase: Record Linkage

The record linkage stage takes as input the most recent metadata records for each metadata source known to the integration server and outputs a set of clusters. The stage applies two record linkage techniques on the metadata records: *string matching* and *type alignment*.

String matching calculates edit distance between entity labels to produce clusters of entities. The name of an entity can be derived from string-valued properties such as `rdfs:label`, or the URI of the entity if no string-valued properties are found. The goal of this step is to use the semantic information sometimes encoded in entity labels as one heuristic for linking [27]. Due to different naming conventions between metadata sources, there can often be greater similarity scores between entities from the same source than between entities of different sources. The algorithm assumes that all entities reported by a metadata source are distinct and only clusters entities from different metadata sources.

Type alignment leverages semantic information from the proposed *types* of each entity to do type-aware clustering. The algorithm identifies all entities with a Brick class and associates with each entity all Brick classes which are equal to or are *superclasses* of its given type. If two or more sources have the same number, k, of entities of a given type, the algorithm produces k clusters containing one entity from each source with the highest pairwise similarity between their names. The clusters produced by this second step are added to the set of clusters produced by the first step.

Second Phase: Graph Union

The second phase of the algorithm takes as input the clusters of entities from the first phase and builds and validates the graphs formed by merging their associated triples. For each cluster, the algorithm produces the graph G using the formula in Equation 6.4. The algorithm also adds statements to the Brick model to merge the different identifiers for the same entity (this uses the `owl:sameAs` property).

Unlike many other metadata sources, Brick is built over formal logic. This allows continuous validation of a Brick model as metadata is added to it, which allows the algorithm to produce a logically valid model. The logical validation is implemented by a process called a *reasoner*, which also generates logical consequences of the statements in a Brick graph (the reasoning process is described in more detail in Chapter 7). The reasoner examines the graph G_j for each cluster and produces a set of logical exceptions. These exceptions

```
# BuildingSync: ahu-1
bldg:bsync-ahu-1    rdf:type   brick:Air_Handling_Unit ;
    rdfs:label "AHU-1" .
# BuildingSync: main-meter
bldg:bsync-meter    rdf:type   brick:Building_Power_Meter ;
    rdfs:label "main-meter" .

# Haystack: rtu-1
bldg:ph-rtu-1    rdf:type   brick:Rooftop_Unit ;
    rdfs:label "RTU 1" ; brick:hasPoint   bldg:oat-1 .
# Haystack: main-meter
bldg:ph-meter    rdf:type   brick:Power_Meter ; rdfs:label "Main Meter" .
```

Figure 6.10: Example Brick metadata produced by BuildingSync and Project Haystack wrappers. The `rdfs:label` property denotes the original name or identifier of the entity in the metadata source.

```
bldg:rtu-1   rdf:type    brick:Rooftop_Unit ; brick:hasPoint bldg:oat-1 .
bldg:meter   rdf:type    brick:Building_Power_Meter .
```

Figure 6.11: The inferred unified metadata model for the triples in Figure 6.10. The most specific type is chosen for each entity, and that associated properties are carried through.

indicate that either the entities in the cluster are not equivalent, or the metadata associated with those entities is incorrect. Examples of exceptions include *incompatible types* (e.g. if a cluster contains entities with disjoint types), *incompatible relationships* (e.g. if the values of an entity's properties and relationships do not match associated constraints) and *semantic "sniff tests"* which are qualities of the Brick graph that are not logical violations but may indicate deeper issues. The primary example of the latter is an entity's types should all be subclasses or superclasses of each other.

When exceptions occur, the algorithm can optionally re-cluster entities using more selective thresholds, or, as in the implemented prototype, by requesting human input on the failing cluster. The algorithm then repeats the graph union step. These steps are iterated until no exceptions are logged, after which all of the cluster-produced graphs are merged into a single graph. The algorithm validates the unified graph; if this passes, the unified graph is returned as the authoritative metadata model.

Human-aided Disambiguation

When the algorithm logs exceptions for the entities in a given "bad" cluster, the algorithm can ask for external input on how to proceed. First, the algorithm asks if it should split

the bad cluster into two or more smaller clusters; this can be performed automatically by adjusting clustering hyperparameters or manually by specifying the new clusters explicitly. If reclustering occurs, then the algorithm begins another iteration of the graph union phase above using the new clusters.

If reclustering does *not* occur for a "bad" cluster, then the algorithm asks for manual resolution of the graph contents before proceeding to the next cluster. This typically involves choosing which Brick class to assign to a group of entities, but may also require editing properties and relationships of entities. The algorithm saves the results of manual resolution so they can be applied during future runs of the reconciliation algorithm.

The reconciliation process can use human feedback to learn how to automatically cluster, classify and disambiguate entities as well as reduce the amount of manual resolution needed. Although this has not been implemented in the current proof-of-concept, the continuous integration architecture can support active learning techniques such as [27].

Reconciliation Example

The behavior of the algorithm can be illustrated with an example of merging the metadata from Haystack and BuildingSync models for a building. The wrappers for these two sources produce the Brick metadata listed in Figure 6.10. The algorithm begins by clustering the entities. The string-matching phase places `bldg:bsync-meter` and `bldg:ph-meter` into the same cluster because their labels are sufficiently similar. The `bldg:bsync-ahu-1` and `bldg:ph-rtu-1` entities are not grouped because the labels are too dissimilar.

The type-aware phase examines the Brick-defined classes for the remaining entities. Using the Brick ontology, the algorithm infers that because `brick:Air_Handling_Unit` is a superclass of `brick:Rooftop_Unit`, each source has metadata for one air handling unit. Because each source has the same number of instances of that type, the algorithm clusters those entities by label similarity. This results in `bldg:bsync-ahu-1` and `bldg:ph-rtu-1` being clustered. The difference in specificity between the original sources is due to the fact that BuildingSync does not differentiate between subclasses of air handling units, but Haystack does.

The algorithm proceeds by unifying the triples for the entities in each cluster and validates the logical and semantic soundness of the resulting graph. In this simple example, the algorithm only needs to verify that the types of each pair of entities are compatible. This is true: `brick:Air_Handling_Unit` is a superclass of `brick:Rooftop_Unit` and `brick:Power_Meter` is a superclass of `brick:Building_Power_Meter`. Finally, the two graphs are merged into a single Brick model (Figure 6.11).

The reconciliation algorithm and metadata integration architecture are implemented in a fully functional prototype. This allows the evaluation of the approach on a set of real and artificial sites.

Site Name	Metadata Source	% Contributed	Unique	Model Size (Triples)	# manual interventions
Carytown	Haystack	32.9%	100%	280	0
	BuildingSync	20%	100%		
DOE Medium Office	Haystack	31.9%	100%	1,698	4
	Modelica	41.9%	98.2%		
	BuildingSync	.8%	98.5%		

Table 6.1: The results of merging multiple metadata models for two different sites, showing the diversity of the metadata available between the available metadata sources. The *% Contributed* percentages do not add up to 100% because the rest of the graph consists of inferred metadata not contained in any particular model.

Evaluation of Integration Server

The reconciliation algorithm and integration server have been realized in a Python implementation. The server exposes the API endpoints required of the metadata synchronization protocol and logs all `sync` messages received from wrappers in a SQLite database. The triples in these messages, which contain the inferred Brick metadata from each wrapper, are inserted into a dedicated table and indexed by their metadata source and timestamp. This allows the definition of a SQL View that contains the most recent triples for each wrapper, which is used as input to the reconciliation algorithm. The server incorporates Allegrograph's reasoner implementation to perform the required logical validation of the Brick metadata [55]. The server also embeds an in-memory instance of HodDB [50] to support application queries against Brick metadata.

Table 6.1 contains the results of reconciling the Brick metadata from each source for each site. The *% Contributed* column contains the proportion of triples in the unified model that were contributed by each source; this includes redundant triples. The *Unique* column contains the proportion of triples in the unified model that are unique to each source.

Although there are only a few sites and models, it is possible to observe some general behavior about the metadata extracted from the available wrappers. First, the metadata from Haystack and BuildingSync wrappers are mostly complementary and there is little overlap between them. This aligns with the respective scopes of each metadata source: BuildingSync describes holistic properties of systems that may not be covered by Project Haystack models (at least in a standard way). Secondly, Modelica wrappers provide more Brick metadata than Haystack wrappers: a Modelica/CDL model can produce a significant portion of the Brick metadata for a building. This aligns with the detailed treatment of HVAC systems found in Modelica models compared with the coarse-grained modeling found in Haystack.

For all sites, the metadata common to all wrappers was very low. This is to some extent due to the completeness of the wrappers at time of writing, but is also limited by the different levels of detail and different perspectives of a building that are communicated by different metadata sources. The metadata contributed from each wrapper was almost completely

unique: even though there is some overlap in the entities described by each wrapper, it is rare for two wrappers to produce Brick metadata at the same level of detail or level of completeness. For example, one sensor was identified as a `brick:Flow_Sensor` by the Modelica wrapper and a `brick:Return_Air_Flow_Sensor` by the Haystack wrapper.

Discussion

The development and evaluation of the proof-of-concept implementation demonstrates that integrating metadata from many different sources is not only practical, but also yields a richer and more complete representation than any individual source. The resulting unified metadata model enables self-adapting applications.

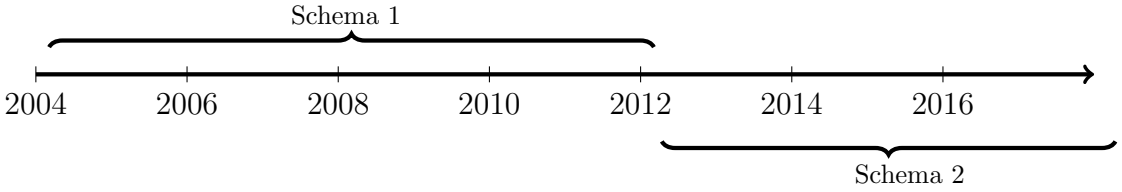
The metadata synchronization protocol successfully decouples the tasks of inferring Brick metadata from a particular source and integrating multiple sources of Brick metadata into a unified model. This establishes a common platform for Brick metadata inference research; for example, inference methods such as [27] and [82] that operate on ad-hoc metadata representations can be adapted to the protocol. This permits direct comparison of the Brick metadata produced by different methods, and eases the integration of these methods with other wrappers. The protocol also offers a clear path for future metadata standards such as ASHRAE's 223P [8] to support or integrate with Brick. Future work will develop the existing wrappers to deliver more complete and accurate Brick metadata, and implement additional wrappers that operate on historical telemetry and unstructured data like BMS labels.

Experiences with the reconciliation algorithm demonstrate that the extension of record linkage techniques to support semantic metadata graphs can successfully produce useful Brick models. Due to the lack of descriptive labels, record linkage using type alignment was much more effective than string matching for producing clusters of entities. In particular, autogenerated labels in Haystack and Modelica models caused a number of false positives when using string matching. Despite these difficulties, the algorithm was able to detect the resulting semantic issues in the merged model by using the Brick ontology. Future work will augment the reconciliation approach with active learning capabilities that can apply human input to automatically perform the required clustering and disambiguation.

6.4 Metadata Management Over Changing Semantics

The structure and semantics of metadata representations also change over time and must be managed. These changes are driven by churn in the domain being modeled, the data being stored, or the requirements of the application. This presents a dilemma for database administrators and application authors alike: do they undertake the expensive and largely manual process of rewriting existing applications to take advantage of the new schema? Does data stored against older schemas need to be transformed or adapted to the new schema?

Building	Meter	Timestamp	Value
Soda Hall	Electric Main #1	2012-01-11	13464800 kWh
Soda Hall	Electric Main #1 Demand	2012-01-11	265.853 kW
Soda Hall	Steam	2012-01-11	19063400 Lbs



Building	Meter	Timestamp	Value
Soda Hall New (as of 1/12/12)	Electric Main #1	2012-01-12	14581085 kWh
Soda Hall New (as of 1/12/12)	Electric Main #1 Inst	2012-01-12	166.14 kW
Soda Hall New (as of 1/12/12)	Steam Meter	2012-01-12	474540 Lbs

Figure 6.12: Visualization of the campus meter database demonstrating how the semantics change independent of the structure. Schema 1 was active until January 12th 2012 at which point Schema 2 becomes the active schema.

If so, how does one determine how well historical data is described by a new schema, and what are the implications of the new structure?

These issues can be addressed by augmenting existing metadata schemas with ontologies which define the formal semantics of the descriptive, non-data fields in the schema — i.e. the *meta-metadata*. Existing methods for handling data evolution concentrate wholly on either the structure (schema) of a database or the semantics of an ontology, but never both. This chapter proposes a different approach in which the taxonomy and database are co-evolved through subsequent versions. Rather than continually migrating historical data stored against prior schema versions to the most recent version to support application queries, this approach *rewrites* application queries to match historical schema versions. The evolution of the taxonomy can inform and in some cases completely automate this process.

The following section describes early work on identifying and characterizing the problem of semantic evolution. This includes the proposal of *segmented query generation*, a new technique for rewriting queries over historical and future versions of a schema that takes into account the semantic evolution of that schema. The full implementation and evaluation of this technique are the subject of future work.

Semantic Evolution Case Studies

Three real-world case studies, including some from outside of the buidling domain, motivate the need to handle changes in semantics over time.

NSF Survey of Earned Degrees [102]: The National Science Foundation (NSF) con-

ducts an annual census of all research doctoral degrees received from accredited U.S. institutions. The degrees are classified according to a taxonomy developed by the NSF; this provides the basis for aggregate statistics such as the number of Computer Science degrees earned in a given year. The degree classification taxonomy has changed over the course of the survey, which began in 1957, to reflect the evolution of degree programs (Figure 6.13).

Because degree programs do not evolve in "lock-step" with one another, the emergence and refinement of newer degree areas such as Computer Science are not immediately reflected in the taxonomy or the survey results. Furthermore, the process of retroactively applying newer classifications to historical data is largely manual and thus slow.

Campus Meter Database: The UC Berkeley campus contracts with an external provider for collecting and storing meter data tracking energy, power, steam and water consumption over time. The stored data is organized into a collection of tables — one for each building — containing timeseries data for each measurement capability of a meter (referred to as a *field*). The set of fields for each building is different: the field names provide some context for what the meter measures and how it relates to the building. The specific quantity being measured is left implicit.

Over time, meters have been replaced, resulting in different names for meter fields, or recalibrated, resulting in a different distribution of data. These changes are reflected in the database as new tables — for example, the table "Soda Hall" becomes "Soda Hall new (as of 1/12/12)" — which raises challenges for historical analyses (Figure 6.12).

Brick Ontology: As the Brick ontology evolves to introduce more classes or to refine the definitions of existing classes, existing databases that depend on older versions of Brick must decide how to adapt. This involves properly translating queries to take advantage of new classes or to reinterpret older classes.

Formalism and Preliminaries

This section formalizes the semantic evolution question for relational databases.

Definition 6.4.1. *Relational Schema. A relational schema R is a set of attributes $a^1, \ldots, a^n$. Each attribute a^i has a name given by name(a^i). The range of values for an attribute a^i in relation R_j is given by A^i_j. The number of attributes in a relational schema R_i is given by $|R_i|$.*

Definition 6.4.2. *Database Instance. A database instance D_i is a set of n tuples $\{d_1, \ldots, d_n\}$ where each tuple* matches *a relational schema R_i (written as $D_i \in R_i$). This means that each tuple $d \in D_i$ obeys $|d| = |R_i|$ (has the same number of attributes) and $d^j \in A^j_i$ (the jth attribute in tuple d has a value within the permissible range given by A^j_i).*

The subscripts on relational schemas R_i and database instances D_j are logical timestamps which increase whenever a new relational schema is introduced. A logical timestamp t corresponds to a physical segment (extent) of time that may or may not be explicitly modeled in the schema. This notation allows the expression of sequences of schemas over time, e.g.

$[R_i, \ldots, R_k]$. All logical timestamps in a sequence are unique and are ordered from smallest (least recent) to largest (most recent).

Definition 6.4.3. *Metadata. In the context of relational databases, metadata properties are the set of attributes in a relational schema that provide context or define properties of some data. The semantics, or meaning, of each attribute determines what is metadata and what is data.*

In the campus meter database example (Figure 6.12), the *data* is the combination of a timestamp and a value. The table contains metadata in the form of which meter originated the data and which building the meter is associated with.

To simplify the discussion, full temporal segmentation is assumed: a database instance D_i is assumed to contain *only* records for the temporal extent corresponding to its logical timestamp, and that records for that temporal extent are only contained within database instance D_i.

The key treatment of each relational schema is to consider each discrete value of an attribute's domain A^i_j as an OWL-style class. For example, each value in the `Meter` column of the schemas in Figure 6.12 can be considered a label of an independent class. Formally, each value v in the domain of attribute A^i at schema version j is a new class C^i_{jv}. This enables a semantic evolution process to reason about the relationship between classes *within* a version using a taxonomy, and *between* versions using a semantic evolution operator[2].

There are three semantic evolution operators:`same`, `merge` and `split`. Together, these characterize most kinds of semantic evolutions. For any pair of classes C_i and C_j for two schema versions i and j:

- `same`(C_i, C_j) indicates that the semantics of a class are unchanged between the two versions
- `merge`(C^1_i, C^2_i, C_j) indicates that the content of class C_j is defined by the union of two (or more) classes from the previous schema version (C^1_i, C^2_i).
- `split`(C_i, C^1_j, C^2_j) indicates that the content of class C_i at a prior schema version has been split into two (or more) new classes in the later schema version (C^1_j, C^2_j).

A process rewriting schema i queries to execute on schema j can use these operators to inform the automated transformation of those queries. `same` operators do not change the query (except to handle renamed fields). `split` operators result in a query which expresses the union of the queries generated by substituting C_i with each of the new C^k_j classes that it was split into. This can be handled automatically. `merge` operators require a disaggregation function to tell the rewritten query "how much" of the new class C_j matches the original class C^1_i.

[2]For simplicity, a taxonomy (tree-base class organization) is used instead of a full ontology which can describe a DAG class organization.

Taxonomy Evolution	Type
$\text{Engineering}_{2004} \subseteq \text{Engineering}_{2010}$	Same
$\text{Econ}_{2004} \subseteq \text{Econ}_{2010}$	Same
$\text{Psych}_{2004} \subseteq \text{Psych}_{2010}$	Same
$\text{Math/CS}_{2004} \subseteq \text{CS/Info Sci}_{2010} \cup \text{Math}_{2010}$	Split
$\text{Other}_{2004} \subseteq \text{Anthro}_{2010} \cup \text{Other}_{2010}$	Split
$\text{Psych}_{2004} \cup \text{Social Sci}_{2004} \subseteq \text{Social Sci}_{2010}$	Merge

Table 6.2: A subset of the split/merge/same relationships between the 2004 and 2010 versions of the NSF degree taxonomy in Figure 6.13

Figure 6.13 illustrates a subset of two versions of the NSF degree classification taxonomy, one from 2004 and another from 2010. Table 6.2 contains a few of the binary relationships between classes across the two versions of the taxonomy.

The set of Engineering degrees recognized by NSF did not change between the two versions, and thus the relationship is *same*. The Mathematics and Computer Sciences (Math/CS) category from 2004 was *split* into two separate categories in the 2010 taxonomy. Similarly, the Other category under Social Sciences was *split* to pull Anthropology degrees out.

There is an interesting *merge* between the two versions: the Psychology category is incorporated under the Social Science category in 2010 despite these being separate in 2004. The 2004 Psychology category is also involved in a *same* relationship with the 2010 Psychology category because they semantically refer to the same concept.

These three types may be broken down into expressions of relational operators that more specifically describe the nature of the relationship between two versions of a taxonomy. Nonetheless the three types are enough to inform query generation for a wide family of semantic evolutions.

Segmented Query Generation

Segmented Query Generation (SQG) is a proposed technique for transparently executing queries over a historical sequence of evolving relational schemas. This is one step of a larger, more complex process and is complementary to classic data cleaning and integration techniques. SQG operates by generating a new query for each historical schema version. This allows an application to transparently query multiple database versions as if they were under a unified schema. SQG accomplishes this without support from the underlying database while reducing the effort required of a database administrator or application developer.

SQG takes advantage of semantic operators describing relationships between taxonomies to transitively and automatically apply SQG across multiple versions of the database. The existence of monotonic relationships (`same` and `split`) between subsequent versions of a taxonomy allows the segmented query generation process to make safe assumptions about

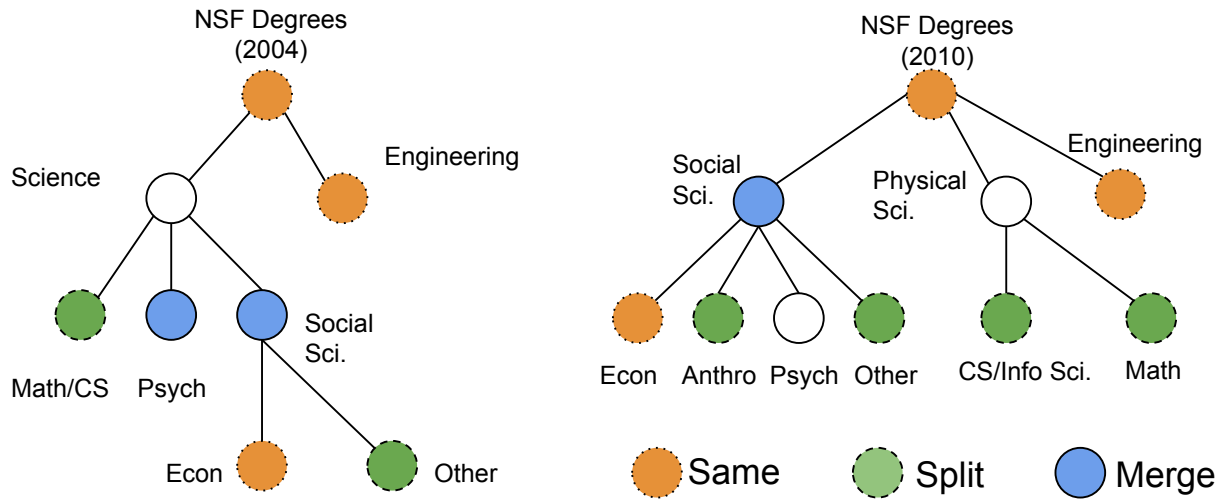


Figure 6.13: Two different versions of the NSF degree taxonomy. Colored and outlined nodes are involved in either a *split*, *merge* or *same* relationship across versions.

how the query can be rewritten. The existence of a non-monotonic relationship (`merge`) is an opportunity for a manual or semi-automated process to provide direction.

SQG operates by taking a query q targeting a schema R_i and produces a sequence of queries $q_0, \ldots, q_{i-1}$ each targeting a previous schema version. Executing each of the generated queries q_k against the corresponding database instance D_k for each prior schema version R_k produces a set of results as if all database instances were members of R_i.

A generated query is produced as follows. Given an input query q_j targeting schema R_j, the query processor first retrieves or computes the *semantic evolution* between R_j and the preceding target schema R_{j-1}. These taxonomies are C_j and C_{j-1}, respectively.

The query processor decomposes q into its relational algebra operators, rewrites each operator according to a set of rules, and reassembles the query. If attribute values a_j^1, a_j^2, a_j^3 are all metadata property values referenced in the query, then the query processor feeds these to a semantic mapping function to derive the generated values a_i^1, a_i^2, a_i^3.

A *semantic mapping function* uses the semantic operators, defined between pairs of schema versions, to rewrite attribute names and values between those schema versions. From this output the query processor can determine the mapped attribute name and value for any other schema version. Mapping functions only handle the immediately preceding schema version. Mappings between non-consecutive taxonomy versions are computed transitively.

This abstraction affords a variety of possible mapping functions implementations. For simple taxonomies with straightforward evolutions, the mapping may be stored as a lookup table. When the semantic mapping is unknown, a probabilistic mapping function may use external knowledge-bases or statistical inference to produce possible mappings [46].

Discussion

Any solution for addressing the issue of semantic evolution should fulfill the following four properties:

- **Correct**: the mechanism should be correct and produce valid schemas that preserve the semantics of the mapping
- **Transparent**: the mechanism (e.g. schema mappings and semantic mappings) should not affect how the query is expressed or executed
- **Efficient**: the mechanism should impose minimal overhead in terms of resource consumption at query time
- **Automatic**: the mechanism should minimize the human/manual effort involved

Work on SQG is still underway, but demonstrates promise in being able to handle the kinds of scenarios described above. SQG has the ability to be automatic, transparent and efficient because it produces queries which execute on the same optimized relational query processors as the original query, with little or no human input. The correctness of SQG has yet to be determined for all cases. The semantic evolution operators cover a variety of semantic evolutions, but it is not known how general they are. Regardless, the need to handle semantic evolution of schemas is an important issue with real-world applications, and should be further researched.

Chapter 7

Platform Design and Implementation for Self-Adapting Software

The final challenge in realizing a vision of self-adapting software is how to manage semantic metadata, at scale, on behalf of applications. Managing semantic metadata requires a platform that can store, update and serve semantic metadata as well as enabling access to the historical telemetry via the metadata. Linked data technologies — especially the formal axioms and rules defined in OWL and SHACL — place requirements on the metadata management platform that are not well-served by existing systems. In particular, the use of both linked data models and timeseries data

This chapter outlines the requirements of semantic metadata management and details why these properties are difficult to achieve at scale, in the context of existing data and metadata platforms. Then, the chapter presents two complementary platforms which facilitate self-adapting software at scale. The first, `reasonable`, provides a performant abstraction for ontology management that simplifies the use of semantic web technologies for platform developers. The `reasonable` software library enables new modes of interaction between linked data and other data models. Second, the chapter develops the implementation of the Mortar platform, which implements the hybrid linked-data/timeseries access required by self-adapting software programming models (Chapter 5).

7.1 Scaling Metadata Management

The question of how to store and access metadata at large scale has become increasingly important for big data and data warehouses [76]. Metadata in these settings describes the source and provenance of the data in the warehouse, including the transformations and treatments of that data. Many of the metadata management platforms developed to meet the needs of large data warehouses leverage a graph-based data model [70]. However, these platforms do not treat the *context* of the data as a first-class object: these kinds of semantic details are to be included in the data being described and are not handled by the platform.

This section explores how metadata is managed in existing platforms designed for cyberphysical and IoT settings. In these settings, the content of metadata critically influences how applications are written and how they find their data. This is especially true for self-adapting software.

Cyberphysical Data Platforms

A number of specialized databases have been developed for cyberphysical and IoT settings. These are predominantly characterized by the ability to ingest high rates of timeseries data from a large array of data sources such as sensors. IoT data is usually organized by attributes of the data source such as location or sensor type in addition to a unique identifier.

sMAP [39] is a protocol which uses a UUID as a primary key for each sensor channel that reports to a data archiver. Each sMAP data source is described with key-value metadata. A number of metadata attributes are required and have well-known definitions, including engineering units and the temporal resolution of the reported data. All other metadata attributes have arbitrary names, semantics and values. This design runs aground of most of the same consistency and interpretability issues as Project Haystack. Other systems like BuildingDepot 2.0 [147] and VOLTTRON [79] adopt similar key-value approaches to metadata: a small number of keys have well-defined semantics, but most attributes used by applications are not standardized.

NoSQL databases are often used in IoT settings due to their flexible data models. Document stores can easily store complex metadata that can describe a data source, but the lack of first-class timeseries support means that NoSQL databases demonstrate poor storage and query performance on large amounts of timeseries data [146]. Databases like IoTDB [146] and InfluxDB [101] adopt a slightly different data model. These store timestamped tuples of key-value pairs and tags. String-valued keys act as attributes of the data source and numerical-valued keys are treated as telemetry. The restricted data model enables more a more efficient and performant implementation. TimescaleDB takes this a step further by implementing specialized timeseries indices and storage in the Postgres RDBMS. These kinds of databases could provide access to telemetry on behalf of self-adapting software, but do not directly support the RDF data model and related technologies.

RDF Data Management

RDF databases typically focus on efficient storage of millions or billions of triples and providing performant SPARQL queries over those triples. Research into how to improve these databases has focused mostly on custom compression and run-length encoding schemes [105], bitmaps [90] or other specialized indices [67] and much less on Selinger-style [130] query optimization techniques for SPARQL execution. As a result, many SPARQL query processors are designed for a single kind of workload: non-interactive but simple queries on very large knowledge bases. On this workload, modern SPARQL query processors use graphics

cards [138] or optimized relational query processors [55] to achieve execution times of seconds or minutes.

For self-adapting software, SPARQL queries are part of the "hot path" of code that is conducting analytics or controls; even a simple application may run several SPARQL queries in order to configure its own execution. [50] studied the performance of common RDF databases on a Brick workload and observed that the storage optimizations made for large scale storage do not always translate to performant queries on smaller graphs. The highly-recursive queries characteristic of many Brick applications constitute a pathological worst-case for the index structures used for large RDF datasets. Thus, there is a need for SPARQL query processors that can "scale down" to provide good performance on smaller datasets.

However, most RDF databases and SPARQL query processors do not support OWL or SHACL reasoning. Notable exceptions are Apache Jena [140], which supports reasoning through a plugin system, Protege [110] and TopBraid. The exact computation required to perform OWL or SHACL reasoning depends on the specific ontology language an ontology is defined in, and which axioms or features it uses. Existing academic research has explored different ways of scaling reasoners for OWL (such as using Map Reduce [142]). However, much less work has focused on scaling SHACL reasoning (due to it being a more recent standard) and the newer OWL ontology languages. Brick is defined in the *OWL 2 RL* profile language [154], which has the advantage of being computable in polynomial time using rule languages such as Datalog. Not all OWL 2 reasoners support the RL profile [58]. A notable exception is RDFox [103], which implements an in-memory Datalog engine providing performant SPARQL query processing and OWL 2 RL reasoning in the same package.

Most databases designed to support RDF do not provide support for versioned or temporal graphs. One reason for this deficiency is that the open-world assumption underlying many semantic web use cases is incompatible with the idea of a graph at a point in time. Temporal graph management must support inquiries of the form *what is the content of the graph at a given point in time*; this requires knowledge of which elements of a graph exist or are valid for a given temporal extent. However, under OWA the content of the graph is not closed. This means that it is impossible to differentiate between whether a given fact is *false* at a given point in time in the graph, or if it is simply not included in the graph. Most databases with support for versioning focus on dataset management, such as for scientific publication. TerminusDB [113] supports a linked data graph model, but other systems like Datahub [24], ProvDB [94], OrpheusDB [157] and DoltHub [43] support arbitrary structured (usually relational) and unstructured datasets. These general solutions do not provide support for RDF data management such as reasoning and graph queries.

Ontology-Based Data Access

Traditionally, RDF data management and timeseries data management techniques have largely remained separate. Ontology-based data access (OBDA) [30, 156] demonstrates how RDF-based metadata can contextualize the data stored in an external database. In

OBDA, the ABox (instance metadata and data) is not stored in the RDF database but in an external database that is mapped into the RDF data model. The RDF database manages the TBox (ontology definition) and can be accessed by the client to provide the semantics for the instance data.

OBDA systems address the same *impedance mismatch* between the capabilities of RDF databases and databases for other data models (specifically SQL in [30]) that makes self-adapting software difficult to implement. While it is possible for RDF to encode timeseries data [71, 104], the resulting models are cumbersome and verbose. RDF is already an unfamiliar technology for most data scientists, most of whom would rather execute SQL queries or download CSV files than author lengthy SPARQL queries against graphs where the actual telemetry is 2 or 3 levels removed from the name of the data stream. Data structures for efficient storage and retrieval of timeseries data, particularly over temporal ranges, are not well served by the graph data structures used by most RDF databases.

How to provide these features in a performant manner at scale remains relatively unexplored for analytics-heavy workloads. OBDA research focuses on data integration, mostly revolving around how integrity constraints over ABox data, such as that stored in a relational database, can be expressed and enforced in terms of the TBox. Support for data-heavy analytics workloads is relatively unexplored.

7.2 `reasonable`: Abstracting Ontology Management

Self-adapting software requires a deeper integration between RDF and timeseries data management systems. `reasonable` is a software library providing efficient, high-level abstractions over RDF graphs and OWL 2 RL ontologies which facilitate the self-adapting software use case. It acts as a conduit between a versioned triple store and a SPARQL query processor: it applies OWL 2 RL reasoning to the triples at a point in time and allows the SPARQL query processor to operate over the augmented graph. `reasonable` is based on a differential Datalog engine and makes use of novel, semantics-preserving transformations of the OWL 2 RL rules to achieve moderate to significant speedups on the reasoning task.

Overview

`reasonable` is a Datalog implementation optimized for OWL 2 RL reasoning over different graphs over time. Figure 7.1 illustrates the logical architecture of `reasonable` in the context of the Mortar platform explored in the next section. `reasonable` ingests a set of triples constituting the content of an RDF graph at a point in time, materializes the OWL 2 RL entailment of that graph, and makes the resulting triples available to a SPARQL query processor.

A triple store, at left, stores the content of many RDF graphs over time. These triples are stored as `g,d,s,p,o,t` tuples. Recall that a graph g is an arbitrary collection of RDF triples; in the cyberphysical context, a graph usually describes a single environment. Updates to

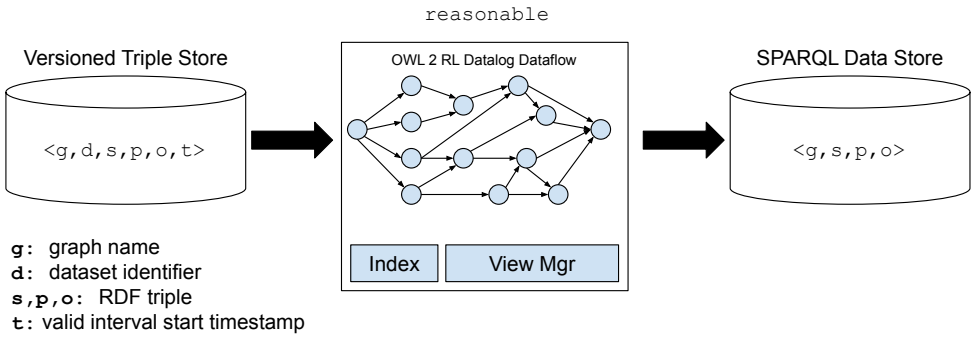


Figure 7.1: Logical architecture of `reasonable` and its interaction with other software components of Mortar

the graph come from *datasets*. A dataset d is a particular source of triples for a particular graph at a particular point in time t; the content of the data source may change over time, but the identity does not. Examples of datasets would be the content of the Brick ontology or the Brick model produced by the integration server described in Chapter 6. s, p and o are the components of an RDF triple in that dataset at that point in time.

During operation, for a given timestamp t and graph g, `reasonable` ingests the *latest* triples for each dataset for that graph. This represents a snapshot of the content of the graph at that time. `reasonable` then computes the OWL 2 RL entailment of the graph by pushing the triples through the dataflow graph. The resulting triples are then exposed to a SPARQL query processor.

Computing OWL 2 RL Entailment

Recall from Chapter 2 that OWL 2 RL is an ontology language that can be computed with a rule engine. Prior work has established that Datalog is sufficient to compute these rules [141, 103]. [100] lists the standard axiomitization of OWL 2 RL as a set of Datalog rules (see [2] for a detailed explanation of Datalog and its properties). A Datalog rule can be written as a Horn clause with a *body* — intuitively, the conditions that must be true for the rule to "fire" — and a *head* — the output of the rule. In [100] each of the Datalog rules is written with respect to a single relation `T(s,p,o)` which contains all of the RDF triples in a graph.

The OWL 2 RL rules define which triples can be added to the graph that are implied by the statements contained within the graph. An OWL ontology is an RDF graph which instantiates the rules to model the semantics of some domain. To compute the entailment of a graph the ontology definition must be loaded into that graph. In `reasonable`, the Brick ontology definition will often be included as a separate dataset as the description of the cyberphysical environment. These two datasets are unioned together when the triples are ingested into `reasonable`.

```
% cax-sco rule for class type inheritance
T(X, "rdf:type", C2) :- T(C1, "rdfs:subClassOf", C2), T(X, "rdf:type", C1).

% scm-sco rule for transitive subclassing
T(C1, "rdfs:subClassOf", C3) :- T(C1, "rdfs:subClassOf", C2), T(C2, "rdfs:subClassOf", C3).
```

Figure 7.2: Two OWL 2 RL rules expressed in Datalog. `T` is the relation `(s,p,o)` corresponding to the triples in an RDF graph.

```
% cls-int1 rule for implementing intersection classes
T(Y, "rdf:type", C) :- T(C, "owl:intersectionOf", X), LIST[X, C1, ..., Cn], T(Y, "rdf:type", CI).
% LIST[X, C1, ..., CN] expands to
T(X, rdf:first, C1), T(X, rdf:rest, Z2),
T(Z2, rdf:first, C2), T(Z2, rdf:rest, Z3),
% ....
T(Zn, rdf:first, CN), T(Zn, rdf:rest, rdf:nil)
```

Figure 7.3: Definition of the `LIST[]` syntax in [100] and an example of a variadic Datalog rule that uses a variable-sized list.

Figure 7.2 lists two OWL 2 RL rules. The first rule, `cax-sco`, implements subtype polymorphism: if an entity `x` is an instance of class `C1` and `C1` is a subclass of `C2`, then `x` is also an instance of `C2`. The first atom, `T(C1, "rdfs:subClassOf", C2)`, matches TBox expressions that describe the structure of a class organization. The second atom, `T(X, "rdf:type", C1)`, matches ABox expressions that specify the type of entities in the graph. The head of the rule produces a new triple which indicates the new inferred type of the entity `x`. The second rule implements the transitive nature of the `rdfs:subClassOf` relationship.

While OWL 2 RL is implementable in Datalog, it is not necessarily convenient to do so. Most of the rules defined in [100] have a fixed size body. This is the default form of Datalog rules and is how essentially all Datalog implementations expect the rules to be expressed. However, a handful of rules make use of RDF lists and are thus defined with variable-sized bodies. Figure 7.3 illustrates one of the OWL 2 RL rules using this variadic form.

This poses a challenge for Datalog rule engines: the rules cannot be be hard-coded into a Datalog program because the exact form of the rule depends on ABox information. Proper handling of variadic rules requires Datalog rules to be *generated* dynamically when the graph is observed. This feature is not supported by most Datalog implementations. RDFox [103] addresses this issue by pre-compiling the Datalog rules as a function of the input RDF graph. However, if the graph changes then this potentially expensive recompilation will need to be redone. Further, if new rules are generated by through the computation of the OWL 2 RL entailment, then the Datalog engine will need to regenerate a new Datalog program that can compute the new rules.

To be effective, `reasonable` must be able to nimbly switch between ontology versions which may contain different instantiations of variadic rules. `reasonable` addresses the variadic rule problem through a novel rule rewriting technique for OWL 2 RL rules. The rewritten rules can be computed monotonically and preserve the semantics of the rules but are not expressed in Datalog. Moreover, implementing the rewritten rules reduces the number of joins that need to be performed during execution, which improves performance.

Semantics-Preserving Rewriting for Variadic Rules

The goal of the rewriting technique is to not require an external recompilation of the Datalog program when new instantiations of variadic rules become available in the graph. These rules are much larger than most OWL 2 RL rules and can require many joins; for example, the `cls-int1` (Figure 7.3) rule has a body of size $2N + 3$ where N is the number of classes in the intersection condition. By avoiding recompilation, which typically involves throwing away program state and restarting the process, a Datalog engine can maintain partially materialized rule evaluations that accelerate future computation.

There are two challenges to variadic rule evaluation that must be addressed. First, because the instances of the variadic rules are derived from an *unordered* set of triples, the rewritten rule must be able to create the lists of conditions (`LIST[X, C1, ..., CN]` in Figure 7.3) in an efficient manner. Secondly, the rewritten rules must be able to determine when all of the conditions contained in the list are correct. The rewritten rules are not expressed in Datalog; instead, they take the form of a small piece of monotonic logic that can be computed as part of a fixed-point Datalog computation scheme. The logic is executed on every iteration of Datalog computation and makes use of several monotonic data structures.

To address the first challenge, the rule rewriting technique makes the observation that the use of RDF lists is intended to make the list a *closed* collection; that is, no other statements can be spliced into the list or appended to the beginning or end of the list. This is true because the head of the list is known (the list is identified by the beginning entity `x`) and the end of the list is fixed (`rdf:nil`). However, the inherent order of the list does not actually factor into the semantics of the rule. Specifically, changing the order of the elements in the list (`C1` through `CN` in Figure 7.3) does not change the output of the rule; this is true for all variadic rules in OWL 2 RL. This observation means that the contents of the RDF lists can be captured with a set rather than a linked list, as they are normally represented in RDF. This also means that the content of all RDF lists in a graph can be determined in a single linear pass over all of the triples by using a disjoint-sets data structure.

This construction works as follows. Recall that the disjoint-set data structure supports three operations. `Add(x)` adds an element `x` to its own set. `Find(x)` returns the set containing `x`. `Union(x,y)` merges the set containing `x` with the set containing `y` into a single set. The list discovery algorithm requires a single disjoint-sets data structure D. During a pass over all of the RDF triples in a graph, any triple with a predicate of `rdf:rest` has both its subject and object `add`ed to D and then `union`ed. The object of a `rdf:rest` triple acts as a pointer to the subject of the next `rdf:rest` triple describing the structure of the list. By

```
1 % original cax-sco rule for class type inheritance
2 T(X, "rdf:type", C2) :- T(C1, "rdfs:subClassOf", C2), T(X, "rdf:type", C1).
3 % rewritten cax-sco rule
4 PSO(P, S, O) :- T(S, P, O). % predicate-indexed relation
5 Instances(X, C) :- PSO("rdf:type", X, C). % entity-indexed type relation
6 Subclasses(C1, C2) :- PSO("rdfs:subClassOf", C1, C2). % class-indexed subclass relation
7 T(X, "rdf:type", C2) :- Subclasses(C1, C2), Instances(X, C1).
```

Figure 7.4: The `cax-sco` rule implemented to take advantage of intermediate relations

`union`ing the heads and tails of the list segments, the sets in D will converge to the contents of all the lists defined in the RDF graph. The order of the values of the lists is not preserved, but this is not required for correctness.

Once the content of the lists are known — after one pass over the input triples — the rewritten rule uses a `Map<URI, Bitset>` named `conditions` to determine when all of the rule conditions in each list are met for each URI. The map has an entry for each URI in the dataset; the value of the map is a bitset of size N where each position $i \in 1, ..., N$ is 1 if the condition Ci in the list is true for that URI and 0 otherwise. On each iteration of the Datalog engine, the rewritten rule checks each of the conditions for each URI and updates the bitset accordingly. If all of the elements of the bitset are set and the other conditions of the rule are met, then the rule fires. One disjoint-sets data structure is used to capture all RDF lists in a graph. A unique `conditions` map is created for each variadic rule.

Redundant checks can be mitigated with two optimizations. First, the rewritten rule checks the other atoms of the body before updating the bitset. This avoids performing the checks on URIs that will never meet the other conditions of the rule. Secondly, the rule checks the value of the bitset before evaluating all of the conditions to avoid rechecking URIs that have already caused the rule to fire.

Implementation

`reasonable` is built on DataFrog [93], a simple, embeddable Rust-based Datalog engine. Because of the embeddable nature of the engine, there is no provided runtime. The host program has to build and repeatedly evaluate the rules until no additional tuples are produced. This constitutes a fixed-point evaluation scheme.

The formalization in [100] only uses a single relation `T(s,p,o)` In a fixed point iteration evaluation scheme, any new tuples produced by a relation will propagate to all dependent relations. A rule is dependent on rule relation if its body contains an atom that is the head of another rule. Each time a new tuple is generated by a rule, all dependent rules must be run that iteration. The iterations stop when no new tuples are generated. Because of these dynamics, the use of a single relation, `T(s,p,o)`, to implement all OWL 2 RL rules makes all rules dependent on one another. This means that if *any* rule generates a new tuple, all

"downstream" rules must be reevaluated, even if that rule would otherwise not fire.

To avoid this issue, `reasonable` redefines the OWL 2 RL rules to make use of intermediate relations. This reduces the total number of iterations by removing dependencies between unrelated rules. Consider the `cax-sco` rule as written in Figure 7.4 as an example. As written, the rule would need to be reexecuted each time the `T` relation changed. Figure 7.4 shows the implementation of `cax-sco` that `reasonable` uses. The implementation makes use of 3 intermediate relations: `PSO(P,S,O)` is a predicate-indexed relation of the tuples in the RDF graph; `Instances(X,C)` is an entity-indexed relation of the types `c` of each entity `x` in the graph; `Subclasses(C1,C2)` is a class-indexed relation of the subclass relationships. These relations do not typically change after the first couple of iterations of Datalog evaluation.

`reasonable` is implemented as a Rust library in order to embed Datafrog and to facilitate embedding in other programs. The Datafrog engine was chosen because it was simple to extend with monotonic, albeit non-Datalog, logic such as the rule rewriting technique described above. The `reasonable` implementation is 3182 lines of code. This does not include Datafrog but does include the binding code to support a Python 3 package that provides a high-level interface to `reasonable`[1].

Currently, `reasonable` can receive triples from Postgres and SQLite, using triggers to prompt the reevaluation of the OWL 2 RL rules when new data is inserted. `reasonable` does not implement any differential computation techniques to reduce redundant computation when triples are removed; instead, `reasonable` just reevaluates the entire Datalog program. `reasonable` can dump the output of the reasoning process into an in-memory or on-disk store which can be accessed by a SPARQL query processor.

Benchmarks and Evaluation

The performance of `reasonable` is evaluated by measuring the execution time of `reasonable` performing OWL 2 RL inference on a collection of RDF graphs. The RDF graphs are sourced from the Mortar dataset described in the next section; there are 109 graphs, ranging in size from 40 triples to 17,500 triples. `reasonable` is compared against OWL-RL, a Python package adopting a naive approach to OWL 2 RL evaluation, and Allegrograph [55], a commercial triple store with an optimized reasoner.

The experimental setup is as follows. For each RDF graph in the dataset, a new graph is created that has the Brick v1.1 ontology and RDFS ontology definitions loaded into the graph; this adds an additional 15,000 triples to each RDF graph. Each augmented graph is imported into a fresh instance of the reasoner — no state is preserved between runs — and the reasoner is triggered. A script measures the execution of the reasoning process as distinct from the time it takes to load the triples into each reasoner. OWL-RL is executed in-memory; Allegrograph is executed from a Docker container. Each RDF graph is reasoned 10 times.

[1] `https://pypi.org/project/reasonable/`

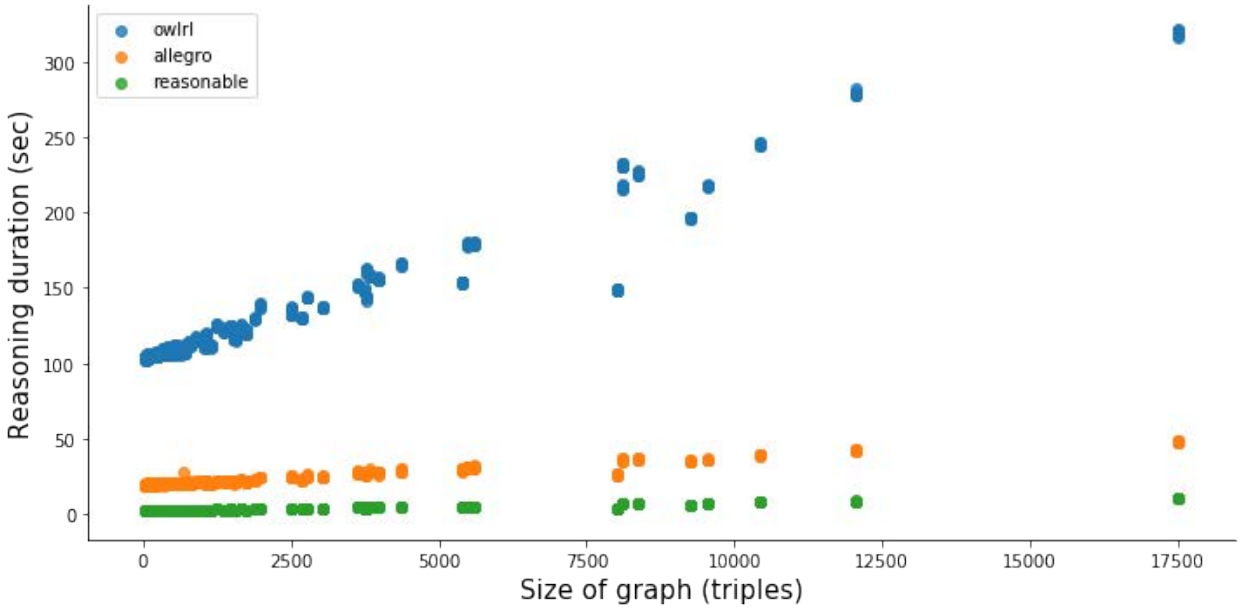


Figure 7.5: Comparing performance of `reasonable` with OWLRL and Allegro over more than 100 Brick models.

The results are summarized in Figure 7.5. On average, `reasonable` is $\tilde{7}$x faster than Allegrograph and $\tilde{4}$1x faster than OWL-RL. If the reasoning duration is normalized to the size of the input graph (before loading in the Brick and RDFS ontology definitions), `reasonable` is $\tilde{9}$x faster than Allegro and $\tilde{4}$8x faster than OWL-RL.

The improved performance of `reasonable` over available alternatives enables a reasoner to adopt a new role as a timely transformation of data that can be incorporated into an analytics workflow rather than an offline batch processing phase. In practice, waiting half a minute or longer for a reasoning step to complete severely impedes exploration and often requires the user to manually manage cached versions of the reasoned graph to avoid incurring the expensive computation.

7.3 Mortar: A Platform for Reproducible Building Science

Mortar — deployed at `https://beta.mortardata.org` — is a data platform which enables the execution and evaluation of self-adapting analytics applications over a large number of metadata-enriched buildings. This section describes the Mortar dataset and the architecture and implementation of the data platform that serves that data in a manner supporting self-adapting applications.

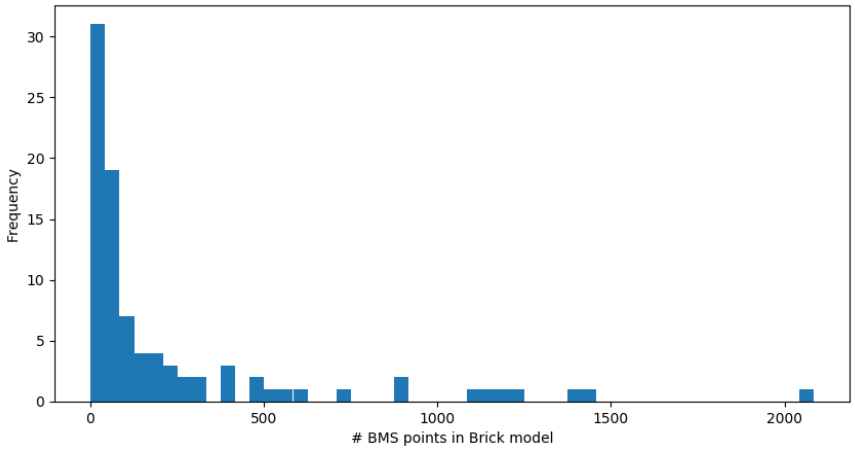


Figure 7.6: Histogram of number of data streams for all sites (μ =241).

Temperature Sensor	7380	Luminance Sensor	257
Occupancy Sensor	445	Pressure Sensor	148
Outside Air Temp. Sensor	362	Cloud Cover	32
Setpoints (generic)	2331	Power Meters	77
VAVs	4724	AHUs	467
HVAC Zones	4887	Dampers	1662
Non BMS Thermostats	123		

Table 7.1: Count of streams and equipment available in the testbed data set, aggregated by type. AHU and VAV totals include related equipment such as fans and pumps.

Public Dataset

Mortar contains timeseries and metadata for over 100 buildings, constituting over 9 billion datapoints and 750 million combined hours of telemetry. The majority of the data streams in the platform adhere to a 15-minute interval (.001 Hz), though some are more fine-grained (up to 1 Hz).

Figure 7.6 describes the distribution of the number of streams per building. Each building is accompanied by a Brick model that describes the building, its equipment and subsystems, available points, and references to timeseries data streams. Table 7.1 enumerates some of the types of available points and equipment in the testbed.

The majority of the dataset is made up of large commercial buildings belonging to a university campus. The buildings are typically used as offices, classrooms, research facilities and health care clinics. The average building has a floor area of 70,000 sqft, 3 floors and more than 100 rooms, while the largest building is a large library with a floor area above

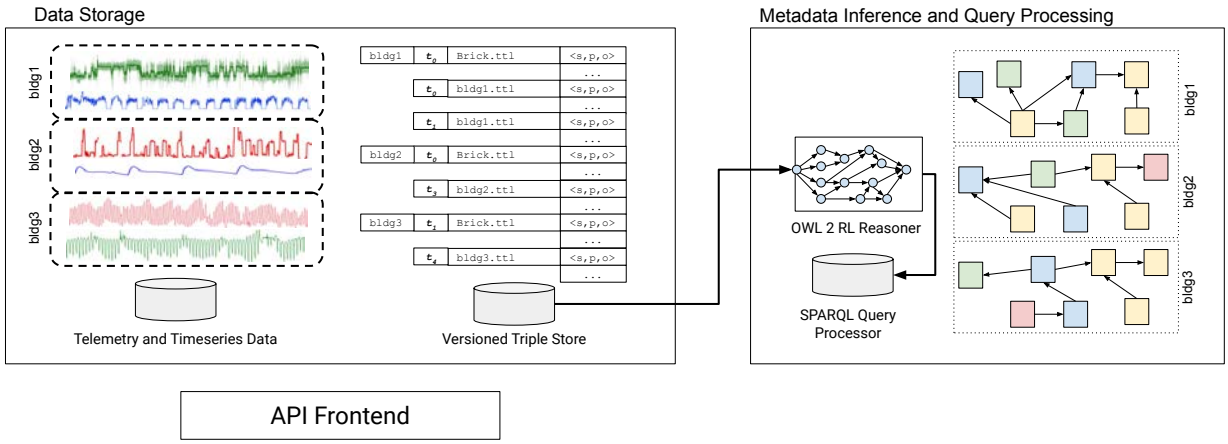


Figure 7.7: Mortar platform

400,000 sqft. Most buildings are conditioned using large built-up HVAC systems with air handlers and local distribution boxes, controlled by building automation systems. Chilled water and hot water are produced by a central plant and distributed through large pipes to most buildings. Some additional chillers are installed in some buildings to complement the central system.

Other buildings in the data set come from a set of independent data collecting efforts. Most of the non-campus buildings are part of an ongoing project to develop a building operating system; these are mostly small commercial buildings ranging from movie theatres to fire stations to animal shelters. Data collected includes sensors, setpoints and other data associated with thermostats, building meters, electric vehicle charging stations, HVAC and lighting systems as well as general occupancy, temperature, illumination and humidity sensors.

Platform Architecture

Mortar — which originally stood for **M**odular **O**pen **R**eproducibility **T**estbed for **A**nalysis and **R**esearch — is a platform for storing contextualized timeseries data to support self-adapting analytics applications. Mortar supports the long-term storage of many timeseries streams and RDF graphs that describe and contextualize that data. The management and access of timeseries data and RDF metadata is provided through a declarative API frontend designed for high numbers of long, concurrent connections typical of clients downloading large amounts of data. The API frontend supports the operations supported by self-adapting software (Chapter 5).

Figure 7.7 illustrates the architecture of the Mortar platform. The data storage component contains two logical databases: a timeseries database that organizes data by graph name (`bldg1`, `bldg2`, `bldg3` in the figure) as well as stream name, and a triple store using the

`g,d,s,p,o,t` schema described in §7.2. The versioned nature of the triple store is illustrated in the figure: for `bldg1`, the triples corresponding to dataset `Brick.ttl` at t_0 and the triples for dataset `bldg1.ttl` at t_1 will constitute the latest version of the graph.

The reasoner wraps the `reasonable` library in a subscription to the versioned triple store. The reasoner maintains the materialization of the most recent version of each graph in the triple store and makes this available to a SPARQL query processor. As new triples are added to the triple store, the reasoner is able to update the materialized graph using a minimal amount of work; this facilitates bulk uploads. Historical versions of a graph are evaluated on demand and cached.

Platform API

An API frontend exposes a set of operations to clients, documented in Table 7.2. These methods interact with both the timeseries and RDF database (triple store). The `register_stream` method adds metadata to the RDF graph about a particular data stream, uniquely identified by a 36-byte UUID. All data streams must exist within exactly one graph. Calling `register_stream` adds the data stream to that RDF graph; providing the extra arguments allows Mortar to add additional triples such as `rdfs:label` for `name`, `rdf:type` for `class` and `brick:hasUnit` for `units`. Any data stream that is registered without a `class` argument will be asserted as an instance of `brick:Point` — the most generic kind of data source. The `add_*` methods support streaming JSON POSTs as well as batch uploads from Turtle files (for metadata) and CSV files (for data).

`qualify` takes as an argument a list of SPARQL queries $q_0, \ldots, q_m$ and an optional list of graph names $g_0, \ldots, g_n$ and returns an $m \times n$ matrix T where each entry T_{ij} is the number of tuples returned by running query q_i on graph g_j: $T_{ij} = |q_i(g_j)|$. This allows a client to quickly determine which graphs in Mortar contain the data and metadata necessary to run an application. If a list of graph names is provided, `qualify` only executes the queries on those graphs, otherwise it executes on all stored graphs.

`get_metadata` takes a SPARQL query as an argument and returns the results of the query executed over the indicated graphs. Because of `reasonable`, the SPARQL queries are always served over fully reasoned graphs.

`get_data` takes a SPARQL query as an argument and returns the timeseries data and metadata for all data streams described by that query. The `start` and `end` arguments bound the temporal extent of the query, and `agg` and `window` describe any requested server-side aggregation (e.g. aggregating each data stream to a 15-minute average). This requires the API frontend to determine which elements of the SPARQL query correspond to timeseries data. Nominally, any instance of `brick:Point` represents a source of data; however, not all of these will have data associated with them, and the name (URI) of the entity needs to be associated with the timeseries UUID. A new Brick relationship, `brick:timeseries`, relates a data source to the metadata about its storage. The subject of `brick:timeseries` is a Brick Point entity; the object is a complex object with several properties:

	API Function Signature
Metadata	`register_stream(uuid, graph, name?, class?, units?)`
	`add_metadata(graph, dataset, [triple])`
	`get_metadata(query, [graph]?) -> table`
	`qualify([query], [graph]?) -> table`
Data	`add_data(uuid, [timestamp, value])`
	`get_data(query, start?, end?, agg?, window?, [graph]?) -> [table]`
	`get_data([uuid], start?, end?, agg?, window?) -> [table]`

Table 7.2: API operations supported on the Mortar platform. A `?` suffix indicates the parameter is optional. A `[parameter]` notation indicates the type is a list.

- `brick:hasTimeseriesId`: a literal-valued property that indicates the primary key identifier for the data stream (in Mortar this is a 36-byte UUID)
- `brick:storedAt`: an optional property that encodes the location of the database containing the data for the data stream. This allows distributed Brick models to always retain a pointer to where related data is stored, and allows Brick points to distribute their storage across multiple databases.

The `brick:timeseries` property is queried automatically by the API frontend when `get_data` is called to determine which data streams must be fetched.

The return value of `get_data` is a set of 3 tables. The first table, *streams*, contains the names, UUIDs, units and enclosing graph names of each of the data streams returned by the query. The second table, *metadata*, contains the results of the SPARQL query. The last table, *data*, contains the timeseries data indexed by timestamp and UUID.

Mortar supports basic access control at the level of graphs. API keys can be created that grant read and/or write access to the metadata and data for a particular graph. The results returned by the Mortar API will have all inaccessible data streams and graphs elided.

Platform Implementation

The Mortar platform is implemented using a collection of existing open-source software and custom-built software. The timeseries and versioned triple store in Figure 7.7 are built on Postgres. The timeseries storage uses TimescaleDB, a timeseries-optimized extension for Postgres. The reasoner is an HTTP server implemented in Rust that wraps `reasonable`. It uses `LISTEN/NOTIFY` to subscribe to changes to the versioned triple store table and propagates those changes to the in-memory reasoner, and eventually to the SPARQL query processor. The API frontend is implemented in Go and incorporates a simple throttle mechanism to limit the number of concurrent connections; this helps limit the ability of a large number of clients to overwhelm the server.

These three services — the database, the reasoner and the API frontend — are all distributed in Docker containers to facilitate deployment. These containers may be hosted directly on a host machine, using a tool like `docker-compose` to manually manage the deployment and scaling of the services, or using a cluster management system like Kubernetes.

Conclusion

This chapter has explored the platform requirements for self-adapting software. The rich semantic metadata supporting self-adapting software requires the use of a reasoner to materialize the inferred elements of the graph. This enables greater flexibility in the queries made by self-adapting software. The integration with a data platform requires the reasoner to be timely and efficient so that materialized RDF graphs can be part of interactive workflows.

`reasonable` is a new OWL 2 RL reasoner optimized to perform reasoning over changing graphs and ontologies and to act as an efficient transformation step between a versioned triple store and a SPARQL query processor. `reasonable` is implemented as a standalone library and demonstrates performance 7-40x times faster than existing OWL 2 RL reasoners.

The Mortar data platform provides timeseries and RDF storage for large amounts of data, organized into graphs and described using Brick metadata. Mortar provides a simple API that allows self-adapting software to discover and access the metadata and data needed to customize its operation. Mortar is available online at `https://beta.mortardata.org`.

Chapter 8

Adoption and Impact

Brick, Mortar and their related technologies detailed in this thesis are influencing data-driven practices, standards and research in academia, industry and government.

8.1 Metadata Standardization and Semantic Interoperability

When Brick was released in 2016 there were few efforts that addressed the need for consistent and interpretable metadata for analytics and controls, and even fewer that were recognized or adopted by the building industry. The dominant practice was for controls and BMS vendors to sell access to telemetry on a point-by-point basis through proprietary APIs and interfaces. Companies like SkyFoundry, the company behind Project Haystack, helped to establish that a distinct analytics platform with first-class support for analytics "apps" and historical telemetry could provide value to building owners and stakeholders. As a result, Project Haystack's terminology and marketing characterized much of the building industry's understanding of "semantic metadata" despite the fact that the underlying product and technology did not address the key data modeling challenges (Chapter 4). Publications, talks, interviews, blog posts and webinars about Brick evinced the shortcomings of informal tag-based metadata and provided an alternative that was built over existing W3C standards. Since 2016, Brick and its ideas have influenced a number of technologies, standards and other ontologies.

ASHRAE 223P

ASHRAE (American Society of Heating, Refrigeration and Air-Conditioning Engineers) is a professional organization behind many standards in the building industry, including BACnet [14]. In 2012, ASHRAE established the Application Profile Working Group (AP-WG) to develop standardized interfaces for interacting with common types of equipment [9]. Accord-

ing to the working group website, the proposed standard would "provide a machine-readable way of representing the capabilities of individual BACnet devices such as services and objects supported [6]." The intended output of the working group was to author annexes to the BACnet standard that would encode this information in the networked representation of each device.

Several years into this effort, it was recognized that standardizing interfaces was only part of the problem: applications needed contextual information in order to find which interfaces of which equipment they needed to operate. In 2018, the goal of the working group was amended to develop a new standard, 223P[1], entitled *Semantic Tags for Building Data*. The focus was still on applications, but rather than specifying the interfaces to the devices in the building, the group would

> "investigate the development of semantic information concepts and vocabularies suitable for building data of various application areas. This will extend the level of semantic information to include application concepts.[7]"

This new direction was intended to incorporate aspects of both the Brick and Project Haystack metadata efforts [8]. At time of writing, a draft of 223P is expected to be released in 2022.

223P will be expressed as a semantic web ontology built using OWL and SHACL. The design is based on the SAREF4SYS ontology, but extends the generic SAREF4SYS concepts with classes which are relevant to the building domain. The core concepts to 223P are Systems, Devices, Connections and Connection Points: this provides an abstract data model which can express more detailed topology and system composition than either Brick or Haystack. Systems are a collection of connected devices or other systems. Devices are tangible objects designed for a specific task, such as a pump or variable frequency drive. Systems can be "connected" to other systems and devices can be "connected" to other devices. Devices may be composed of atomic Parts, but Parts do not have any connections to other Parts or other entities in the model. Unlike Brick, 223P Connections are objects in the model; in Brick, connections are captured through the `brick:feeds` relationship which is a simple RDF predicate. By making Connections objects, it is possible to attach additional properties to them. Example of Connections in 223P include pipes, ducts and wires. Connections connect to devices and systems by means of Connection Points, which are also objects and can also be described with additional properties such as the direction of some medium flowing through the connection.

The relationship between Brick and 223P is promising. The two ontologies differ in their intended use cases and expected data sources. Brick is designed for "brownfield" settings, in which the ground truth of which entities and data sources are *actually* present in a building is not necessarily known or captured in a structured manager. As a result, Brick models are designed to be created from partial or incomplete information. Furthermore, by concentrating on enabling data-driven applications, Brick eschews detailed "white-box" descriptions of building subsystems: this detail is largely unnecessary, hard to come by, and difficult to

[1]The **P** suffix refers to the fact that the standard is considered *proposed* until it is ratified.

maintain. In contrast, 223P is designed to be implemented during the commissioning stage of a building. Parts of a 223P model could be packaged with the equipment and controllers that are installed in the building. A systems integrator would stitch 223P-compliant metadata snippets together into a detailed and complete representation of the building's subsystems.

Brick and 223P are on track to be compatible. Several of 223P's concepts are rooted in concepts imported from Brick; at time of writing, the imported concepts include substances, equipment, points and locations. All elements of 223P's data model can be simplified into a form that Brick can express: the relationships between Systems and Devices can be expressed as compositional and topological relationships in Brick; sources of timeseries data and their properties may be represented in Brick directly. This means that Brick model will be derivable from a 223P model, ensuring forward compatibility.

DOE Semantic Interoperability Project

The U.S. Department of Energy has identified the lack of semantic interoperability between technologies as a fundamental barrier to achieving grid-interactive, energy efficient (GEB) buildings at scale. In [19], researchers from the National Institute of Standards and Technologies (NIST) and several national labs and universities propose a 3-part pathway for enabling semantic interoperability.

The first component is working with industry and existing metadata representation efforts such as Brick and Haystack to address known shortcomings of these existing approaches. As established in [51] and this thesis, Haystack models suffer from significant customization in the field which reduces consistency and interpretability between deployments. Brick addresses these issues, but is built on semantic web technologies that are unfamiliar to many practitioners. [19] proposes leveraging the existing overlap between Brick and Haystack to develop a new standard, ASHRAE 223P, to unify industry and academic metadata efforts.

The second component is to design this new standard in such a way that it can be used by building owners to identify and require interoperable components when procuring equipment, controllers and applications. The key idea is to use the standard not as a post-hoc description of existing cyberphysical resources but as the mechanism through which interoperable cyberphysical systems can be assembled. Products should be bought because they support this new standard, and the standard should provide a consistent way of describing the capabilities and structure of the products. This will ultimately reduce commissioning and system integration costs.

The third component is the creation of standard tools to assist in the implementation of the standard and a testing framework which can verify compliance of products to the standard. This is the approach taken by the BACnet standard [14]. According to [19], it took BACnet more than two decades to achieve its current 64% market share: this was greatly accelerated by the existing of an independent testing group called BACnet Testing Laboratories. A similar approach is proposed for semantic metadata. The testing framework should ensure that standard-compliant products are able to support an array of representative real-world use-cases and applications.

Work on this semantic interoperability vision began in 2020 and will accelerate over the next few years. Software teams ant NREL have begun implementation of ontology-agnostic tools for constructing metadata models of buildings that can be "compiled" to either Haystack or Brick. As part of this effort, a recent review [117] compares and contrasts a variety of metadata schemas and ontologies focusing on energy-oriented applications for buildings.

Haystack v4

Project Haystack has been undertaking a redesign of the metadata model, addressing long-standing issues in the design that have been noted by the community and identified by the work in this thesis [121]. Among the changes is a shift to a Brick-inspired design. Rather than relying entirely on ad-hoc collections of tags for meaning, the new design defines well-known sets of tags which have documented definitions:

> "It has become clear over the last few months that there is no getting around coining an identifier for every single point type.[123]"

This addresses some of the consistency issues incumbent in the existing Haystack design, but does not fully address structured extensibility or unclear subclass relationships — both issues formalized in Chapter 4. Haystack 4 also provides an RDF-compatible export. Note that this does *not* constitute the development of an ontology: the graph export is simply an RDF representation of the existing Haystack data model. As as result, the inference techniques for extracting Brick metadata from a Haystack model — covered in Chapter 6 — are still necessary.

Other Alignments and Ontologies

A number of other ontologies and metadata representations for buildings have been developed since 2016. The Brick ontology has established formal alignments and translators with most of these other representations.

Building Topology Ontology

The Building Topology Ontology (BOT), introduced in [126], is a simple upper ontology which defines core physical concepts of a building's construction. Because the concepts defined in BOT are generic, it is possible to align it with other ontologies such as Brick and IFC [129]. BOT is being developed by the Linked Building Data community group through the W3C organization. Brick maintains a formal alignment with the BOT ontology, meaning that BOT concepts can be derived from Brick models.

RealEstateCore

RealEstateCore (REC), introduced in [63], is a domain ontology developed to support data-oriented real estate applications. There is a high degree of overlap between the concepts

defined in Brick and those defined in REC, and REC does cite Brick as an inspiration for its design of data sources and sinks. However, REC takes the extra steps of modeling the interfaces to different building devices, as well as modeling organizations and individuals and other administrative structures and entities that interact with buildings. Brick maintains a formal bidirectional alignment with the REC ontology, meaning that REC and Brick models can be derived from each other.

Virtual Buildings Information System

The Virtual Buildings Information System [145] is a asset tagging and classification scheme that defines short, hierarchical 4-level names for common types of assets in buildings. VBIS covers equipment for a broad range of building subsystems including HVAC, water, security, lighting, fire suppression and kitchens. VBIS is designed to annotate existing records stored in other databases or files, so it does not define an API (other than requiring string matching). Because VBIS is distributed as a set of spreadsheets, it is not possible for Brick to define a formal mapping between the two metadata representations. Instead, Brick defines a new property for relating a VBIS tag to a Brick entity. The implementation uses SHACL shapes to define regular expressions which constrain the content of the VBIS tag to be appropriate to the Brick class. The implementation can also infer a Brick class for an entity with a given VBIS tag.

Google's Digital Buildings Ontology

Google has also been developing its own metadata model for real estate, equipment and equipment data which takes inspiration from Brick [20]. The ontology, named Digital Buildings, is built on a custom configuration language and custom tooling which can produce an OWL-based ontology as output. Part of the reason for a custom configuration language and tooling is to circumvent some of the modeling issues which arise from the open-world assumption that underlies OWL. Digital Buildings defines Protobuf messages and fields for many different kinds of equipment. In contrast to Brick, Digital Buildings equipment are defined by which inputs and outputs they represent; for example, a VAV with Reheat concept in Digital Buildings would require that the model includes points about a reheat valve in addition to the points for the VAV. The Digital Buildings effort is open-source and maintains compatibility with Brick and Haystack as a long-term goal [59].

8.2 Open Data Research Efforts

While Mortar is not the first open-data research platform for buildings and other cyberphysical data, the use of an expressive ontology to describe the hosted data has inspired and complemented several other open data research efforts.

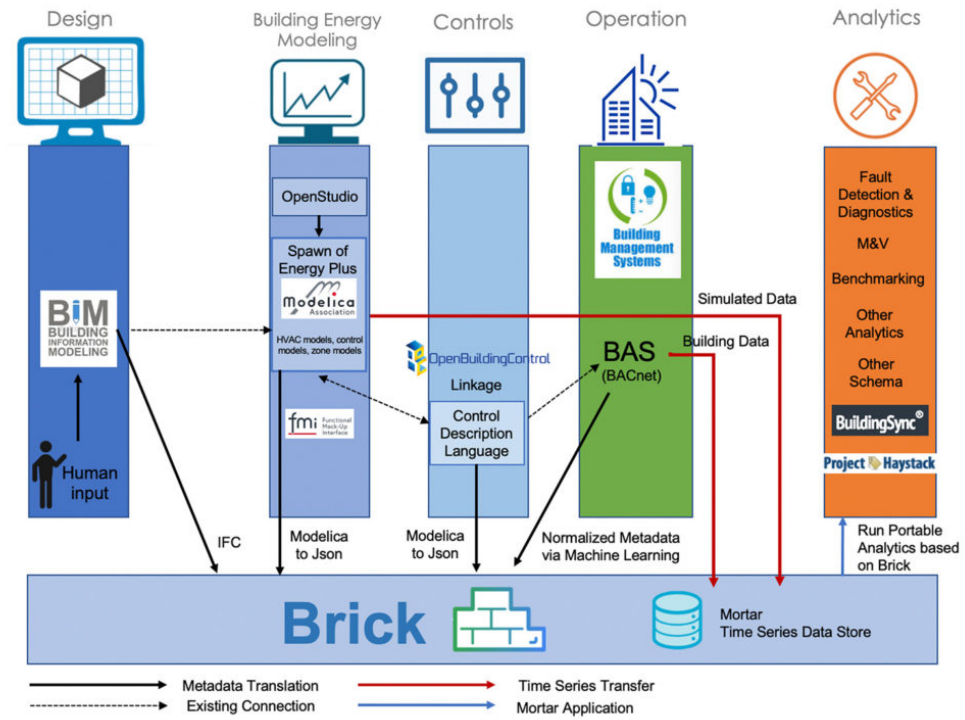


Figure 8.1: Technologies for design, operation, analytics, controls and modeling of buildings are siloed and rarely interoperable

Skewering the Silos

In 2019, DOE accepted a grant proposal for the further development of Brick as a "lingua franca" unifying the silos of technologies used across different data-oriented processes (Figure 8.1). As part of executing the grant, the Mortar platform would be extended and further developed into a production-ready analytics platform containing reference implementations of many common analytics and controls applications. Collaborations with industry and academic partners have resulted in a large array of datasets being donated to the research team. These datasets will be cleaned, anonymized and hosted on a publicly-accessible Mortar instance.

The grant work also involves developing connectors or translators between Brick and other building technologies, including Modelica [150], CDL [149], Haystack [118], gbXML [60] and BuildingSync [88]. The work in this thesis advances the grant work in an open-source manner, and establishes a common platform for continuing that kind of research.

Annex 81: Data-Driven Smart Buildings

Annex 81 is an International Energy Agency project bringing together international experts in data, metadata, ontologies and buildings to establish best-practices [77]. The project is divided into multiple sub-tasks, each of which focuses on a different component of data-driven buildings. Two sub-tasks are directly relevant to the work in this thesis. The "Open Data and Data Platform" subtask will review and report on standards, protocols and procedures for capturing data about a variety of building systems and making that data available for research and applications. This will involve grappling with questions of data governance, security and privacy in addition to the mechanical issues of how to manage that data. The "Applications and Services" subtask will explore the range of applications and use-cases that are enabled through access to high-quality and ubiquitous data about buildings. At time of writing, Brick and Mortar have already been presented to the group and are helping to frame the scope of the project over the next 5 years.

CSIRO Data Clearing House

The Commonwealth Scientific and Industrial Research Organisation (CSIRO) in Australia is undertaking the development of a Smart Building Data Clearing House (referred to as the DCH), a single location for building data at a national scale [3]. The platform will support secure, differentiated access to thousands of buildings across many different sectors of the economy and support data-driven applications providing audits, energy efficiency measures, fault detection and other features. Brick is a key technology to the development of the DCH. Like in Mortar, each building in the DCH will be described with a Brick model which represents the structure of the building and its subsystems and the context of the available data sources. The adoption of Brick in the DCH creates an opportunity for external and independent development efforts to address some of the open usability issues around Brick, such as effective query languages and graph management.

8.3 Brick Consortium

Representatives of UC Berkeley, UC San Diego, Carnegie Mellon University, Johnson Controls Inc. and IBM came together to form an open industrial consortium for the continued development, support and research of Brick [33]. The consortium will publish a Brick Specification that details how products and buildings can become compliant with Brick. This will help evolve Brick to a *de-facto* standard, a stance reinforced by its deep integration with the emerging ASHRAE 223P standard. The consortium is implemented as a non-profit corporation with carefully designed disclosure bylaws that ensure that the Brick specification is unencumbered with proprietary intellectual property.

A Technical Committee publishes new versions of the Brick ontology and specification. Development of the ontology and specification are undertaken by a collection of Working Groups which are open, volunteer-run groups of individuals focused on one of four areas:

- **Ontology Development**: This working group discusses additions, extensions and fixes to the Brick ontology definition. This includes, but is not limited to, new classes, modeling domains, changes to the ontology structure, and documentation for Brick.
- **Tooling**: This working group develops, tests and documents open-source software that facilitates use of Brick. This includes software for producing Brick models from existing representations, as well as software, libraries and databases for querying, managing and storing Brick models.
- **Data and Metadata**: This working group collects, creates and releases Brick models, with the aim of illustrating and documenting idiomatic and correct usage of Brick across a variety of settings. In addition, the group will gather, clean and release real-world building data, contextualized with Brick, which will enable data-driven research.
- **Applications and Analytics**: This working group develops, tests and documents open-source implementations of controls, analytics and other applications for the Brick community to use.

8.4 Growing Adoption

Brick addresses a real problem in the industry: how to consistently describe the increasing amount of data that is being gathered and leveraged in buildings. Because it is built on semantic web technology, Brick integrates with the family of surrounding metadata standards and ontologies in the building space. The ontology captures a critical mass of the concepts that are recognized and required by the industry and has thus attracted developers from across the world who would rather invest in an existing and extensible effort than go through the trouble of creating their own. Brick is unique in that it defines how to use an ontology as a complement to gathered telemetry, rather than awkwardly coercing that telemetry into the same data model. Mortar illustrates how this hybrid data model enables a new approach to authoring and deploying analytics software; this is being studied in Annex 81 and DCH is adopting a similar architecture.

8.5 Availability of Open Source Code

Essentially everything described in this thesis is implemented and available as permissively-licensed, open-source software. Most Brick-related resources can be found through the organization's website: `brickschema.org`. Individual repositories relating to this thesis work are listed below.

- The authoritative Brick implementation, described in Chapter 4: `https://github.com/BrickSchema/Brick`

- `py-brickschema`, a Python library for working with Brick: `https://github.com/BrickSchema/py-brickschema`
- Developer-focused documentation for Brick, hosted online at `https://docs.brickschema.org`
- The metadata integration work described in Chapter 6 is available at `https://github.com/gtfierro/shepherding-metadata`
- Brick Builder and the Reconciliation server are a simple tools for constructing Brick models in an automated manner: `https://github.com/gtfierro/brick-builder`, `https://github.com/BrickSchema/reconciliation-api`
- The HodDB query processor, described in [54], is available at `https://github.com/gtfierro/hoddb`
- The source code for `reasonable` (described in Chapter 7) is hosted at `https://github.com/gtfierro/reasonable`
- The source code for the Mortar platform (described in Chapter 7) is available at `https://github.com/gtfierro/mortar`
- The library of analytics described in Chapter 5 is online at `https://github.com/SoftwareDefinedBuildings/mortar-analytics`
- XBOS and DEGC, large-scale control and monitoring platforms for smart buildings and smart grids, use Brick as part of their implementation. Their source code is available at `https://github.com/gtfierro/xboswave/` and `https://github.com/gtfierro/DEGC`, respectively.

Chapter 9

Conclusion

> I would have written a shorter letter, but I did not have the time.
>
> *Blaise Pascal*

This chapter summarizes the results and contributions of the thesis, proposes future directions of research that build on these results, and reflects on the thesis work.

9.1 Results and Contributions

This thesis proposes a new paradigm — self-adapting software – for programming and interacting with data in cyberphysical environments. Self-adapting software removes the manual and deployment-specific configuration and reimplementation costs that limit the widespread deployment and adoption of data-driven practices. Key to achieving this vision is the insight that *both* the semantic representation of the environment and an effective programming model are necessary. Without a metadata model, cyberphysical software has no representation from which to bootstrap its understanding of an environment and thus configure its operation. To make self-adapting software practical, this thesis examines not just the design of the metadata and programming models, but also the concerns of the systems that support the new paradigm. This thesis offers four contributions, each addressing one of the challenges laid out in Chapter 1.

For descriptions of heterogeneous environments (challenge #1), this thesis presents Brick, a semantic ontology representing a graph-based data model of data sources and their context in smart buildings. Brick is designed to abstract away the irrelevant details of complex cyberphysical systems, in order to present a simplified yet functional view that can be accessed by applications. Crucial to the success of an ontology for cyberphysical systems is the ability to capture entities — including data sources, equipment and other components that operate on the environment — as well as the relationships between those entities. Entities have

semantic definitions, meaning that their behavior and purpose are explicitly represented by the ontology. This thesis observes that a formal definition of such a data model is necessary for that model to be interpretable, consistent while remaining extensible to new kinds of cyberphysical systems. These design principles are implemented in a set of novel ontology features.

This thesis defines two different programming models for leveraging semantic descriptions of the built environment (challenge #2). The first model is a staged execution model appropriate for batch processing of large amounts of data on many different environments. This model is implemented in a Python library and is supported by the Mortar data platform. The second model is an relational algebra-esque model that facilitates the simultaneous and incremental exploration of both metadata and data for an environment. These two models demonstrate two different methodologies for expressing self-adapting software over graph-based metadata. The former uses static queries that project complex graphs to a simpler, application-specific structure that can be programmed against. The latter uses dynamic queries that are built up as the developer explores available metadata and data.

The challenge of managing metadata over time (challenge #3) is addressed through the development of a protocol for gathering inferred semantic metadata from a variety of sources and an algorithm for merging those models together into a semantically and formally valid whole. This contribution establishes a common architecture for bringing current and future metadata inference research together into a usable system. Importantly, this approach decouples the method of inference from how the produced metadata is incorporated into applications. The algorithm and protocol have been successfully implemented for a variety of building metadata standards and representations.

Lastly, to store, manage and serve the metadata and data that enable data-driven self-adapting applications (challenge #4), this thesis presents the design and implementation of Mortar and a supporting system, `reasonable`. `reasonable` is a software library which provides efficient and timely reasoning for formal linked data models in a package that is easy to integrate into existing databases and architectures. Mortar is a data platform incorporating a relational database for timeseries data that also supports a linked data workload, facilitated by `reasonable`. A public instance of Mortar, hosted on `mortardata.org`, contains data and metadata for over 100 real-world buildings that is available for research. These systems demonstrate the existence of an "impedance mismatch" between the capabilities of existing data systems and what is required to support self-adapting software, and identify the necessary features that can bridge that gap.

9.2 Future Work

The work presented in this thesis lays the foundation for future work into semantic metadata for the built environment and novel programming models and data systems that support that metadata. Below, several concrete directions of future work are identified.

Semantic Overlay for the Built Environment: Buildings, and by extension the Brick ontology, are just one piece of a larger picture. Other sectors of the built environment such as power generation and water treatment and management also possess complex cyberphysical infrastructure that is increasingly monitored but remains "information-poor". There is a need for semantic metadata models which lower the cost of applying data to the resilient and efficient operation of smart grids, distributed energy resources, water treatment plants and other critical systems. The ontology design patterns developed in this thesis offer a concrete basis for developing and maintaining these models. Ultimately, the existence of structured and formal digital representations of these systems could be combined into a unified semantic "overlay" for the built environment. This would allow data from different sectors to complement and inform the maintenance and operation of other sectors, promoting a more resilient and co-optimized existence.

Robust Programming Models for Linked Data: The programming models explored in this thesis are merely a starting point for a more detailed look at how software can become self-customizing. The semantics for how software reacts to the changing content of the metadata model must be established. There is an opportunity to pull techniques from related domains such as software-defined networking which can ensure that software configuration remains correct and valid. Moreover, it is not a foregone conclusion that the self-adapting programming models remain so closely tied to the expression of the semantic metadata. Future research is required to determine intuitive and effective programming models that place less of a burden on the programmer to be intimately familiar with the formal and linked nature of the metadata underlying the software.

Handling Semantic Evolution: The question of how to handle evolution of data semantics, outlined at the end of Chapter 6, revealed itself to be a much more complex task than at first glance. Although techniques from data integration and schema evolution literature offer promising approaches to how to handle this kind of evolution, these solutions do not take into account the fact that different consumers of data may care about different aspects of the evolved data. This means that there is no application-agnostic "correct answer" for how to properly rewrite queries to execute on older or newer semantics. Instead, the correct transformation is dependent upon the semantics of the application, which is not currently captured in any representation. Future work will need to determine how to capture the semantics of applications so that the proper rewriting of their queries can be performed.

Distributed Management of Semantic Metadata: Current research on linked data databases, including this thesis, primarily focuses on how to manage and query larger and larger graphs across increasing numbers of machines. However, if linked data is to enable self-adapting data-driven software across many kinds of cyberphysical systems, then linked databases must *scale down* as well as scale up. It is unrealistic to rely on large, cloud-based storage clusters for the storage and performance capabilities required. Remote, administratively centralized databases that require an internet connection to access are anathema to the resilient and independently operated cyberphysical systems that compose the future built environment. Future research should investigate the design of linked data databases that can provide necessary features and decent performance on single-node deployments.

Further, it is important to investigate how modern database technologies and techniques may be applied to linked data management in order to improve performance and adoption.

9.3 Reflections and Final Remarks

This thesis comes almost 10 years after the idea of portable applications for buildings was first published [84]. In that time, the landscape has changed significantly. "Big data" has evolved from being the charge of a few large tech companies to being ubiquitous and characteristic of the larger information technology industry. Data science technologies have matured and been reinvented several times over. Software defined networking has demonstrated how a rethinking of information organization can dramatically simplify the task of managing large-scale systems. Many of the pieces for extracting value out of cyberphysical data exist in other domains, and yet the application of these ideas and technologies to critical cyberphysical infrastructure remains underdeveloped. This thesis presents a vision of how this gap can be bridged.

It has been clear since the beginning of this journey that graph-based metadata is a crucial component of the solution. The effective expression and structure of that graph are less obvious. Ad-hoc graphs with no regulation to their definition do not sufficiently generalize and are not necessarily interpretable by those who did not develop the graph. The semantic web and its technologies offer a cure for inconsistency; however, these communities are more traditionally more concerned with the philosophical consequences of the models they develop than the practical implications and barriers to usage. A significant number of the ontologies and other formal data models developed for IoT and the built environment experience little to no adoption. Semantic web technology is often rejected on the grounds of being overly "academic" and impractically complex for simple tasks.

Why then has Brick experienced adoption? Brick has succeeded because it models a novel abstraction of the built environment that is closer to the needs of the application than it is to traditional ontology modeling techniques. This use-case driven design philosophy is crucial to generalizing Brick-like concepts to new models for other sectors of the built environment. Complex systems are simpler from an application's perspective. By clearly defining and modeling the application's perspective, complex systems can be captured with a simpler and more usable ontology.

Standardized metadata representations of cyberphysical environments enable a fundamentally richer set of applications that are easier to write, deploy and measure. Self-adapting software is one view of how programming and data systems can evolve to meet the demands of this new domain. It is my hope that this thesis inspires and informs future work into how data can be better applied to sustainable practices.

Bibliography

[1] *Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries*. Standard. Geneva, CH: International Organization for Standardization, Nov. 2018.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Vol. 8. Addison-Wesley Reading, 1995.

[3] Australian Renewable Energy Agency. *Smart Building Data Clearing House*. 2021. URL: http://web.archive.org/web/20210313091107/http://www.ihub.org.au/ihub-initiatives/smart-building-data-clearing-house/.

[4] Daniel Aguado et al. "Digital Water: The value of meta-data for water resource recovery facilities". In: *IWA Digital Water Programme* (2021).

[5] Janelcy Alferes et al. "Advanced monitoring of water systems using in situ measurement stations: data validation and fault detection". In: *Water science and technology* 68.5 (2013), pp. 1022–1030.

[6] American Society of Heating, Refrigerating and Air-Conditioning Engineers. *AP Working Group*. http://web.archive.org/web/20120806111158/http://www.bacnet.org/WG/AP/index.html. 2012.

[7] American Society of Heating, Refrigerating and Air-Conditioning Engineers. *AP Working Group*. http://web.archive.org/web/20190718105003/http://www.bacnet.org/WG/AP/index.html. 2019.

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

[9] American Society of Heating, Refrigerating and Air-Conditioning Engineers. *BACnet Application Interface Development: Call for Participants!* http://web.archive.org/web/20210403220822/http://www.bacnet.org/Oldnews/Oldnews-12.htm. 2012.

[10] Robert Arp and Barry Smith. "Function, role, and disposition in basic formal ontology". In: *Nature Precedings* (2008), pp. 1–1.

[11] Inc Autodesk. *Revit: Multidisciplinary BIM software for higher-quality, coordinated designs*. 2021. URL: `https://www.autodesk.com/products/revit/overview` (visited on 05/11/2021).

[12] Salman Azhar. "Building information modeling (BIM): Trends, benefits, risks, and challenges for the AEC industry". In: *Leadership and management in engineering* 11.3 (2011), pp. 241–252.

[13] Michael Bächle and Paul Kirchberg. "Ruby on rails". In: *IEEE software* 24.6 (2007), pp. 105–108.

[14] *BACnet: A Data Communciation Protocol for Building Automation and Control Networks*. ASHRAE Standard 135-2016. Atlanta, Ga: ASHRAE, 2016.

[15] Bharathan Balaji et al. "Brick : Metadata schema for portable smart building applications". In: *Applied Energy* 226 (2018), pp. 1273–1292. ISSN: 0306-2619. DOI: `https://doi.org/10.1016/j.apenergy.2018.02.091`. URL: `http://www.sciencedirect.com/science/article/pii/S0306261918302162`.

[16] Bharathan Balaji et al. "Brick: Towards a unified metadata schema for buildings". In: *Proceedings of the ACM International Conference on Embedded Systems for Energy-Efficient Built Environments (BuildSys). ACM*. 2016.

[17] Dave Beckett and Brian McBride. "RDF/XML syntax specification (revised)". In: *W3C recommendation* 10.2.3 (2004).

[18] David Beckett et al. "RDF 1.1 Turtle". In: *World Wide Web Consortium* (2014), pp. 18–31.

[19] Harry Bergmann et al. "Semantic Interoperability to Enable Smart, Grid-Interactive Efficient Buildings". In: *ACEEE* (2020).

[20] Keith Berkoben, Charbel Kaed, and Trevor Sodorff. "A Digital Buildings Ontology for Google's Real Estate". In: *International Semantic Web Conference (ISWC)* (2020).

[21] Philip A Bernstein and Sergey Melnik. "Model management 2.0: manipulating richer mappings". In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 2007, pp. 1–12.

[22] Philip A Bernstein, Sergey Melnik, and John E Churchill. "Incremental schema matching". In: *VLDB*. Vol. 6. Citeseer. 2006, pp. 1167–1170.

[23] Philip A Bernstein et al. "Implementing mapping composition". In: *The VLDB Journal* 17.2 (2008), pp. 333–353.

[24] Anant Bhardwaj et al. "Datahub: Collaborative data science & dataset version management at scale". In: *arXiv preprint arXiv:1409.0798* (2014).

[25] Arka Bhattacharya. "Enabling Scalable Smart-Building Analytics". PhD thesis. EECS Department, University of California, Berkeley, 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-201.html.

[26] Arka Bhattacharya, Joern Ploennigs, and David Culler. "Short Paper: Analyzing Metadata Schemas for Buildings: The Good, the Bad, and the Ugly". In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 33–34.

[27] Arka A. Bhattacharya et al. "Automated Metadata Construction to Support Portable Building Applications". In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments - BuildSys '15* (2015), pp. 3–12. DOI: 10.1145/2821650.2821667. URL: http://dl.acm.org/citation.cfm?doid=2821650.2821667.

[28] George EP Box. "Robustness in the strategy of scientific model building". In: *Robustness in statistics*. Elsevier, 1979, pp. 201–236.

[29] Michael R Brambley et al. *Advanced sensors and controls for building applications: Market assessment and potential R&D pathways*. Tech. rep. Pacific Northwest National Lab.(PNNL), Richland, WA (United States), 2005.

[30] Diego Calvanese et al. "The MASTRO system for ontology-based data access". In: *Semantic Web* 2.1 (2011), pp. 43–53.

[31] Stuart Cheshire and Marc Krochmal. *Multicast DNS*. RFC 6762. 2013. DOI: 10.17487/RFC6762. URL: https://rfc-editor.org/rfc/rfc6762.txt.

[32] Anthony Cleve and Jean-Luc Hainaut. "Co-transformations in database applications evolution". In: *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer. 2005, pp. 409–421.

[33] Brick Consortium. *The Brick Consortium*. 2021. URL: http://web.archive.org/web/20210408005045if_/https://brickschema.org/consortium/.

[34] Lluís Corominas et al. "Performance evaluation of fault detection methods for wastewater treatment processes". In: *Biotechnology and bioengineering* 108.2 (2011), pp. 333–344.

[35] Carlo A Curino, Hyun J Moon, and Carlo Zaniolo. "Graceful database schema evolution: the prism workbench". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 761–772.

[36] Steven E Czerwinski et al. "An architecture for a secure service discovery service". In: *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. 1999, pp. 24–35.

[37] Laura Daniele, Frank den Hartog, and Jasper Roes. "Created in close interaction with the industry: the smart appliances reference (SAREF) ontology". In: *International Workshop Formal Ontologies Meet Industries*. Springer. 2015, pp. 100–112.

[38] Anish Das Sarma, Xin Dong, and Alon Halevy. "Bootstrapping pay-as-you-go data integration systems". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, pp. 861–874.

[39] Stephen Dawson-Haggerty et al. "{BOSS}: Building operating system services". In: *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 2013, pp. 443–457.

[40] Stephen Dawson-Haggerty et al. "sMAP: a simple measurement and actuation profile for physical information". In: *Proceedings of the 8th ACM conference on embedded networked sensor systems*. 2010, pp. 197–210.

[41] Suhrid Deshmukh, Leon Glicksman, and Leslie Norford. "Case study results: fault detection in air-handling units in buildings". In: *Advances in Building Energy Research* 0.0 (2018), pp. 1–17. DOI: 10.1080/17512549.2018.1545143. eprint: https://doi.org/10.1080/17512549.2018.1545143. URL: https://doi.org/10.1080/17512549.2018.1545143.

[42] Django Community. *Django Python Package*. https://www.djangoproject.com/. Accessed: February 12, 2020. 2020.

[43] Inc. DoltHub. *DoltHub*. 2021. URL: http://web.archive.org/web/20210311022618/https://dolthub.com/.

[44] Bing Dong et al. "A comparative study of the IFC and gbXML informational infrastructures for data exchange in computational design support environments". In: *IBPSA 2007 - International Building Performance Simulation Association 2007* 3 (Jan. 2007), pp. 1530–1537.

[45] Xin Dong, Alon Halevy, and Jayant Madhavan. "Reference reconciliation in complex information spaces". In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 2005, pp. 85–96.

[46] Xin Luna Dong, Alon Halevy, and Cong Yu. "Data integration with uncertainty". In: *The VLDB Journal* 18.2 (2009), pp. 469–500.

[47] Martin Duerst and Michel Suignard. *Internationalized Resource Identifiers (IRIs)*. RFC 3987. Jan. 2005. DOI: 10.17487/RFC3987. URL: https://rfc-editor.org/rfc/rfc3987.txt.

[48] Ronald Fagin et al. "Clio: Schema mapping creation and data exchange". In: *Conceptual modeling: foundations and applications*. Springer, 2009, pp. 198–236.

[49] Karim Farghaly et al. "Taxonomy for BIM and asset management semantic interoperability". In: *Journal of Management in Engineering* 34.4 (2018), p. 04018012.

[50] Gabe Fierro and David E Culler. "Design and analysis of a query processor for brick". In: *ACM Transactions on Sensor Networks (TOSN)* 14.3-4 (2018), pp. 1–25.

[51] Gabe Fierro et al. "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*. 2019, pp. 125–134.

[52] Gabe Fierro et al. "Mortar: an open testbed for portable building analytics". In: *Proceedings of the 5th Conference on Systems for Built Environments*. ACM. 2018, pp. 172–181.

[53] Gabe Fierro et al. "Mortar: an open testbed for portable building analytics". In: *ACM Transactions on Sensor Networks (TOSN)* 16.1 (2019), pp. 1–31.

[54] Gabriel Fierro. "Design of an Effective Ontology and Query Processor Enabling Portable Building Applications". MA thesis. EECS Department, University of California, Berkeley, 2019. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-106.html`.

[55] Inc Franz. *AllegroGraph: Semantic Graph Database*. 2017. URL: `https://allegrograph.com/allegrograph/`.

[56] Hector Garcia-Molina et al. "The TSIMMIS approach to mediation: Data models and languages". In: *Journal of intelligent information systems* 8.2 (1997), pp. 117–132.

[57] NIST GCR. *Cost analysis of inadequate interoperability in the US capital facilities industry*. Tech. rep. National Institute of Standards and Technology (NIST), 2004, pp. 223–253.

[58] Birte Glimm et al. "HermiT: an OWL 2 reasoner". In: *Journal of Automated Reasoning* 53.3 (2014), pp. 245–269.

[59] Google. *Google Digital Buildings*. 2020. URL: `http://web.archive.org/web/20200919075346/https://github.com/google/digitalbuildings`.

[60] Inc Green Building XML Schema. *Green Building XML*. 2021. URL: `https://www.gbxml.org` (visited on 01/18/2021).

[61] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. "Macro-programming wireless sensor networks using kairos". In: *International Conference on Distributed Computing in Sensor Systems*. Springer. 2005, pp. 126–140.

[62] Armin Haller et al. "The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation". In: *Semantic Web* 10.1 (2019), pp. 9–32.

[63] Karl Hammar et al. "The realestatecore ontology". In: *International Semantic Web Conference*. Springer. 2019, pp. 130–145.

[64] Dave Hardin et al. *Buildings interoperability landscape*. Tech. rep. Pacific Northwest National Lab.(PNNL), Richland, WA (United States), 2015.

[65] Paul Harmon. "Object-oriented AI: a commercial perspective". In: *Communications of the ACM* 38.11 (1995), pp. 80–86.

[66] Steve Harris and Andy Seaborne. "SPARQL 1.1 Query Language". In: *W3C recommendation* (2013).

[67] Andreas Harth and Stefan Decker. "Optimized index structures for querying rdf from the web". In: *Web Congress, 2005. LA-WEB 2005. Third Latin American*. IEEE. 2005, 10–pp.

[68] Bob Hayes. *How do Data Professionals Spend their Time on Data Science Projects?* 2021. URL: `http://web.archive.org/web/20210424231945/https://businessoverbroadway.com/2019/02/19/how-do-data-professionals-spend-their-time-on-data-science-projects/` (visited on 02/19/2019).

[69] Fang He et al. "EnergonQL: A Building Independent Acquisitional Query Language for Portable Building Analytics". In: *Proceedings of the 7th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*. 2020, pp. 266–269.

[70] Joseph M Hellerstein et al. "Ground: A Data Context Service." In: *CIDR*. Citeseer. 2017.

[71] Cory Andrew Henson et al. "An ontological representation of time series observations on the semantic sensor web". In: (2009).

[72] Heinrich Herre. "General Formal Ontology (GFO): A foundational ontology for conceptual modelling". In: *Theory and applications of ontology: computer applications*. Springer, 2010, pp. 297–345.

[73] *High-Performance Sequences of Operations for HVAC Systems*. ASHRAE Guideline 36. Atlanta, Ga: ASHRAE, 2018.

[74] Pascal Hitzler et al. "OWL 2 web ontology language primer". In: *W3C recommendation* 27.1 (2009), p. 123.

[75] Ralph Hodgson et al. *QUDT-quantities, units, dimensions and data types ontologies*. URL: `http://web.archive.org/web/20201023201424/http://www.qudt.org/` (visited on 03/01/2021).

[76] William H Inmon. *Building the data warehouse*. John wiley & sons, 2005.

[77] International Energy Agency. *IEA EBC - Annex 81 - Data-Driven Smart Buildings*. 2021.

[78] Huang Jiayi, Jiang Chuanwen, and Xu Rong. "A review on distributed energy resources and MicroGrid". In: *Renewable and Sustainable Energy Reviews* 12.9 (2008), pp. 2472–2483.

[79] Srinivas Katipamula et al. "VOLTTRON: An open-source software platform of the future". In: *IEEE Electrification Magazine* 4.4 (2016), pp. 15–22.

[80] *SHACL Advanced Features*. Tech. rep. W3C, June 2017. URL: `https://www.w3.org/TR/shacl-af/`.

[81] *Shapes constraint language (SHACL)*. Tech. rep. W3C, July 2017. URL: https://www.w3.org/TR/shacl/.

[82] Jason Koh et al. "Scrabble: transferrable semi-automated semantic metadata normalization using intermediate representation". In: *Proceedings of the 5th Conference on Systems for Built Environments*. ACM. 2018, pp. 11–20.

[83] Matija König, Jaka Dirnbek, and Vlado Stankovski. "Architecture of an open knowledge base for sustainable buildings based on Linked Data technologies". In: *Automation in Construction* 35 (2013), pp. 542–550. ISSN: 0926-5805. DOI: https://doi.org/10.1016/j.autcon.2013.07.002. URL: http://www.sciencedirect.com/science/article/pii/S0926580513001118.

[84] Andrew Krioukov et al. "Building application stack (BAS)". In: *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. 2012, pp. 72–79.

[85] Henrik Lange, Aslak Johansen, and Mikkel Baun Kjærgaard. "Evaluation of the opportunities and limitations of using IFC models as source of building metadata". In: *Proceedings of the 5th Conference on Systems for Built Environments*. 2018, pp. 21–24.

[86] Inc. Leidos. *Trends in Commercial Whole-Building Sensors and Controls*. Tech. rep. U.S. Energy Information Administration, 2020.

[87] Maurizio Lenzerini. "Ontology-based data management". In: *Proceedings of the 20th ACM international conference on Information and knowledge management*. 2011, pp. 5–6.

[88] Nicholas Long et al. *BuildingSync®*. Tech. rep. National Renewable Energy Lab.(NREL), Golden, CO (United States), 2018.

[89] Samuel R Madden et al. "TinyDB: An acquisitional query processing system for sensor networks". In: *ACM Transactions on database systems (TODS)* 30.1 (2005), pp. 122–173.

[90] Kamesh Madduri and Kesheng Wu. "Massive-scale RDF processing using compressed bitmap indexes". In: *International Conference on Scientific and Statistical Database Management*. Springer. 2011, pp. 470–479.

[91] Adam Mathes. *Folksonomies - Cooperative Classification and Communication Through Shared Metadata*. 2021. URL: http://web.archive.org/web/20210128003104/https://adammathes.com/academic/computer-mediated-communication/folksonomies.html (visited on 01/27/2021).

[92] Johanna L Mathieu et al. "Quantifying changes in building electricity use, with application to demand response". In: *IEEE Transactions on Smart Grid* 2.3 (2011), pp. 507–518.

[93] Frank McSherry. *A relatively simple Datalog engine in Rust*. 2021. URL: `http://web.archive.org/web/20201109012007/https://github.com/frankmcsherry/blog/blob/master/posts/2018-05-19.md` (visited on 05/19/2018).

[94] Hui Miao, Amit Chavan, and Amol Deshpande. "Provdb: Lifecycle management of collaborative analysis workflows". In: *Proceedings of the 2nd Workshop on Human-in-the-Loop Data Analytics*. 2017, pp. 1–6.

[95] Renée J Miller et al. "The Clio project: managing heterogeneity". In: *ACM Sigmod Record* 30.1 (2001), pp. 78–83.

[96] Modbus Organization. *Modbus*. URL: `http://web.archive.org/web/20201226111707/https://modbus.org/` (visited on 12/26/2020).

[97] Modelica Association. *Modelica Language*. 2021. URL: `http://web.archive.org/web/20201030043752/https://www.modelica.org/modelicalanguage` (visited on 01/19/2021).

[98] *Modelica to JSON parser*. 2021. URL: `http://web.archive.org/web/20210119222351/https://github.com/lbl-srg/modelica-json` (visited on 01/19/2021).

[99] Fiona Moore, David Churcher, and Sarah Davidson. *BIM Interoperability Expert Group Report*. Report. Center for Digital Built Britain, Mar. 2020.

[100] Boris Motik et al. "OWL 2 web ontology language profiles". In: *W3C recommendation* 27 (2009), p. 61.

[101] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. "Time series databases and influxdb". In: *Studienarbeit, Université Libre de Bruxelles* (2017), p. 12.

[102] National Science Foundation. *Survey of Earned Doctorates*. `https://www.nsf.gov/statistics/srvydoctorates`. 2020.

[103] Yavor Nenov et al. "RDFox: A highly-scalable RDF store". In: *International Semantic Web Conference*. Springer. 2015, pp. 3–20.

[104] Holger Neuhaus and Michael Compton. "The semantic sensor network ontology". In: *AGILE workshop on challenges in geospatial data harmonisation, Hannover, Germany*. 2009, pp. 1–33.

[105] Thomas Neumann and Gerhard Weikum. "RDF-3X: a RISC-style engine for RDF". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 647–659.

[106] Ryan Newton, Greg Morrisett, and Matt Welsh. "The regiment macroprogramming system". In: *2007 6th International Symposium on Information Processing in Sensor Networks*. IEEE. 2007, pp. 489–498.

[107] Natalya F Noy and Michel Klein. "Ontology evolution: Not the same as schema evolution". In: *Knowledge and information systems* 6.4 (2004), pp. 428–440.

[108] Natalya F Noy and Mark A Musen. "The PROMPT suite: interactive tools for ontology merging and mapping". In: *International journal of human-computer studies* 59.6 (2003), pp. 983–1024.

[109] Natalya F Noy et al. "A framework for ontology evolution in collaborative environments". In: *International semantic web conference*. Springer. 2006, pp. 544–558.

[110] Natalya Fridman Noy et al. "Protégé-2000: an open-source ontology-development and knowledge-acquisition environment." In: *AMIA... Annual Symposium proceedings. AMIA Symposium*. Vol. 2003. American Medical Informatics Association. 2003, pp. 953–953.

[111] Frauke Oldewurtel et al. "Use of model predictive control and weather forecasts for energy efficient building climate control". In: *Energy and Buildings* 45 (2012), pp. 15–27.

[112] OSTI. *The National Opportunity for Interoperability and its Benefits for a Reliable, Robust, and Future Grid Realized Through Buildings*. Tech. rep. Feb. 2016. DOI: 10.2172/1420233. URL: https://doi.org/10.2172/1420233.

[113] Matthijs van Otterdijk, Gavin Mendel-Gleason, and Kevin Feeney. "Succinct Data Structures and Delta Encoding for Modern Databases". In: (2020).

[114] Viorica Pătrăucean et al. "State of research in automatic as-built modelling". In: *Advanced Engineering Informatics* 29.2 (2015). Infrastructure Computer Vision, pp. 162–171. ISSN: 1474-0346. DOI: https://doi.org/10.1016/j.aei.2015.01.001. URL: http://www.sciencedirect.com/science/article/pii/S1474034615000026.

[115] Pieter Pauwels and Walter Terkaj. "EXPRESS to OWL for construction industry: Towards a recommendable and usable ifcOWL ontology". In: *Automation in Construction* 63 (2016), pp. 100–133.

[116] Maria Poveda-Villalón and Raúl García-Castro. "Extending the SAREF ontology for building devices and topology". In: *Proceedings of the 6th Linked Data in Architecture and Construction Workshop (LDAC 2018), Vol. CEUR-WS*. Vol. 2159. 2018, pp. 16–23.

[117] Marco Pritoni et al. "Metadata Schemas and Ontologies for Building Energy Applications: A Critical Review and Use Case Analysis". In: *Energies* 14.7 (Apr. 2021), p. 2024. ISSN: 1996-1073. DOI: 10.3390/en14072024. URL: http://dx.doi.org/10.3390/en14072024.

[118] *Project Haystack*. 2021. URL: http://web.archive.org/web/20210111211811/https://project-haystack.org/ (visited on 01/11/2021).

[119] Project Haystack. *Project Haystack Documentation: Defs*. 2019. URL: %5Curl%7Bhttp://web.archive.org/web/20190629183024/https://project-haystack.dev/doc/docHaystack/Defs%7D (visited on 06/29/2019).

[120] Project Haystack. *Project Haystack Documentation: VFDs*. 2019. URL: %5Curl%7Bhttp://web.archive.org/web/20190629182856/https://project-haystack.org/doc/VFDs%7D (visited on 06/29/2019).

[121] *Project Haystack v4*. 2021. URL: http://web.archive.org/web/20210119225945/https://project-haystack.dev/ (visited on 01/19/2021).

[122] *Project Haystack: Air Tag Definition*. 2021. URL: https://web.archive.org/web/20210207212635/https://project-haystack.org/tag/air (visited on 02/07/2021).

[123] *Proposal: Exploding points types as new first class defs*. 2021. URL: http://web.archive.org/web/20210408051839/https://project-haystack.org/forum/topic/889 (visited on 03/03/2021).

[124] Paul Raftery et al. "Evaluation of a cost-responsive supply air temperature reset strategy in an office building". In: *Energy and Buildings* 158 (2018), pp. 356–370. ISSN: 0378-7788. DOI: https://doi.org/10.1016/j.enbuild.2017.10.017. URL: http://www.sciencedirect.com/science/article/pii/S0378778817310939.

[125] Thillaigovindan Ramesh, Ravi Prakash, and KK Shukla. "Life cycle energy analysis of buildings: An overview". In: *Energy and buildings* 42.10 (2010), pp. 1592–1600.

[126] Mads Holten Rasmussen et al. "Proposing a central AEC ontology that allows for domain specific extensions". In: *Joint Conference on Computing in Construction*. Vol. 1. 2017, pp. 237–244.

[127] Jeffrey Schein et al. "A rule-based fault detection method for air handling units". In: *Energy and Buildings* 38.12 (2006), pp. 1485–1492.

[128] Jeffrey Schein et al. "A rule-based fault detection method for air handling units". In: *Energy and Buildings* 38.12 (2006), pp. 1485–1492. ISSN: 0378-7788. DOI: https://doi.org/10.1016/j.enbuild.2006.04.014. URL: http://www.sciencedirect.com/science/article/pii/S0378778806001034.

[129] Georg Ferdinand Schneider. "Towards aligning domain ontologies with the building topology ontology". In: *Proceedings of the 5th Linked Data in Architecture and Construction Workshop (LDAC 2017)*. 2017.

[130] P Griffiths Selinger et al. "Access path selection in a relational database management system". In: *Readings in Artificial Intelligence and Databases*. Elsevier, 1989, pp. 511–522.

[131] Zixiao Shi et al. "Evaluation of clustering and time series features for point type inference in smart building retrofit". In: *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*. 2019, pp. 111–120.

[132] Pierluigi Siano. "Demand response and smart grids—A survey". In: *Renewable and sustainable energy reviews* 30 (2014), pp. 461–478.

[133] John F Sowa. "Ontology, metadata, and semiotics". In: *International conference on conceptual structures*. Springer. 2000, pp. 55–81.

[134] Manu Sporny et al. "JSON-LD 1.0". In: *W3C recommendation* 16 (2014), p. 41.

[135] J-Ph Steyer, A Genovesi, and Jérôme Harmand. "Advanced monitoring and control of anaerobic wastewater treatment plants: fault detection and isolation". In: *Water science and technology* 43.7 (2001), pp. 183–190.

[136] Ljiljana Stojanovic et al. "User-driven ontology evolution management". In: *International Conference on Knowledge Engineering and Knowledge Management*. Springer. 2002, pp. 285–300.

[137] David Sturzenegger et al. "Semi-automated modular modeling of buildings for model predictive control". In: ACM. 2012, pp. 99–106.

[138] SYSTAP, LLC. *Bigdata Database Architecture Whitepaper*. `https://www.blazegraph.com/whitepapers/bigdata_architecture_whitepaper.pdf`. 2017.

[139] Shu Tang et al. "BIM assisted Building Automation System information exchange using BACnet and IFC". In: *Automation in Construction* 110 (2020), p. 103049.

[140] The Apache Software Foundation. "A free and open source Java framework for building Semantic Web and Linked Data applications". In: *https://jena.apache.org/* (2017).

[141] Jacopo Urbani, Ceriel Jacobs, and Markus Krötzsch. "Column-oriented datalog materialization for large knowledge graphs". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 30. 1. 2016.

[142] Jacopo Urbani et al. "OWL reasoning with WebPIE: calculating the closure of 100 billion triples". In: *Extended Semantic Web Conference*. Springer. 2010, pp. 213–227.

[143] Renaud Vanlande, Christophe Nicolle, and Christophe Cruz. "IFC and building lifecycle management". In: *Automation in construction* 18.1 (2008), pp. 70–78.

[144] Panos Vassiliadis. "A survey of extract–transform–load technology". In: *International Journal of Data Warehousing and Mining (IJDWM)* 5.3 (2009), pp. 1–27.

[145] Virtual Buildings Information System. *Virtual Buildings Information System*. 2021. URL: `http://web.archive.org/web/20210407004118/https://vbis.com.au/` (visited on 04/06/2021).

[146] Chen Wang et al. "Apache IoTDB: time-series database for internet of things". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 2901–2904.

[147] 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*. 2013, pp. 1–8.

[148] Samuel R West et al. "Automated fault detection and diagnosis of HVAC subsystems using statistical machine learning". In: *12th International Conference of the International Building Performance Simulation Association*. 2011.

[149] Michael Wetter, Milica Grahovac, and Jianjun Hu. "Control description language". In: *Proceedings of The American Modelica Conference 2018, October 9-10, Somberg Conference Center, Cambridge MA, USA*. 154. Linköping University Electronic Press. 2019, pp. 17–26.

[150] Michael Wetter et al. "Modelica Buildings library". In: *Journal of Building Performance Simulation* 7.4 (2014), pp. 253–270. DOI: 10.1080/19401493.2013.765506. eprint: https://doi.org/10.1080/19401493.2013.765506. URL: https://doi.org/10.1080/19401493.2013.765506.

[151] Michael Wetter et al. "Openbuildingcontrol: Modeling feedback control as a step towards formal design, specification, deployment and verification of building control sequences." In: *Proceedings of Building Performance Modeling Conference and SimBuild co-organized by ASHRAE and IBPSA-USA, , Chicago IL, USA*. 2018.

[152] Gio Wiederhold and Michael Genesereth. "The conceptual basis for mediation services". In: *IEEE Expert* 12.5 (1997), pp. 38–47.

[153] William E. Winkler. *The State of Record Linkage and Current Research Problems*. Tech. rep. Statistical Research Division, U.S. Census Bureau, 1999.

[154] World Wide Web Consortium. *OWL 2 Web Ontology Language*. 2021. URL: http://web.archive.org/web/20210301164820/http://www.w3.org/TR/owl-overview/ (visited on 04/12/2021).

[155] World Wide Web Consortium. *Web of Things*. 2021. URL: http://web.archive.org/web/20210120092058/http://www.w3.org/WoT/ (visited on 01/20/2021).

[156] Guohui Xiao et al. "Ontology-based data access: A survey". In: International Joint Conferences on Artificial Intelligence. 2018.

[157] Liqi Xu et al. "Orpheusdb: a lightweight approach to relational dataset versioning". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 1655–1658.

[158] Q.Z. Yang and Y. Zhang. "Semantic interoperability in building design: Methods and tools". In: *Computer-Aided Design* 38.10 (2006), pp. 1099–1112. ISSN: 0010-4485. DOI: https://doi.org/10.1016/j.cad.2006.06.003. URL: http://www.sciencedirect.com/science/article/pii/S0010448506001011.

[159] The City Of New York. *New York Local Law 87*. 2020. URL: http://web.archive.org/web/20200531233953/https://www1.nyc.gov/html/gbee/html/plan/ll87.shtml.

[160] Deze Zeng, Song Guo, and Zixue Cheng. "The web of things: A survey". In: *JCM* 6.6 (2011), pp. 424–438.

[161] Zigbee Alliance. *Connected Home Over IP*. URL: http://web.archive.org/web/20210115065718/https://www.connectedhomeip.com/ (visited on 01/14/2021).

Appendix

```
@prefix brick: <https://brickschema.org/schema/1.1/Brick#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix tag: <https://brickschema.org/schema/1.1/BrickTag#> .
@prefix unit: <http://qudt.org/vocab/unit/> .
@prefix ex: <ex:> .

# Spatial elements
ex:Site   a   brick:Site ;
    brick:hasPart   ex:Building ;
.

ex:Building a   brick:Building ;
    brick:hasPart   ex:Floor ;
.

ex:Floor    a   brick:Floor ;
    brick:hasPart   ex:HVAC_Zone_1, ex:HVAC_Zone_2, ex:Open_Office,
                    ex:Bathroom, ex:Private_Office, ex:Kitchenette, ex:Corridor,
                    ex:Lighting_Zone_001, ex:Lighting_Zone_002, ex:Lighting_Zone_Bathroom,
                    ex:Lighting_Zone_PO, ex:Lighting_Zone_Kitchenette, ex:Lighting_Zone_Corridor
    ;
.

ex:Open_Office a brick:Room .
ex:Bathroom a brick:Room .
ex:Private_Office a brick:Room .
ex:Kitchenette a brick:Room .
ex:Corridor a brick:Space .

ex:HVAC_Zone_1  a   brick:HVAC_Zone ;
    brick:hasPart  ex:Open_Office, ex:Bathroom, ex:Corridor, ex:Private_Office ;
.

ex:HVAC_Zone_2  a   brick:HVAC_Zone ;
    brick:hasPart  ex:Kitchenette, ex:Corridor ;
.


# Lighting elements
ex:OO_Lighting_1_Sensor a brick:Illuminance_Sensor .
ex:OO_Lighting_2_Sensor a brick:Illuminance_Sensor .
ex:Bathroom_Lighting_Sensor a brick:Illuminance_Sensor .
ex:Private_Office_Lighting_Sensor a brick:Illuminance_Sensor .
ex:Kitchenette_Lighting_Sensor a brick:Illuminance_Sensor .
```

```
ex:Corridor_Lighting_Sensor a brick:Illuminance_Sensor .

ex:OO_Lighting_1 a brick:Luminaire ;
    brick:hasPoint  ex:OO_Lighting_1_Sensor ;
.
ex:OO_Lighting_2 a brick:Luminaire ;
    brick:hasPoint  ex:OO_Lighting_2_Sensor ;
.
ex:Bathroom_Lighting a brick:Luminaire ;
    brick:hasPoint  ex:Bathroom_Lighting_Sensor ;
.
ex:Private_Office_Lighting a brick:Luminaire ;
    brick:hasPoint  ex:Private_Office_Lighting_Sensor ;
.
ex:Kitchenette_Lighting a brick:Luminaire ;
    brick:hasPoint  ex:Kitchenette_Lighting_Sensor ;
.
ex:Corridor_Lighting a brick:Luminaire ;
    brick:hasPoint  ex:Corridor_Lighting_Sensor ;
.

ex:Lighting_Zone_OO1 a brick:Lighting_Zone ;
    brick:isFedBy ex:OO_Lighting_1 ;
    brick:isPartOf ex:Open_Office ;
.
ex:Lighting_Zone_OO2 a brick:Lighting_Zone ;
    brick:isFedBy ex:OO_Lighting_2 ;
    brick:isPartOf ex:Open_Office ;
.
ex:Lighting_Zone_Bathroom a brick:Lighting_Zone ;
    brick:isFedBy ex:Bathroom_Lighting ;
    brick:isPartOf ex:Bathroom ;
.
ex:Lighting_Zone_Private_Office a brick:Lighting_Zone ;
    brick:isFedBy ex:Private_Office_Lighting ;
    brick:isPartOf ex:Private_Office ;
.
ex:Lighting_Zone_Kitchenette a brick:Lighting_Zone ;
    brick:isFedBy ex:Kitchenette_Lighting ;
    brick:isPartOf ex:Kitchenette ;
.
ex:Lighting_Zone_Corridor a brick:Lighting_Zone ;
    brick:isFedBy ex:Corridor_Lighting ;
    brick:isPartOf ex:Corridor ;
.

# VAVs
ex:VAV_Box_1 a brick:VAV ;
    brick:hasPoint ex:SATS_1, ex:SATSP_1 ;
    brick:hasPart  ex:Damper_1, ex:Heating_Coil_1 ;
    brick:feeds ex:HVAC_Zone_1 ;
.
ex:SATS_1 a brick:Supply_Air_Temperature_Sensor .
ex:SATSP_1 a brick:Supply_Air_Temperature_Setpoint .
ex:Damper_1 a brick:Damper ;
    brick:hasPoint ex:Damper_1_Position ;
.
ex:Damper_1_Position    a   brick:Damper_Position_Command .
ex:Heating_Coil_1 a brick:Heating_Coil ;
    brick:hasPart ex:Heating_Valve_1 ;
```

```
.
ex:Heating_Valve_1  a   brick:Heating_Valve ;
    brick:hasPoint  ex:Valve_Command_1 ;
.
ex:Valve_Command_1  a   brick:Valve_Command .

ex:VAV_Box_2 a brick:VAV ;
    brick:hasPoint ex:SATS_2, ex:SATSP_2 ;
    brick:hasPart  ex:Damper_2, ex:Heating_Coil_2 ;
    brick:feeds ex:HVAC_Zone_2 ;
.
ex:SATS_2 a brick:Supply_Air_Temperature_Sensor .
ex:SATSP_2 a brick:Supply_Air_Temperature_Setpoint .
ex:Damper_2 a brick:Damper ;
    brick:hasPoint ex:Damper_2_Position ;
.
ex:Damper_2_Position    a   brick:Damper_Position_Command .
ex:Heating_Coil_2 a brick:Heating_Coil ;
    brick:hasPart ex:Heating_Valve_2 ;
.
ex:Heating_Valve_2  a   brick:Heating_Valve ;
    brick:hasPoint  ex:Valve_Command_2 ;
.
ex:Valve_Command_2  a   brick:Valve_Command .


ex:Exhaust_Fan_1 a brick:Exhaust_Fan ;
    brick:isFedBy ex:Bathroom ;
.

# AHU
ex:AHU  a brick:AHU ;
    brick:hasPart   ex:Supply_Fan, ex:Return_Fan,
                    ex:Cooling_Coil, ex:Heating_Coil, ex:Filter,
                    ex:Outside_Damper, ex:Mixed_Damper, ex:Exhaust_Damper ;
    brick:feeds ex:VAV_Box_1, ex:VAV_Box_2 ;
    brick:isFedBy ex:HVAC_Zone_1 ;
    brick:hasPoint  ex:OATS, ex:RATS, ex:MATS, ex:SATS,
                    ex:pre_hc_temp, ex:pre_cc_temp,
                    ex:RAFS, ex:SAFS ;
.
ex:OATS a brick:Outside_Air_Temperature_Sensor .
ex:RATS a brick:Return_Air_Temperature_Sensor .
ex:MATS a brick:Mixed_Air_Temperature_Sensor .
ex:SATS a brick:Supply_Air_Temperature_Sensor .
ex:pre_hc_temp a brick:Temperature_Sensor .
ex:pre_cc_temp a brick:Temperature_Sensor .
ex:SAFS a brick:Supply_Air_Flow_Sensor .
ex:RAFS a brick:Return_Air_Flow_Sensor .

ex:Supply_Fan a brick:Supply_Fan ;
    brick:hasPoint  ex:SF_Speed ;
.
ex:SF_Speed a brick:Frequency_Command .

ex:Return_Fan a brick:Return_Fan ;
    brick:hasPoint  ex:SF_Speed ;
.
ex:RF_Speed a brick:Frequency_Command .

```

```
ex:Cooling_Coil a brick:Cooling_Coil ;
    brick:hasPoint ex:CC_Valve ;
.
ex:CC_Valve a brick:Cooling_Valve ;
    brick:hasPoint  ex:CC_Valve_CMD ;
.
ex:CC_Valve_CMD a brick:Valve_Command ;
.
ex:Heating_Coil a brick:Heating_Coil ;
    brick:hasPoint ex:HC_Valve ;
.
ex:HC_Valve a brick:Heating_Valve ;
    brick:hasPoint  ex:HC_Valve_CMD ;
.
ex:HC_Valve_CMD a brick:Valve_Command ;
.

ex:Filter a brick:Filter .

ex:Outside_Damper a brick:Damper ;
    brick:hasPoint ex:Outside_Damper_Command ;
.
ex:Outside_Damper_Command a brick:Damper_Position_Command .
ex:Mixed_Damper a brick:Damper ;
    brick:hasPoint ex:Mixed_Damper_Command ;
.
ex:Mixed_Damper_Command a brick:Damper_Position_Command .
ex:Exhaust_Damper a brick:Damper ;
    brick:hasPoint ex:Exhaust_Damper_Command ;
.
ex:Exhaust_Damper_Command a brick:Damper_Position_Command .


# controller
ex:Controller a brick:Controller ;
    brick:controls ex:RF_Speed ;
    brick:hasInput ex:RAFS ;
.

# internal topology of AHU
ex:RATS brick:feeds ex:Return_Fan .
ex:Return_Fan brick:feeds ex:RAFS .
ex:RAFS brick:feeds ex:Exhaust_Damper, ex:Mixed_Damper .
ex:Mixed_Damper brick:feeds ex:MATS .
ex:Outside_Damper brick:feeds ex:MATS .
ex:MATS brick:feeds ex:Filter .
ex:Filter brick:feeds ex:pre_hc_temp .
ex:pre_hc_temp brick:feeds ex:Heating_Coil .
ex:Heating_Coil brick:feeds ex:pre_cc_temp .
ex:pre_cc_temp brick:feeds ex:Cooling_Coil .
ex:Cooling_Coil brick:feeds ex:SATS .
ex:SATS brick:feeds ex:Supply_Fan .
ex:Supply_Fan brick:feeds ex:SAFS .
```

Figure 9.1: Brick expression of the building and subsystems illustrated in Figure 2.2

```
@prefix brick: <https://brickschema.org/schema/1.1/Brick#> .
@prefix bsh: <https://brickschema.org/schema/1.1/BrickShape#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix tag: <https://brickschema.org/schema/1.1/BrickTag#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .

bsh:Temperature_Sensor_TagShape a sh:NodeShape ;
    sh:rule [ a sh:TripleRule ;
            sh:condition [ # _:has_Point_condition
                sh:property [
                    sh:path brick:hasTag ;
                    sh:qualifiedMinCount 1 ;
                    sh:qualifiedValueShape [
                        sh:hasValue tag:Point ;
                    ] ;
                ] ;
            ],
            [ # _:has_Sensor_condition
                sh:property [
                    sh:path brick:hasTag ;
                    sh:qualifiedMinCount 1 ;
                    sh:qualifiedValueShape [
                        sh:hasValue tag:Sensor ;
                    ] ;
                ] ;
            ],
            [ # _:has_Temperature_condition
                sh:property [
                    sh:path brick:hasTag ;
                    sh:qualifiedMinCount 1 ;
                    sh:qualifiedValueShape [
                        sh:hasValue tag:Temperature ;
                    ] ;
                ] ;
            ] ,
            [ # _:has_exactly_3_tags_condition
                sh:property [
                    sh:maxCount 3;
                    sh:minCount 3;
                    sh:path brick:hasTag ;
                ] ;
            ] ;
            sh:object brick:Temperature_Sensor ;
            sh:predicate rdf:type ;
            sh:subject sh:this ] ;
    sh:targetSubjectsOf brick:hasTag .
```

Figure 9.2: Expanded RDF graph for the shape described in Figure 4.11.