A Programming Model for Portable Fault Detection and Diagnosis

Dimitris Mavrokapnidis
University College London
d.mavrokapnidis@ucl.ac.uk

Gabe Fierro
Colorado School of Mines
gtfierro@mines.edu

Ivan Korolija
University College London
i.korolija@ucl.ac.uk

Dimitrios Rovas
University College London
d.rovas@ucl.ac.uk

ABSTRACT

Portable applications support the write once, deploy everywhere paradigm. This paradigm is particularly attractive in building applications, where current practice involves the manual deployment and configuration of such applications, requiring significant engineering effort and concomitant costs. This is a tedious and error-prone process which does not scale well. Notwithstanding recent advances in semantic data modelling that allow a unified representation of buildings, we still miss a paradigm for deploying portable building applications at scale. This paper introduces a portable programming model for such applications, which we examine in the context of Fault-Detection and Diagnosis (FDD). In particular, we look at the separation of the FDD logic and the configuration with specific data inputs. We architect a software system that enables their self-configuration and execution across various building configurations, expressed in terms of Brick metadata models. Our initial results from authoring and executing APAR (AHU Performance Assessment Rules) on multiple AHUs of two museums demonstrate the potential of our model to reduce repetitive tasks and deployment costs of FDD applications.

CCS CONCEPTS

• **Software and its engineering** → **Abstraction, modeling and modularity**; • **Information systems** → *Graph-based database models.*

KEYWORDS

Programming, FDD, Portability, Scalability, Brick, RDF, SHACL, Metadata, Semantic Web, Ontologies

ACM Reference Format:
Dimitris Mavrokapnidis, Gabe Fierro, Ivan Korolija, and Dimitrios Rovas. 2023. A Programming Model for Portable Fault Detection and Diagnosis. In *The 14th ACM International Conference on Future Energy Systems (e-Energy '23), June 20–23, 2023, Orlando, FL, USA.* ACM, Coimbra, Portugal, 5 pages. https://doi.org/10.1145/3575813.3595190

e-Energy '23, June 20–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0032-3/23/06…$15.00
https://doi.org/10.1145/3575813.3595190

1 INTRODUCTION

The value and insights generated by data-driven services are being increasingly appreciated in the context of the built environment. For example, Fault Detection and Diagnosis (FDD) applications use building data to uncover hidden system inefficiencies and identify opportunities to reduce energy costs and increase occupants' comfort [11, 22]. However, the configuration of such applications requires discovering and accessing data from diverse data sources [14] including Building Information Models (BIM) [21] and Building Management Systems (BMS) [18]. The challenge of discovering and reusing building data, combined with the poor state of documentation of many BMS systems, has led to limited adoption of such approaches in the majority of buildings [20]. As a result, many applications are still developed on an *ad-hoc* basis and are hardly reusable in different buildings [11, 20].

To address these challenge of data representation under a unified model, semantic data models such as Brick [7], Project Haystack [5], Real Estate Core [16], and ASHRAE 223p [12] have emerged with the purpose of making building (meta-)data easily discoverable and accessible through uniform and machine-interpretable metadata representations. Nevertheless, despite the potential of those advancements to enable application portability, we still miss the paradigm for developing applications once and executing them in multiple buildings. This paper introduces a programming model for authoring and reusing portable building applications focusing, without loss of generalisation, on the case of FDD.

1.1 Requirements for portable applications

We identify the following challenges (C1-C4) in developing portable building applications.

- C1: **Data availability**: The need to configure a building application depending on the available data.
- C2: **Model expressivity**: The ability of the modeller to discover the required semantic metadata across different building configuration.
- C3: **System applicability**: The need to ensure whether an application is usable across various building system configuration.
- C4: **Data modalities**: Ensure proper units, temporal resolutions, and temporal semantics from data sources.

In this paper, we focus mainly on the case of rule-based FDD applications. To illustrate the challenges of making FDD rules portable, we use the running example of rule R_1 from the Air-Handling Unit (AHU) Performance Assessment Rules (APAR) [25]. R_1 which verifies that an AHU's supply air temperature is greater than the mixed air temperature plus the temperature drop over the supply fan when the AHU is in heating mode. The rule can be expressed as the following inequality:

$$R_1 : T_{\text{sa}} < T_{\text{ma}} + \Delta T_{\text{sf}} - \varepsilon \quad (1)$$

To configure and execute this rule on a given building, the developer must determine if there are any AHUs in the building (**C3**), determine the presence and identity of the mixed and supply air temperature sensors in those AHUs (**C1**), author a Brick query to discover those AHUs and sensors in a building (**C2**), and finally fetch the data and perform any unit conversion and data cleaning (**C4**). This intensive and largely manual process must be repeated for each AHU in each building where the developer intends the rule to run [10].

Recent work on semantic metadata has offered new ways for intelligent building software to become wholly or partially self-configuring [15, 17]. Still, it requires developers to become familiar with an "alphabet soup" of different technologies: RDF[3], OWL[1], SPARQL[2], and so on. Our proposed programming model for portable FDD applications *separates* the expression and execution of FDD rules from how those rules are configured for each building. This increases the usability of portable building software and enables cheaper deployment of fault detection rules.

2 BACKGROUND AND PRIOR WORK

We discuss recent efforts to enable "portable" applications in smart buildings using recent metadata models.

2.1 Building metadata models

Past work has established the difficulty of writing data-driven applications for buildings, due in part to a lack of standard digital representations that facilitate data discovery [9, 10]. Contemporary research develops standardized representations that address this lack of introspection and discoverability, including Project Haystack [5], Brick Schema [7], Real Estate Core (REC) [16], and the Building Topology Ontology (BOT) [24], amongst others [23].

These representations work by abstracting complex and highly-interrelated cyberphysical systems like HVAC, lighting, electrical and plumbing systems into directed graph structures. These graphs provide a machine-readable interface that directly encodes the identity of data sources, building assets, and the relationships between them. Applications query these graphs to configure their operation; this involves retrieving the composition and topology of building systems and identifying data sources or control inputs to be used in the application.

Despite the promising use of these emerging metadata standards, accessing the required data for specific applications remains challenging. For example, Bhattacharya et al. reported an inability to run three simple diagnostics applications in a portfolio of 10 buildings due to a lack of the required semantic information richness [9].

2.2 Portable Application Platforms

A significant challenge for enabling the deployment of building applications at scale is the time-consuming and site-specific effort of configuring an application to run on a given building. Recent analytics platforms offer several methods for enabling the *mass-customization* of applications: the (semi-)automated process of configurating an application through the use of semantic metadata [7].

Building Application Stack (BAS) [19] provides a fuzzy query interface over a graph of building components and control interfaces. BuildingDepot [26] adopts a template-based approach which restricts user applications to those that can be expressed using a pre-defined sets of building entities and data sources. This trades expression of arbitrary applications for a simplified configuration experience. Mortar [15] requires tens of lines of code to express queries and application configuration logic; SkyFoundry [6] and Energon [17] use non-standard and purpose-built programming and query languages to reduce lines of code, but still require developer-driven reconfiguration between deployment sites. [8] proposed a query relaxation algorithm to improve the retrieval of building data through SPARQL results and, therefore, increase query portability across different building configurations.

These approaches often address only the relationship between descriptions of the building and the actual telemetry, leaving the implementation of application portability to the developer. Our proposed approach separates the portability mechanism from the expression of the application logic, simplifying the development experience.

3 PORTABLE PROGRAMMING MODEL

In this section we present our formalism for expressing *portable* rule-based fault detection and diagnosis applications. Our programming model eliminates the complex and site-specific configuration effort borne by other approaches by decoupling the *identity* of logical quantities used in the rule (e.g. *supply air temperature*) from their definition in the underlying building. Developers express FDD rules using portable *computational quantities* (CQs) which are defined by functions over an underlying formal representation of a building, its assets and data sources. Individual CQs access this formal representation for a particular building to *resolve* themselves to a real-valued quantity for rule evaluation.

3.1 Computational Quantities

We express rules in terms of **computational quantities**. A computational quantity (CQ) is a *portable* definition of a physical quantity in the building such as the supply air flow rate or the mixed air temperature of an air-handling unit. A CQ is portable because it encodes multiple ways that a quantity may be found in the building: a quantity may be (a) observed directly by sensors or other digital I/O points from the building management system, (b) computed indirectly through other observations or CQs, or (c) assumed to be a default value. *Resolving* a CQ on a building model identifies and executes a specific programmatic implementation for returning the value of the desired quantity in that building; thus, rule developers do not have to handle the complexity of expressing a rule in a portable manner.

Formally a CQ is a function $CQ : G \rightarrow \mathbb{R}$ that returns a real-valued quantity when executed against a building's Brick model; this quantity can be used in any subsequent computation like an FDD rule. Each CQ is defined by a set of possible *resolutions*; a resolution is an executable plan for producing a value from the building that can be used in an FDD rule calculation. We define three types of CQ resolutions: graph, computational, and default.

A *Graph CQ resolution* is a SHACL [4] shape that semantically describes the set of ways a CQ could be found in a Brick model. For example, T_{ma} in our running example could be resolved as an instance of the Brick class *Mixed Air Temperature Sensor* associated with an instance of Brick's *AHU* class.

A *Computational CQ resolution* is a function over other CQs that allows expressing computational relationships between those CQs. In our running example, if a sensor could not be found in the graph, the mixed air temperature could be estimated using the equation: $T_{ma} = T_{oa} * F_{oa} + T_{ra} * F_{ra}$ where T_{oa}/T_{ra} & F_{oa}/F_{ra} stand for the *Outside/Return Air Temperature & Air Flow Rate* respectively.

A *Default or user-defined CQ resolution* is a human-provided default value. This accounts for rule parameters (ϵ in our running example), unknown but assumable quantities (ΔT_{sf} in our running example), and other constants.

A CQ is expressed as a "decision tree" of possible resolutions. These are ordered by the accuracy and relevancy of each resolution: typically, graph resolutions that identify actual values in the building management system are preferred over computational resolutions, which are preferred over default values. *Resolving* a CQ on a graph involves determining the most preferred resolution for that CQ and returning the corresponding value. Different resolutions may result in differing accuracy in the estimation or measurement of the actual quantity. To account for this, the decision tree can be structured to prefer certain resolutions over others.

3.2 Expressing Portable Rules

We express an FDD rule R_i as a function over a graph G and a set of n computational quantities $CQ_1, \ldots, CQ_n$.

$$R_i : (G, f(CQ_1, \ldots, CQ_n)) \xrightarrow{resolve} f'(v_1, \ldots, v_n) \xrightarrow{execute} \{\text{T}, \text{F}\} \quad (2)$$

Above, the function $f(\ldots)$ is the portable site-agnostic expression of the FDD rule. Figure 1 contains an example of such a rule: note that the rule definition is similar to the original mathematical formulation in §1.1, and that the rule definition contains *no site-specific logic*. A function $f(\ldots)$ is ported to a new building through the process of *resolution*. Resolving a rule produces a new function $f'(\ldots)$ in which each of the CQs has been resolved to an actual real-valued quantity that can be used for evaluating the rule ($CQ_i : G \longrightarrow v_i \in \mathbb{R}$). In case that a CQ_i cannot be resolved, the platform produces an error that the rule cannot be executed on G.

In Figure 1, expressions like `Tsa(G)` perform the resolution of the CQ `Tsa` on the graph G. For simplicity we consider boolean-valued FDD rules which return true if a fault was detected, but the proposed programming model extends easily to other types of output. Figure 1 illustrates the definition and evaluation of two APAR rules in terms of our reference implementation.

```python
# import provided definitions of common computational quantities
from APAR.computational_quantities import Tsa, Tma, Tsf, Toa
def rule1(G: Graph) -> bool:
    return Tsa(G) < Tma(G) + Tsf(G) - ϵ # ϵ is the error threshold
def rule5(G: Graph) -> bool:
    return Toa(G) > Tsa(G) - Tsf(G) + ϵ # ϵ is the error threshold
```

Figure 1: Python implementation of Rule 1 and Rule 5 from APAR [25] using the proposed portable programming solution. `Tsa`, `Tma`, `Tsf`, and `Toa` are CQs

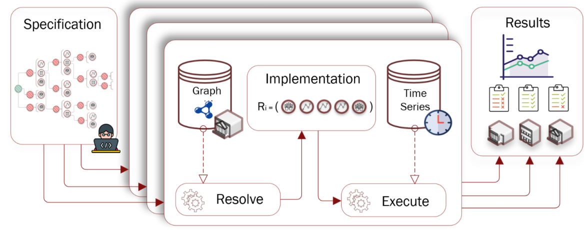

Figure 2: Portable FDD software architecture

4 IMPLEMENTATION

Our portable programming model acts as an interface between any rule-based FDD algorithm and its actual implementation in software. In this section, we explain our Graph CQ resolution mechanism, which enables the portable expression of data requirements across many different Brick models and therefore buildings. In contrast to prior mechanisms [8, 15, 17] we enable portability without placing an undue burden on the rule developer. We then explore the proposed CQ resolution algorithm in the context of APAR R_1 ([25]) and demonstrate its self-configuration and execution on two buildings and 24 AHU components.

4.1 Resolution of Graph CQs

All the potential Graph CQ resolutions are implemented as SHACL [4] shapes during the specification of a rule. These incorporate various semantic descriptions of how the Graph CQ may be expressed in a Brick model. For example, the *"Mixed Air Temperature"* CQ can be expressed in a Brick model either as a point of an AHU or as a point of a *"Mixed Damper"* that is part of the same AHU. The shape authoring process is facilitated through user-friendly functions that transform path descriptions into SHACL shapes.

During the resolution of a rule, whenever a successful path is discovered, we enrich the Brick model with an additional semantic relationship that creates uniform structures within each Brick model. This allows us to automate the extraction of Graph CQs without model-specific queries.

4.2 Self-configuration

Recall that a rule is expressed as a set of computational quantities. The execution of that rule (e.g. in Equation 2) involves resolving each of the CQs on the target graph G by traversing the decision tree associated with each CQ. Figure 2 illustrates a high-level overview of how our system resolves and executes a portable FDD library.

The self-configuration process transforms a portable *specification* of an FDD rule into an executable implementation of that rule on a particular Brick model. The implementation describes the CQs required to run the rule, which have either been discovered in the model as Graph CQs or provided as default CQs.

Figure 3 illustrates how a rule specification is self-configured to produce an executable implementation. The left part of Figure 3 presents an example of a specification of a self-configurable rule. In particular, rule R_1 is defined by three computational quantities CQ_1, CQ_2, CQ_3, each of which is expressed as a decision tree of potential resolutions. To elucidate the self-configuration process, we focus on CQ_1. If CQ_1 cannot be resolved as a Graph CQ, we can then look for its successful resolution as Computational CQ which requires the resolution of CQ_4 and CQ_5. In the case those

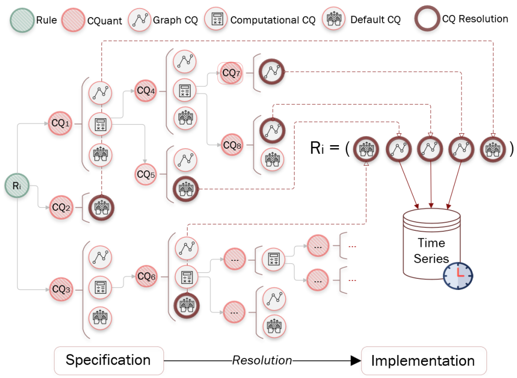

Figure 3: Specification of a self-configurable rule. CQs can be reused across multiple rules

CQs cannot be resolved, then the user may provide a default value to resolve it. Overall, the variety of possible resolutions is defined by the application developer according to the logic of the FDD rules.

The right part of Figure 3 illustrates the implementation of rule R_1 for a specific building (expressed in a graph). The self-configuration of R_1 results in a specific implementation: $R_1 = (CQ_2, CQ_5, CQ_6, CQ_7, CQ_8)$ where CQ_5, CQ_6, CQ_7 are successfully resolved as Graph CQs over the Brick model of a particular building; CQ_2, CQ_8 are Default CQs.

4.3 Evaluation

We evaluate our programming model by determining how well it can express rules from the industry-standard APAR ruleset [25]. We implement these rules using our programming model and execute them over two real buildings, each represented by a Brick model, comprising 24 air handling units (AHUs).

Figure 1 contains the implementation of two different APAR rules; for space reasons, we elide the implementation of the other APAR rules. This demonstrates that our programming model is *succinct*: it can express a single rule in 2 lines of code; this might take 5-10 lines of code in Energon [17] and 50-100 lines of code in Mortar [15]. The code sample also demonstrates that it is possible to reuse a library of CQs to implement multiple rules.

To demonstrate the portability capabilities of our proposed model, we show the results of executing Rule R_1 over 3 AHUs in Table 1. Columns 2 and 3 of the table show how the *resolution* of the rule differed across 3 different AHUs. AHU1 contains a mixed air temperature sensor which was found by the CQ resolution; however, because AHU5 lacks this sensor, the CQ resolution identifies a combination of air flow and air temperature sensors to estimate the mixed air temperature. AHU2 lacks these flow sensors, so the CQ resolution identifies a default mixing ratio of 50:50 for estimating the mixed air temperature. These results demonstrate the ability of our programming model to configure rules automatically according to the available data. Figure 2 displays the results of executing our CQ-based rules over 24 AHUs, showing that the model can effectively perform fault detection in real settings.

4.4 Discussion

Developing portable applications using Mortar[15], Energon[17], depends on the ability of the rule developer to write successful

AHU	**Implementation of** $R_1 : T_{\mathbf{sa}} < T_{\mathbf{ma}} + \Delta T_{\mathbf{sf}} - \varepsilon$	
AHU1 (Bldg1)	GraphCQ (T_{ma})	`brick:Mixed_Air_Temp.`
AHU5 (Bldg1)	GraphCQ (T_{ra})	`brick:Return_Air_Temp.`
	GraphCQ (T_{oa})	`brick:Outside_Air_Temp.`
	GraphCQ (F_{ra})	`brick:Return_Air_Flow`
	GraphCQ (F_{oa})	`brick:Outside_Air_Flow`
AHU2 (Bldg2)	GraphCQ (T_{ra})	`brick:Return_Air_Temp.`
	GraphCQ (T_{oa})	`brick:Outside_Air_Temp.`
	DefaultCQ (F_{ra})	0.5
	DefaultCQ (F_{oa})	0.5
All AHUs	GraphCQ (T_{sa})	`brick:Supply_Air_Temp.`
	DefaultCQ (ΔT_{sf})	1.0
	DefaultCQ (ε)	1.1

Table 1: Implementation Description of Portable APAR R_1

	Rule1	Rule2	Rule3	Rule4
# AHUs with Fault	11/24	9/24	15/24	8/24

Table 2: The number of faulted AHUs found across two buildings and 24 AHUs with the same rule implementations

queries on the graph model. Instead, our model shifts the burden of configuration to the CQ specification process. Rule developers, unlike in prior systems, do not need to write any queries or handle the portability of applications themselves.

Writing FDD rules only requires a few lines of code (Figure 1); in prior systems these rules would often require multiple queries. For example, AHU2 and AHU5 in Table 1 do not contain the expected `brick:Mixed_Air_Temp.` sensor. In other platforms, developers would have to re-write the query to consider alternative data sources or assign default values – both of these cases are handled automatically by the CQ resolution mechanism. In our proposed model, developers specify the application logic once using portable abstractions of building data.

Our programming model relies on a data access service that resolves identifiers in the graph to the actual telemetry. Configuring the data access service and Brick model are building-specific one-time costs necessary for data-driven applications [13, 15, 17, 26].

5 CONCLUSIONS AND FUTURE WORK

In this paper we have introduced a new programming model for portable rule-based FDD applications that eliminates the need for rule authors to explicitly query metadata models. We have employed our model to express and execute standard FDD rules in two different buildings, demonstrating its potential to reduce manual and repetitive tasks for rule developers, thereby accelerating the adoption of FDD applications at scale. Our next steps include the implementation of our programming model in further FDD rulesets and execution in larger and more diverse building portfolios.

ACKNOWLEDGMENTS

The research leading to these results has been partially funded by the CBIM-ETN funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 860555, and supported by the technical team of the Museum of London, who provided access to data from two Museums.

REFERENCES

[1] 2012. Web Ontology Language (OWL). https://www.w3.org/OWL/

[2] 2013. SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/

[3] 2014. Resource Description Framework (RDF). https://www.w3.org/RDF/

[4] 2017. Shapes Constraint Language (SHACL). https://www.w3.org/TR/shacl/

[5] 2023. Home – Project Haystack. https://project-haystack.org/

[6] 2023. Home – SkyFoundry. https://skyfoundry.com/

[7] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Bergés, David Culler, Rajesh K. Gupta, Mikkel Baun Kjærgaard, Mani Srivastava, and Kamin Whitehouse. 2018. Brick : Metadata schema for portable smart building applications. *Applied Energy* 226 (2018), 1273–1292. https://doi.org/10.1016/j.apenergy.2018.02.091

[8] Imane Lahmam Bennani, Anand Krishnan Prakash, Marina Zafiris, Lazlo Paul, Carlos Duarte Roa, Paul Raftery, Marco Pritoni, and Gabe Fierro. 2021. Query Relaxation for Portable Brick-Based Applications. In *Proceedings of the 8th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation* (Coimbra, Portugal) *(BuildSys '21)*. Association for Computing Machinery, New York, NY, USA, 150–159. https://doi.org/10.1145/3486611.3486671

[9] Arka Bhattacharya, David Culler, Dezhi Hong, Kamin Whitehouse, and Jorge Ortiz. 2014. Writing Scalable Building Efficiency Applications Using Normalized Metadata: Demo Abstract. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings* (Memphis, Tennessee) *(BuildSys '14)*. Association for Computing Machinery, New York, NY, USA, 196–197. https://doi.org/10.1145/2674061.2675031

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

[11] H. Burak Gunay, Weiming Shen, and Guy Newsham. 2019. Data analytics to improve building performance: A critical review. *Automation in Construction* 97 (Jan. 2019), 96–109. https://doi.org/10.1016/j.autcon.2018.10.020

[12] ASHRAE's BACnet Committee. 2018. https://www.ashrae.org/about/news/2018/ashrae-s-bacnet-committee-project-haystack-and-brick-schema-collaborating-to-provide-unified-data-semantic-modeling-solution

[13] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David Culler. 2013. BOSS: Building Operating System Services. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 443–457. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/dawson-haggerty

[14] Gabe Fierro, Anand Krishnan Prakash, Cory Mosiman, Marco Pritoni, Paul Raftery, Michael Wetter, and David E Culler. 2020. Shepherding metadata through the building lifecycle. In *Proceedings of the 7th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*. 70–79.

[15] Gabe Fierro, Marco Pritoni, Moustafa Abdelbaky, Daniel Lengyel, John Leyden, Anand Prakash, Pranav Gupta, Paul Raftery, Therese Peffer, Greg Thomson, and David E. Culler. 2019. Mortar: An Open Testbed for Portable Building Analytics. *ACM Trans. Sen. Netw.* 16, 1, Article 7 (dec 2019), 31 pages. https://doi.org/10.1145/3366375

[16] Karl Hammar, Erik Oskar Wallin, Per Karlberg, and David Hälleberg. 2019. The realestatecore ontology. In *The Semantic Web–ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part II 18*. Springer, 130–145.

[17] Fang He, Yang Deng, Yanhui Xu, Cheng Xu, Dezhi Hong, and Dan Wang. 2021. Energon: A Data Acquisition System for Portable Building Analytics. In *Proceedings of the Twelfth ACM International Conference on Future Energy Systems* (Virtual Event, Italy) *(e-Energy '21)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/3447555.3464850

[18] Jason Koh, Dezhi Hong, Rajesh Gupta, Kamin Whitehouse, Hongning Wang, and Yuvraj Agarwal. 2018. Plaster: An Integration, Benchmark, and Development Framework for Metadata Normalization Methods. In *Proceedings of the 5th Conference on Systems for Built Environments* (Shenzen, China) *(BuildSys '18)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3276774.3276794

[19] Andrew Krioukov, Gabe Fierro, Nikita Kitaev, and David Culler. 2012. Building application stack (BAS). In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys '12)*. Association for Computing Machinery, New York, NY, USA, 72–79. https://doi.org/10.1145/2422531.2422546

[20] Guanjing Lin, Hannah Kramer, Valerie Nibler, Eliot Crowe, and Jessica Granderson. 2022. Building Analytics Tool Deployment at Scale: Benefits, Costs, and Deployment Practices. *Energies* 15, 13 (2022). https://doi.org/10.3390/en15134858

[21] Dimitris Mavrokapnidis, Kyriakos Katsigarakis, Pieter Pauwels, Ekaterina Petrova, Ivan Korolija, and Dimitrios Rovas. 2021. A Linked-Data Paradigm for the Integration of Static and Dynamic Building Data in Digital Twins. In *Proceedings of the 8th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation* (Coimbra, Portugal) *(BuildSys '21)*. Association for Computing Machinery, New York, NY, USA, 369–372. https://doi.org/10.1145/3486611.3491125

[22] Evan Mills. 2011. Building commissioning: a golden opportunity for reducing energy costs and greenhouse gas emissions in the United States. *Energy Efficiency* 4, 2 (May 2011), 145–173. https://doi.org/10.1007/s12053-011-9116-8

[23] Marco Pritoni, Drew Paine, Gabriel Fierro, Cory Mosiman, Michael Poplawski, Avijit Saha, Joel Bender, and Jessica Granderson. 2021. Metadata Schemas and Ontologies for Building Energy Applications: A Critical Review and Use Case Analysis. *Energies* 14, 7 (2021). https://www.mdpi.com/1996-1073/14/7/2024

[24] Mads Holten Rasmussen, Maxime Lefrançois, Georg Ferdinand Schneider, and Pieter Pauwels. 2021. BOT: The building topology ontology of the W3C linked building data group. *Semantic Web* 12, 1 (2021), 143–161.

[25] Jeffrey Schein, Steven T. Bushby, Natascha S. Castro, and John M. House. 2006. A rule-based fault detection method for air handling units. *Energy and Buildings* 38, 12 (2006), 1485–1492. https://doi.org/10.1016/j.enbuild.2006.04.014

[26] Thomas Weng, Anthony Nwokafor, and Yuvraj Agarwal. 2013. 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 (BuildSys'13)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/2528282.2528285