Application-Driven Creation of Building Metadata Models with Semantic Sufficiency

Gabe Fierro
gtfierro@mines.edu
Colorado School of Mines
National Renewable Energy Laboratory
Golden, Colorado, U.S.A.

Avijit Saha
Avijit.Saha@nrel.gov
National Renewable Energy Laboratory
Golden, Colorado, U.S.A.

Tobias Shapinsky
Tobias.Shapinsky@nrel.gov
National Renewable Energy Laboratory
Golden, Colorado, U.S.A.

Matthew Steen
Matthew.Steen@nrel.gov
National Renewable Energy Laboratory
Golden, Colorado, U.S.A.

Hannah Eslinger
Hannah.Eslinger@nrel.gov
National Renewable Energy Laboratory
Golden, Colorado, U.S.A.

ABSTRACT

Semantic metadata models such as Brick, RealEstateCore, Project Haystack, and BOT promise to simplify and lower the cost of developing software for smart buildings, enabling the widespread deployment of energy efficiency applications. However, creating these models remains a challenge. Despite recent advances in creating models from existing digital representations like point labels and architectural models, there is still no feedback mechanism to ensure that the human input to these methods results in a model that can actually support the desired software.

In this paper, we introduce the notion of *semantic sufficiency*, a practical principle for semantic metadata model creation that asserts that a model is "finished" when it contains the metadata necessary to support a given set of applications. To support semantic sufficiency, we design a standard representation for capturing application metadata requirements and a templating system for generating common metadata model components with limited user input. We then construct an iterative model creation workflow that integrates metadata requirements to direct the model creation effort, and present several novel optimizations that increase the model utility while minimizing the effort by a human operator. These new abstractions for model creation and validation lower model development costs and ensure the utility of the resulting model, thus facilitating the adoption of intelligent building applications.

CCS CONCEPTS

• **Information systems** → **Information systems applications**; **Ontologies**; **Retrieval tasks and goals**.

KEYWORDS

Brick, ontology, metadata, semantics, applications

BuildSys '22, November 9–10, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9890-9/22/11.
https://doi.org/10.1145/3563357.3564083

ACM Reference Format:
Gabe Fierro, Avijit Saha, Tobias Shapinsky, Matthew Steen, and Hannah Eslinger. 2022. Application-Driven Creation of Building Metadata Models with Semantic Sufficiency. In *The 9th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys '22), November 9–10, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3563357.3564083

1 INTRODUCTION

The digitization of the built environment will enable a wide array of intelligent, data-driven applications — including model predictive control, predictive maintenance, and fault detection and diagnosis — but will also require confronting new data management challenges. In particular, the increasing prevalence of digital control and monitoring systems in buildings means more data is available about buildings than ever before [13]. However, the effective use of that data is impeded by (1) the heterogeneity of the building stock and (2) the lack of standardized descriptions of buildings and the data they produce [7].

Semantic metadata models have emerged to address this need. Graph-based models such as Brick [5], RealEstateCore [18], Building Topology Ontology (BOT) [30], and the proposed ASHRAE Standard 223P [4] use graph-based data models to capture rich, formal descriptions of buildings: their architecture, the structure and composition of their subsystems, and the data they produce. Other building metadata models like Google Digital Buildings [8] and Project Haystack [2] capture similar building characteristics. These metadata models are shown to reduce development time for many different kinds of applications for buildings [28]. In platforms like Mortar [16], Energon [19], and the DataClearingHouse [3], applications query metadata models to configure themselves for execution on a particular building. This reduces the amount of manual configuration necessary. While use of these models is growing across industry, academia, and regulatory sectors, *creating* metadata models remains a barrier to their widespread adoption.

Prior work has explored techniques for producing metadata models from existing digital representations such as building management system (BMS) points [10, 20], building information models (BIM) [22], and others [15]. While these efforts all help to automate

the model authoring process, they omit the crucial feedback of whether or not the produced model is actually useful: in short, how does one know when a building metadata model is "done?" To date, there has been no effort to align the effort of *creating* a metadata model to the *needs* of downstream applications.

In this work, we introduce and formalize the notion of **semantic sufficiency**, a principle for model creation stating that a model is "complete" when it contains enough semantic metadata to support a desired suite of applications. We show how the **metadata requirements** of applications can be represented in a structured and formal manner using *shapes*. We also introduce the idea of **templates** that automate the creation of common patterns in metadata models. We then show how templates and shapes can inform the creation of a building metadata model to both reduce required input from a human operator and ensure that the resulting model supports all of the intended applications. To wit, the contributions of this paper are:

- introduction and formalization of *semantic sufficiency* (§3), a practical principle for metadata model creation
- design and implementation of *shapes* (§3) and *templates* (§4) for capturing application requirements and automating model creation, respectively
- a novel algorithm for incremental model creation (§5)
- an open-source reference implementation of the above (§7)

We also demonstrate the utility of the proposed approach in a case study implementing high-performance sequences of operations on a public reference building (§6). Although the examples and evaluations in this paper focus on the Brick ontology, none of the techniques or contributions above are specific to Brick — they will work on any Resource Description Framework (RDF) [23]-based ontology including RealEstateCore [18], ASHRAE 223P [4] and BOT [30].

2 BACKGROUND AND PRIOR WORK

Semantic metadata models are digital representations of buildings, their components and their data; these can be expressed naturally as graphs. Graphs are effective for capturing the composition and topology of building systems, which tend to be complex and highly inter-related [14]. Many of these graphs are built with the RDF standard, which is an expressive and flexible data model standard for describing directed, labeled graphs. Graphs can be structured according to *ontologies*, which are formal descriptions of knowledge domains. Ontologies provide structure through formal logic axioms that ensure the statements in the graph have computationally interpretable meanings (i.e. "semantics"). Examples of contemporary semantic metadata models for buildings include Brick [5], RealEstateCore [18], BOT [30] and ASHRAE 223P [4]. Other metadata models like Google Digital Buildings [8] and Project Haystack [2] are built on bespoke rather than standard technologies, but can be exported to RDF graphs.

Automating the creation of building metadata models is a critical feature due to the size and complexity of building subsystems. Initial approaches focused on applying active learning techniques to parse unstructured labels in building management systems (BMS) to order to infer the types of sensors and other I/O points [10, 20]. These techniques struggle to infer basic relationships between the points and other devices in the building [24]. Other work has leveraged existing, structured, digital representations of buildings such as building information models [15, 22]. These techniques often have trouble accurately determining system topology and composition due to the variability in how these formats are applied in practice. As a result, there is still a need for human input both in disambiguating ad-hoc labels and descriptors, as well as determining which elements of large digital models are the most helpful.

The use of external constraints to inform the creation of a digital model has been established in the built environment literature, and is already used by several existing projects. BuildingSync uses XML schema documents to define what fields are required for different kinds of energy audits [25]. Model View Definitions in Industry Foundation Classes (IFC) define the subset of a building information model that must exist to enable certain kinds of applications. [27] uses the Shapes Constraint Language (SHACL) to validate a linked data model for particular building use cases (such as for the BOT ontology). Reasonable Ontology Templates (OTTR) [32] is a recently developed templating mechanism for authoring ontologies and semantic metadata models; we will investigate the use of OTTR in future work.

3 SEMANTIC SUFFICIENCY

A key concern when developing a semantic metadata model of a building is knowing when that model is "complete". We contend that the traditional notion of "completeness" – modeling all possible assets, data sources, components and relationships in a building – is intractable in most settings for two reasons. First, this metadata may simply be unavailable. Building subsystems are large, complex, and have many components, yet much of this detail is unavailable in brownfield settings where the only reliable sources of metadata are BMS graphics and point lists. Second, because there are innumerable perspectives of a building that can be modeled, metadata representations may be unbounded in their complexity, forcing developers to wade through thousands of unrelated statements to find the information they need. These motivate the need for a new guiding principle for semantic model creation.

To this end, we propose and formalize *semantic sufficiency*, a definition of "completeness" for semantic model creation that minimizes model creation effort and maximizes the utility of the produced model. Semantic sufficiency holds that a semantic model is "complete" when it contains sufficient information to support the configuration and execution of a *predetermined set* of applications. With this principle, we seek to formalize results from prior work that finds only a subset of all building points are required for many common data-driven applications [9, 10, 31].

To formalize semantic sufficiency, we must be able to specify application requirements and verify that a model satisfies those requirements. We begin by presenting a formalism for defining RDF-based metadata models and constraints over those models, then apply this formalism to defining application requirements.

3.1 Preliminaries

For simplicity, we restrict the definition of semantic metadata models to those that are defined using the RDF data model [23]. The RDF model defines a directed, labeled graph as a set of 3-element

```
@prefix brick: <https://brickschema.org/schema/Brick#> .
@prefix ex: <urn:bldg#> .
ex:vav1 a   brick:VAV ;
    brick:hasPoint ex:safs1, ex:sats1, ex:zts1, ex:co2s1 ;
    brick:hasPart  ex:vdmp1 ;
    brick:feeds ex:zone1 .
ex:safs1    a   brick:Supply_Air_Flow_Sensor .
ex:sats1    a   brick:Supply_Air_Temperature_Sensor .
ex:zts1     a   brick:Zone_Air_Temperature_Sensor .
ex:co2s1     a   brick:CO2_Sensor .
ex:vdmp1    a   brick:Damper ;
    brick:hasPoint ex:dmppos1 .
ex:dmppos1  a   brick:Damper_Position_Command .
ex:zone1    a   brick:HVAC_Zone ;
    brick:hasPart   ex:room1, ex:room2 .
ex:room1    a   brick:Room ;
    brick:hasPoint ex:co2s1 .
```

Figure 1: A Brick model defining a variable air volume box and its relationship to a zone and constituent room.

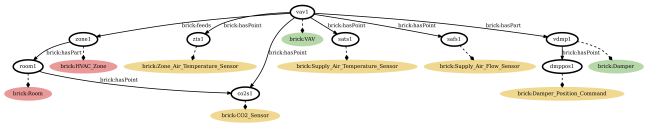

Figure 2: A graphical representation of the Brick model in Figure 1. Bold, white nodes are *entities* in the building whose types are given by *classes* (solid colored nodes).

statements called *triples*. Together, these statements constitute a digital representation of the building and its components. A triple $t \in T$ is a 3-tuple $(subject, predicate, object)$ — also written (s, p, o) — stating the relationship (*predicate*) that a *subject* node has to a *object* node. A graph G can be expressed as a set of triples:

$$G = \{t : t \in T\}$$

See [23] for a more complete description of RDF.

Figure 1 contains a textual representation of a Brick model; Figure 2 illustrates the graph described by these triples. The structure of these models is informed not only by the characteristics and components of a building, but also by the *ontology* used. We focus on ontologies built with the Web Ontology Language (OWL) [6] or SHACL [1]. A full tutorial on these languages is beyond the scope of the paper, but it is sufficient to know that an ontology is a formal encoding of domain knowledge into types, constraints, properties, and rules that can be computationally verified.

3.2 Application Requirements

To scope the discussion of how to specify application requirements, we first synthesize a generic definition of applications that incorporate semantic metadata. An **application** is a piece of software that accesses or queries a metadata model of a building in order to configure its operation, retrieve data, and discover actuatable resources. Our definition borrows from prior building application literature [16, 19, 21, 33] and encompasses a wide-range of existing building-related software such as (automated) fault detection and diagnosis, sequences of operation, measurement and verification, and building energy modeling.

A specification of required metadata — a *manifest* — must therefore capture a description of what the application expects to be present in the model. Manifests may be composed from many sources, including customer specifications, regulatory requirements, industry recommendations, and equipment specification (spec) sheets. Formally, a **manifest** M is a function that takes a model G as an argument and returns true or false if the model satisfies the predicates encoded in the manifest:

$$M(G) \rightarrow \{\texttt{true}, \texttt{false}\}, G'$$

If the check returns false, a manifest can also return optional metadata G' (specifically, a SHACL validation report as defined in [1]) specifying what is deficient about the model. Our incremental model creation algorithm (§5) uses this metadata to generate an efficient plan for creating a semantically sufficient semantic metadata.

There are two ways of structuring the manifest: query-oriented and shape-oriented.

3.2.1 Query-Oriented Manifests. First, a manifest can consist of a set of queries[1]. SPARQL is the standard query language for RDF models [29]. If all of the queries return results when evaluated against the input model, then the model passes the check and the manifest is satisfied. These types of manifests are easier to author and consume because they use the same queries as the application itself. A fundamental limitation of this query-oriented model is that query evaluation cannot identify the parts of the graph that *do not* satisfy the query predicate. This makes it more difficult to describe missing or incorrect metadata (a critical feature explored in §5) and also gives no guarantees that an application deployment is correct.

3.2.2 Shape-Oriented Manifests. A manifest can also consist of a set of *shapes*. A **shape** is a set of conditions expressed in the form of an RDF graph according to the SHACL standard [1]. The SPARQL and SHACL languages have similar expressive power; however, there are two advantages of using the shape-oriented approach over the query-oriented approach. First, the declarative nature of SHACL conditions and constraints makes it possible to identify exactly what is deficient or incorrect about a particular subgraph. This provides an avenue by which suggestions or improvements can be provided to the model author. §5 demonstrates how we capitalize on this feedback to further automate model creation. Second, SHACL-based constraints are more composable than their equivalent statements in SPARQL. SHACL shapes can be named, enabling references between shapes, and parameterized, enabling customized behavior based on further conditions. This composability enables our model authoring approach to reason about *collections* of application requirements, rather than dealing with each application individually. Therefore, we focus our discussion of application requirements on SHACL-based manifests.

3.2.3 SHACL Shapes. While a full description of the SHACL language and its extensions is beyond the scope of this document, we outline the essential structure and behavior of the language as it applies to application requirements. SHACL is a language for validating RDF graphs against a set of constraints and conditions and for generating new information about RDF graphs with rules. Constraints, conditions, and rules are grouped into *shapes*, which are expressed as RDF graphs.

[1]This is the formulation adopted in the original Brick paper [5]

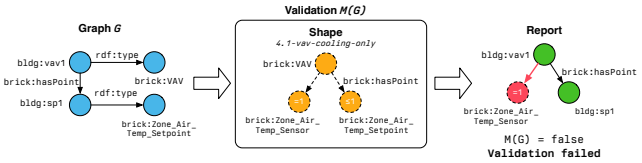

Figure 3: SHACL shape for part of a VAV system configuration being used to validate a Brick model

A shape contains a *target*: a specification of which nodes in an RDF graph the shape applies to. Only matching nodes will be subject to the rules, constraints, and conditions defined by the shape. The SHACL language allows shapes to require the presence or absence of relationships between certain kinds of nodes (e.g. "all air handling units [AHUs] must have a downstream variable air volume [VAV] terminal"), place bounds on the cardinality of certain relationships (e.g. "buildings must have at least one electric meter"), and other structural and semantic constraints. Although not the focus of this paper, SHACL can also generate new information in an RDF graph, e.g. classifying all AHUs with both hot and cold ducts as "dual-duct" AHUs.

Shapes can refer to other shapes, and can leverage the many common boolean operators such as `and`, `not`, `or`, and `xor`. This permits the expressive composition and reuse of shapes; we leverage this feature to formally define semantic sufficiency below (§3.3).

3.2.4 Example: ASHRAE Guideline 36 Point List. We briefly illustrate some essential SHACL features by formulating an application manifest for a industry-recognized sequence of operations. ASHRAE Guideline 36 [17] defines point lists of input/outputs that must be presented to the controller for the sequence of operations to operate. Figure 3 illustrates a subset of the *VAV Terminal Unit - Cooling-Only* configuration from Guideline 36 with a shape named `4.1-vav-cooling-only`. The shape constrains the type of the equipment and the BMS points associated with that equipment.

3.3 Model Verification

Using the above formulation of application requirements as shape-oriented manifests, we now formalize how sets of application requirements can be composed to verify whether a metadata model satisfies the requirements, i.e. whether it is *semantically sufficient*.

We refine our earlier definition of a manifest M as a set of shapes:

$$M = \{S : S \in \mathcal{S}\}$$

where $\mathcal{S}$ is the set of all SHACL shapes. This simple definition gracefully permits the composition of manifests: to specify that a semantic metadata model should contain the information for two applications with manifests M_1 and M_2, we simply create a new manifest that is the union of the shapes in each manifest:

$$M_{\text{unified}} = M_1 \cup M_2 = \{S : S \in M_1\} \cup \{S : S \in M_2\}$$

We can now formally define semantic sufficiency. A model (RDF graph), G, is *semantically sufficient* with respect to a set of manifests $\mathbf{M}$ when it satisfies all of the constraints (SHACL shapes) contained in the union of those manifests:

$$\bigcup_{\mathbf{M}}(G) = \text{true}, G'; |G'| = 0$$

3.4 Discussion

Semantic sufficiency closes the loop between creating a semantic metadata model and using that model to support real applications. This facilitates the adoption of semantic metadata by making the structure, contents, and uses of that metadata more concrete.

With semantic sufficiency, building stakeholders can place stricter and more descriptive requirements on the metadata delivered during building design, construction, and operation such as BMS installation or commissioning. It also allows stakeholders to automatically verify that the correct metadata has been delivered. Standardized representations of application metadata requirements also make it possible to compare vendors for a particular application and to make more precise cost-based decisions on what applications to install. For example, the estimated savings through installing an advanced control sequence can be balanced against the cost of commissioning the BMS points required to support that control sequence. Lastly, making application metadata requirements transparent makes it easier to re-use metadata between applications, ultimately leading to lower commissioning and integration costs.

4 TEMPLATES

Manifests express the semantic metadata requirements of applications, but they do not directly inform the creation of semantic metadata models. To this end, we introduce *templates*, which abstract away the piecemeal "triple-by-triple" approach to authoring RDF models to simplify and accelerate the model development process. At a high level, templates are functions that generate graphs. The *evaluation* (or *instantiation*) of a template produces part of a building graph in a consistent manner, allowing model authors to avoid manually creating the graph. Figure 4 contains an example of a simple template. In this section, we formally define templates and the operations they support.

4.1 Preliminaries

A template $T(P, G, D)$ is characterized by a set of parameters P, the body of the template G, and a set of dependencies D. The set of all templates is given by $\mathcal{T}$.

We write the parameters of a particular template T as T_P. Parameters are the *inputs* to the template and are required unless explicitly marked as optional. The body is an RDF graph that will be output by the template, with at least $|T_P|$ labeled "holes" corresponding to where the template parameters will be substituted in.

Dependencies are inter-template references that enable template re-use and composition. A dependency $d(n, B) \in D$ is parameterized by a name n and a set of bindings B. A dependency's name $d_n \in \mathcal{T}$ is a pointer to the template dependency, denoted as T^n. A binding $b \in T_P$ is an association $b_{\text{dependency}} \rightarrow b_{\text{dependent}}$, where $b_{\text{dependency}} \in T_P$ and $b_{\text{dependent}} \in T^n_P$. Intuitively, the binding associates parameters in the dependency to parameters in the dependent (the template owning the dependency). The set of bindings d_B for the dependent template must be a subset of the dependent's parameters: $d_B \subseteq T_P$.

```
vav:
  body: >
    {name} a brick:VAV ;
        brick:hasPoint {temp}, {sp}, {flow} .
  dependencies:
  - name: sup-air-temp
    args: {"temp": "name"}
sup-air-temp:
  body: >
    {name} a brick:Supply_Air_Temperature_Sensor ;
        brick:hasUnit unit:DEG_C .
```

Figure 4: Simple template defining a VAV with some points. One of the template dependencies is also listed.

4.2 Template Usage

Evaluation is the process by which a template produces an RDF graph. The *evaluation* of a template is given by the function

$$T(B) \rightarrow \begin{cases} T'(T_P - B, D') & \text{if } |B| < |T_P| \\ G & \text{if } |B| = |T_P| \end{cases}$$

that accepts a set of *bindings* (B) to 1 or more parameters. A binding $b(p, e) \in B$ is an association between a parameter $p \in T_P$ and an RDF node or property $e \in \mathcal{I} \times \mathcal{B} \times \mathcal{L}$. $\mathcal{I}$ is the set of RDF IRIs, $\mathcal{B}$ is the set of RDF blank nodes, and $\mathcal{L}$ is the set of RDF literals.

If all of the template parameters (T_P) have a corresponding binding, then template evaluation returns an RDF graph G (case 2 above). If there are fewer bindings than parameters (case 1 above), then evaluation returns a new template T' that adopts the remaining parameters and adjusts the template body T_G to incorporate the bindings provided during evaluation.

By reducing the problem of creating graphs to passing parameters to functions, it becomes possible to support various input modalities and formats. One goal of standardizing a template structure is to provide a common basis for future innovation in authoring metadata models. We describe three existing input modalities supported by our reference implementation:

Direct evaluation: Templates can be evaluated directly by software packages such as the reference implementation described in §7. This allows the population of a building model to be done in tandem with other processes, such as scraping a BMS network, applying OCR techniques to building plans (blueprints), or translating from existing digital building representations.

Spreadsheets and tables: The input parameters for a template can be treated as the columns of a tabular schema in a spreadsheet or relational database. A simple software shim can read the table and evaluate the template over each row of arguments, thereby producing parts of a building model. Abstracting template inputs behind a table offers a familiar and more-easily understandable interface to model authors.

Web forms: Templates may also be automatically transformed into web forms, permitting interactive creation of semantic metadata models. This interface can be augmented by providing autocomplete on inputs and continual model verification to reduce the chance of mistakes.

4.3 Template Bodies

A template body T_G is a *connected* RDF graph consisting of one or more triples, at least one of which contains a parameter called `name`. We require template bodies to be connected to ensure that any composition of templates results in a connected graph. The `name` parameter acts as a common join key for composing templates.

Consider the following template body containing four parameters: `name`, `x`, `y` and `z`:

```
{name} a brick:VAV ;
  brick:hasPoint {x}, {y} ;
  brick:hasPart {z} .
```

The parameters `x`, `y`, and `z` relate to `name` in specific ways, but the body itself does not give us any additional information on what they might be. This can be mitigated in part by giving descriptive names to the parameters (e.g., `sat` instead of `x`). Ultimately, *explicit* information about these parameters is contained within the template's *dependencies* T_D, which will relate the `name` parameter of each dependency template to a parameter in this dependent template. Templates can also depend on each other using other parameters beyond `name`.

To facilitate authoring and composing complex templates, we develop a deterministic inlining algorithm that produces predictable parameter names. This allows template authors to specify relationships between recursively included dependencies. A full description of the algorithm is beyond the scope of this paper.

Templates are nominally written as YAML documents with text-serialized RDF graphs as the body (Figure 4), but we are also developing syntactic sugar for common patterns such as point lists.

We now describe how templates contribute to the process of creating a semantically sufficient metadata model.

5 INCREMENTAL MODEL CREATION

By structuring both the metadata requirements of applications (via *shapes*) and common graph patterns (via *templates*), it becomes possible to optimize the model creation process by directing the author's effort towards adding the metadata that is the most useful for the most applications. We begin by introducing an intuitive but naïve approach to creating a semantic model, then introduce three complementary optimizations that both reduce the manual effort incurred by the author and increase the utility of intermediate stages of model creation.

5.1 Naïve Template-Driven Model Creation

We describe basic workflow for creating semantic metadata models using templates, as illustrated in Figure 5. First, a manifest must be procured that guides the model creation process. Manifests may come from anywhere, such as provided by a building owner or facility manager to the model author, who may work for a system integrator.

Given the manifest, the model author browses a set of libraries for relevant templates. The search for templates is driven by both the semantic requirements of the eventual model as well as the author's knowledge of what devices and configurations are actually available in the target building. Once the author selects a template they provide a set of bindings to evaluate the template into part of

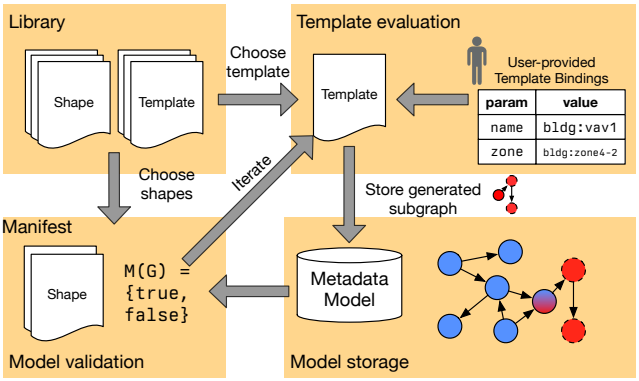

Figure 5: Using a library of shapes and templates to create a metadata model and verify its semantic sufficiency with respect to a manifest.

```
unit-with-occupancy:
  body: >
    {name} a brick:Terminal_Unit ;
        brick:feeds {zone} .
  dependencies:
  - name: zone-with-occupancy
    args: {"zone": "name"}
zone-with-occupancy:
  body: >
    {name} a brick:HVAC_Zone ;
      brick:hasPoint {occ} .
  dependencies:
  - name: https://brickschema.org/schema/Brick#Occupancy_Sensor
    library: https://brickschema.org/schema/1.3/Brick
    args: {"occ": "name"}
```

Figure 6: Two templates for creating a terminal unit with an occupancy sensor in its related zone.

the building metadata model. The model author may provide all of the bindings required by the template, but may also leave some of the template parameters blank if that information is not known at that time. As an example, this may happen when the author is modeling the HVAC system, but does not yet know the names of the BMS points for the comprising devices.

The author merges the metadata model produced by this single template evaluation into the existing building metadata model. The author can then verify the semantic sufficiency of the model by evaluating the manifest on the model. If the manifest is satisfied (i.e., $M(G) \rightarrow \texttt{true}$) then the author is done; otherwise, the author must return to the template library to patch the model with missing information. The author can also add semantic metadata to the model directly without going through template evaluation. The author repeats this iterative process until the metadata model is semantically sufficient with respect to the manifest.

5.2 Subgraph Monomorphism Search

We first consider the problem of how to avoid redundant inputs during template evaluation. When a model author selects a template to instantiate in a semantic model, there is a possibility that some of the metadata produced by the template already exists within the model. The model author must either query the existing model for existing metadata that can be provided to the template, remember past inputs provided to previously instantiated templates, or otherwise risk the inclusion of duplicate metadata.

Consider a model author attempting to instantiate the template in Figure 6 and add the resulting graph to the Brick model in Figure 1. To reduce the configuration burden on the model author, we wish to determine how much of the `unit-with-occupancy` template already exists within the model and remove the corresponding redundant inputs from the template. This problem reduces to finding *subgraph monomorphisms* of the building metadata model.

5.2.1 Preliminaries. To define subgraph monomorphisms, we reframe our semantic metadata models as directed, labeled graphs $G(V, E, L)$ where V is the set of vertices (*subjects* and *objects* in RDF), E is the set of edges (*predicates* in RDF) and L is the set of labels on those components. A graph $G'(E', V', L')$ is a *subgraph* of $G(E, V, L)$ if $E' \subseteq E$, $V' \subseteq V$ and $L' \subseteq L$. A node-induced subgraph G' of G has $N' \subseteq N$ and E' is *the* subset of edges in E that relate nodes in N'. A subgraph monomorphism G' of G has $N' \subseteq N$ and E' is *a* subset of edges in E that relate nodes in N'.

We can now formally define the problem. Given a building graph G and a template T, our goal is to find the largest subgraphs of G that are monomorphic to subgraphs of T (specifically, T_B). These subgraphs correspond to the parts of T that already exist in the building metadata graph and do not need to be re-evaluated by the author. In our example above, the largest subgraph would be:

```
@prefix brick: <https://brickschema.org/schema/Brick#> .
@prefix ex: <urn:bldg#> .
ex:vav1 a   brick:VAV ;
    brick:feeds ex:zone1 .
ex:zone1    a   brick:HVAC_Zone .
```

We also define the notion of *semantic compatibility* for comparing nodes and edges in a graph. Two nodes are semantically compatible:

(1) If *both nodes are* `owl:Class`, then the nodes are semantically compatible if one is a `rdfs:subClassOf` the other
(2) If *both nodes are instances*, then the nodes are semantically compatible if their types are covariant
(3) If the nodes refer to the *same entity* (same IRI or value), then they are semantically compatible
(4) Otherwise, the nodes are **not** semantically compatible

We use the notation $\text{SC}(n_i, n_j), n_i, n_j \in V$ to indicate that two nodes are semantically compatible, and $\overline{\text{SC}}(n_i, n_j), n_i, n_j \in V$ to indicate that they are not. This definition generalizes easily for edges as well as vertices by replacing the first condition with:

(1) If both edges are `rdf:Property`, then the edges are semantically compatible if one is a `rdfs:subPropertyOf` the other

and eliminating the second condition. Finally, if the first condition is true for a pair of nodes or edges, then $n_i < n_j$ if one is a subclass (or subproperty) of the other.

5.2.2 Search Algorithm. We can motivate the intuition of the algorithm as follows. Consider first the set of subgraphs of G that are monomorphic to T (*not* subgraphs of T). Each of these subgraphs necessarily corresponds to a fully evaluated template; that is, each node (parameter) in the template graph has a corresponding value in the subgraph of G. This is helpful for telling the user what is in the building graph that already matches the template.

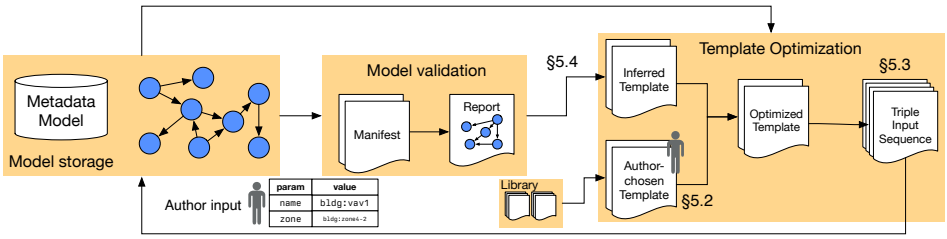

Figure 7: The optimized workflow for incremental model creation

Template Parameter	Building Graph Node
{unit}	ex:vav1
{zone}	ex:zone1
{occ}	N/A

Table 1: A monomorphism mapping between the templates in Figure 6 and the model in Figure 1

It may be the case that there are parts of the building graph that match *part* of a template, but still need some input to be complete. To find those, we search for subgraphs of G that are monomorphic to *each subgraph of* T. If we find such a monomorphism, we can then look at what nodes of T are in the template, but not the monomorphism. These "unbound" nodes correspond to the template parameters that need additional input. Table 1 contains the monomorphism mapping for our running example; note that the occ parameter does not have a corresponding mapping and thus requires input from the model author.

We adopt the VF2 algorithm [11] for finding subgraph monomorphisms, but wrap the execution of the algorithm in an additional loop through all possible subgraphs of T. Because the base VF2 algorithm only incorporates syntactic constraints that look at the structure of the graph around a node, we additionally incorporate optional *semantic requirements* for determining if two nodes are *semantically compatible*. If two nodes are semantically compatible, then they can be mapped to one another as part of a valid monomorphism. We use semantic compatibility checks to compare nodes and edges in the model G and the template body T_B:

$$\mathrm{SC}(n_i, n_j), n_i \in G, n_j \in T_B$$

We use the subgraph monomorphism process to provide the model author with a flavor of "autocomplete" when instantiating templates to include in a metadata model. This reduces the number of redundant inputs (there is no need to recreate a terminal unit in the model when one with the correct name already exists) and eliminates the need to manually search through a model for work has already been done.

5.3 Greedy Incremental Template Evaluation

The model authoring processes described above assume that authors evaluate templates in their entirety, without taking into account the priority or utility of those templates. This may require authors to expend valuable effort inputting metadata that is only required for a small number of applications, instead of metadata that is used by a broader set of applications.

Here we introduce our second model creation optimization, which adopts a greedy approach. Given a set of templates that an author wishes to instantiate within a metadata model, our optimization produces a sequence of metadata inputs that prioritizes commonly-used metadata and postpones unique metadata that only enables a handful of applications. Because template bodies are graphs, this sequence takes the form as a list of RDF triples.

To generate this sequence, we first construct a histogram of triples across all of the template bodies. Each bucket of the histogram contains triples that are semantically compatible. Two triples t_a = <s_a, p_a, o_a> and t_b = <s_b, p_b, o_b> are semantically compatible if their pairwise components are semantically compatible, that is:

$$\mathrm{SC}(t_a, t_b) \text{ iff } \mathrm{SC}(s_a, s_b) \wedge \mathrm{SC}(p_a, p_b) \wedge \mathrm{SC}(o_a, o_b)$$

Intuitively, this means that two triples appear in the same bucket if they could be substituted into each other's templates without loss of correctness. The sequence of triples presented to the model author is the most "specific" triple in each of the histogram buckets, in decreasing order of bucket size. The most specific triple is the *minimum* triple by the < operator defined in §5.2.1. This algorithm ensures that a model author does not need to evaluate *all* of the suggested templates in order to have a useful model. Certain prefixes of the produced sequence will contain enough metadata to support certain applications. This algorithm also eliminates redundant inputs between templates.

5.4 Incorporating Validation Feedback

Our last optimization closes the iterative loop in Figure 5 by inferring from the semantic sufficiency check a set of templates whose evaluation fixes the metadata model.

The semantic sufficiency check corresponds to the SHACL validation process [1]. Validation produces a *report*: an RDF graph that contains a set of validation *results*. A result describes a condition or constraint that was violated in the input graph and identifies where in the graph that violation occurred (the *target* of the result).

Our approach groups the validation results by their target, and then transforms the set of constraint violation descriptions into the minimal template whose instantiation would reconcile the failed constraint. This is possible because the SHACL standard defines a common language for describing the violations; these can be "inverted" into an RDF graph that becomes the generated template's body. The generated template incorporates the result target in its

definition so it is clear which part of the building model is being edited. The body of the template contains parameterized triples that reconcile all constraint violations for the corresponding target.

5.5 Optimized Incremental Model Creation

We now assemble the three optimizations above into an incremental model creation workflow. The templates inferred from the semantic sufficiency validation can be subject to the two optimizations above. Figure 7 illustrates the application of the three optimizations above to the base model creation process described in Figure 5.

First, the semantic sufficiency check produces a validation report. Interpreting the validation report (§5.4) produces a set of reconciling templates. The subgraph monomorphism search method (§5.2) transforms any author-provided templates into partially-evaluated templates that account for existing metadata that is already in the model. These two batches of templates can then be transformed into a sequence of inputs (§5.3) that prioritizes the utility of intermediate stages of model development. The workflow and the above optimizations are generic to any RDF-based ontology.

6 EVALUATION

To evaluate the utility and efficacy of our proposed approach to semantic model creation, we first analyze a public dataset of semantic metadata models to verify that there is effort that can be saved. Then, we present a case study constructing a metadata model to support the deployment of advanced sequences of operation in a standard reference building.

6.1 Measuring Redundant Graph Structures

The utility of the template-based approach to authoring metadata models hinges on the amount of redundant structure within graphs that can be automated away with templates. Here, we demonstrate the potential utility of our approach by measuring the frequency of redundant structures in a public dataset of 46 Brick models from the Mortar testbed [16].

To simplify this task, we consider a common motif in building metadata models: an entity and its immediate properties and relationships. Examples of this motif include: a thermostat, its location and its related points; a room, its area, volume, and associated floor and HVAC zone. We call this pattern an entity's "neighborhood".

For each graph, we determine the set of entities by querying the graph for all instances of Brick classes. We calculate the neighborhood for each entity by first retrieving the diameter-2 subgraph rooted at the entity (i.e., all of the triples for which the entity is the subject). We then rewrite this graph by aggregating nodes by their types in order to produce a "class graph" of the entity's neighborhood. We use the graph isomorphism algorithm described in [26] to determine if two class graphs are equivalent. We apply this process to all of the graphs in the Mortar dataset to compute the histogram of isomorphic class graph frequency within each graph.

Figure 8 illustrates the distribution of *distinct* motifs of a given size across the Mortar dataset. This figure does not include motifs with fewer than 4 triples to focus on the potential of motifs to generate larger portions of a metadata model. We can draw two conclusions from the distribution. First, there are a large number of distinct motifs that could be represented as templates. Second,

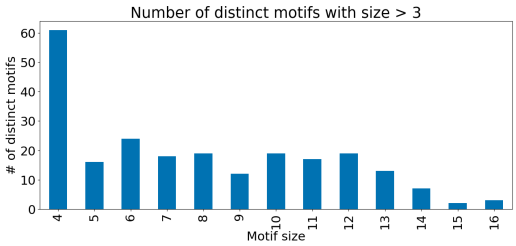

Figure 8: The frequency of motifs with at least 4 triples across the Mortar dataset. There are 564 motifs of size 3.

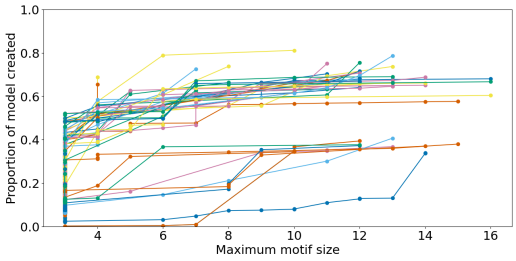

Figure 9: Cumulative proportion of each model that can be represented as neighborhood motifs of at most a given size

21% of the distinct "neighborhood" motifs are at least 4 triples. This suggests that such nontrivial motifs could dramatically simplify creating models of complex entities.

Figure 9 shows the cumulative proportion of each model in the Mortar dataset that can be represented by motifs of at most a given size. The results demonstrate that at most 80% of Mortar Brick models can be represented as neighborhood motifs (considering more kinds of motifs will raise this number). The figure also shows that there is no easy determination of how much of a model can be made composed of motifs of a given size. For some models, small motifs (4 or fewer triples) will make up more than 40% of the graph; other models require a broader range of motifs. Metadata models have many different kinds of redundancy that are not always easily characterized. This motivates the need for an approach such as ours that can take advantage of structural and semantic redundancy in a model to reduce the authoring effort.

6.2 Removing Redundant Inputs

Having demonstrated the potential utility of our approach, we not evaluate the ability of the proposed approach to reduce redundant inputs during model creation. We consider *manual, from-scratch* model creation as our baseline; despite the need for model authoring tools, metadata models are still created manually in current practice.

An *input* is a piece of information input by the model author. This may be the name of an equipment or device, the engineering units for a point, the relationship between a terminal unit and a zone, or any other node or edge in a metadata model. Specifically, the author-provided inputs for a model consist of: (a) the names of entities in the model (each triple containing an entity counts as a different input), and (b) the edges that connect to or from an entity.

```
@prefix brick: <https://brickschema.org/schema/Brick#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix constraint: <anonymized-for-review> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix g36: <https://data.ashrae.org/nonexistent/guideline36> .
@prefix : <urn:medium-office-brick-constraints/> .
# metadata referencing external libraries and concepts
: a owl:Ontology ;
  owl:imports <https://brickschema.org/schema/1.3/Brick> ;
  owl:imports <https://data.ashrae.org/nonexistent/guideline36> .
:ahu-count a sh:NodeShape ;
  sh:message "The model must contain exactly 3 AHUs" ;
  sh:targetNode : ; # indicates a reference to the current model
  constraint:exactCount 3 ;  constraint:class brick:AHU . # custom constraint
:ahu-feeds-rvavs a sh:NodeShape ;
  sh:message "AHUs feed 5 VAVs; all VAVs must support G36 SOO 4.1" ;
  sh:targetClass brick:AHU ;
  sh:property [
      sh:path brick:feeds ; sh:class g36:vav-with-reheat ;
      sh:minCount 5 ; sh:maxCount 5 ] .
```

Figure 10: A simple manifest for creating a Brick model of the DOE Medium Commercial Office reference building.

To evaluate the ability of our proposed system to reduce author inputs and maximize the utility of the resulting metadata model, we consider a case study of creating a Brick model for the Department of Energy's Medium Office Commercial Reference Building [12]. The reference building contains three multi-zone AHUs, each of which feeds a number of VAV terminals with reheat (VAVRs). shFor the evaluation, we construct a manifest specifying that all VAVRs must support the relevant Guideline 36 (G36) sequence of operations [17] (Figure 10). The manifest contains two shapes; one for the two constraints above. The shapes are simple because they import pre-defined shapes from an external library. For clarity, we also require the model to have five VAVRs per AHU (this is not specified in the reference building definition).

We compare the number of required author inputs across five modeling scenarios. In these description, the *existing* model is a Brick model containing the declarations of AHUs and RVAVs, but none of their properties, points or relationships.

(1) naïve, manual model creation, from scratch
(2) naïve, manual model creation, from existing model
(3) using a provided G36 template for VAVRs, no existing model
(4) using a provided G36 template for VAVRs, from existing model
(5) using templates *inferred* from the existing model

The results of this investigation are in Table 2. The baseline of 303 author inputs (row 1) incorporates names of AHUs, VAVRs, their points, and knowledge of which AHUs are connected to which VAVRs. If the model author chooses to create the model from the Guideline 36 template provided as part of our reference implementation, they only need to provide 158 inputs. These inputs are also scaffolds: they tell the author exactly what needs to be provided.

The final two rows have the same number of inputs but correspond to very different model development experiences. In the first (row 4), the author manually chooses the template to be instantiated in the model and uses the monomorphism search (§5.2) to pre-populate those templates with existing information. In the last row (row 5), the author uses the validation report to automatically determine the missing semantic information with respect to the manifest (Figure 10) and infer a set of templates (§5.4). This does not require any familiarity with the available templates, but prompts the author for the same information.

Modeling Approach	# Inputs	% Red.
Manual creation, from scratch	303	0%
Manual creation, from existing model	285	6%
Existing templates, from scratch	158	52%
Existing templates, from existing model	140	54%
Inferred templates, from existing model	140	54%

Table 2: The number of author-provided inputs required to create the medium office reference building model using each of the proposed techniques and optimizations. The last column records the % reduction in the number of inputs compared to manual model creation.

From these results we can conclude that the proposed techniques reduce the number of required author inputs and make effective use of existing information in the model and manifest.

7 IMPLEMENTATION

We have constructed reference implementations of templates (§4), shapes (§3) and the model creation workflow (§5) to encourage community involvement and to promote further innovation. The implementation is open source, permissively licensed, and is available online: https://github.com/NREL/BuildingMOTIF. The reference implementation takes the form of a software development kit (SDK) designed to be integrated into other user-facing pieces of software. The SDK provides APIs for storing, managing, and verifying collections of templates, shapes, and metadata models; it also implements the iterative model creation workflow (§5).

The SDK is agnostic to the source of metadata. We envision the SDK complementing existing metadata ingestion tools that infer metadata from BMS point labels [10, 20] or translate from existing digital representations such as BIM, energy audits, or other semantic metadata models [15].

The ability to create, share, and exchange templates and shapes is critical to creating an ecosystem that encourages and rewards investment from the broader community. To encourage distribution and reuse, we group templates and shapes together into *libraries*. Libraries have globally unique names (such as a DOI or URI) to facilitate organization and to enable cross-library references.

Cross-library references allow templates or shapes in one library to refer to templates or shapes in another library as dependencies. This allows library authors to incorporate existing templates and avoid redefining (potentially complex) base concepts, thus encouraging the reuse of such specifications. External template references can be done by including a `library` key in a template's dependencies (Figure 4). External shape references can be done by referring to the URI of the dependency shape.

We have implemented some libraries to prove out their utility and expressiveness. Our reference implementation includes a library containing system configurations and equipment types defined in ASHRAE Guideline 36. The library includes templates and shapes for variable air volume boxes in both cooling-only and reheat configurations and several different air handler unit configurations. Each of these templates corresponds directly to fault detection algorithms also defined in Guideline 36.

8 CONCLUSION

In this paper we have introduced *semantic sufficiency*, a new approach to creating semantic metadata models that uses formal descriptions of application requirements to inform an incremental model creation workflow that directs author effort towards useful metadata models. We also introduce and formalize *templates*, which automate the construction of common patterns within metadata models. These contributions lower the barrier to adoption of semantic metadata models, ultimately helping to enable the widespread deployment of intelligent building applications.

ACKNOWLEDGMENTS

This work was authored by the National Renewable Energy Laboratory, operated by Alliance for Sustainable Energy, LLC, for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308. Funding provided by U.S. Department of Energy Office of Energy Efficiency and Renewable Energy Building Technologies Office, specifically through the Semantic Interoperability Research and Development project, CPS agreement number 34579. The views expressed in the article do not necessarily represent the views of the DOE or the U.S. Government. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this work, or allow others to do so, for U.S. Government purposes. We also thank Cory Mosiman and Austin Viveiros for their early contributions to this work.

REFERENCES

[1] 2017. *Shapes constraint language (SHACL).* Technical Report. W3C. https://www.w3.org/TR/shacl/

[2] 2021. *Project Haystack.* http://web.archive.org/web/20210111211811/https://project-haystack.org/

[3] 2022. *Data Clearing House.* https://web.archive.org/web/20220728042356/https://dataclearinghouse.org/

[4] American Society of Heating, Refrigerating and Air-Conditioning Engineers. 2018. ASHRAE's BACnet Committee, Project Haystack and Brick Schema Collaborating to Provide Unified Data Semantic Modeling Solution. http://web.archive.org/web/20181223045430/https://www.ashrae.org/about/news/2018/ashrae-s-bacnet-committee-project-haystack-and-brick-schema-collaborating-to-provide-unified-data-semantic-modeling-solution.

[5] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David Culler, Rajesh Gupta, Mikkel Baun Kjærgaard, Mani Srivastava, and Kamin Whitehouse. 2016. Brick: Towards a Unified Metadata Schema For Buildings. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments* (Palo Alto, CA, USA) *(BuildSys '16)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/2993422.2993577

[6] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah McGuinness, Peter Patel-Schneijder, and Lynn Andrea Stein. 2004. *OWL Web Ontology Language Reference.* Recommendation. World Wide Web Consortium (W3C). See http://www.w3.org/TR/owl-ref/.

[7] Harry Bergmann, Cory Mosiman, Avijit Saha, Selam Haile, William Livingood, Steve Bushby, Gabe Fierro, Joel Bender, Michael Poplawski, Jessica Granderson, and Marco Pritoni. 2020. Semantic Interoperability to Enable Smart, Grid-Interactive Efficient Buildings. (12 2020). https://doi.org/10.20357/B7S304

[8] Keith Berkoben, Charbel El Kaed, and Trevor Sodorff. 2020. A Digital Buildings Ontology for Google's Real Estate.. In *ISWC (Demos/Industry)*. 392–394.

[9] 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.

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

[11] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2001. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*. Citeseer, 149–159.

[12] M Deru, K Field, D Studer, K Benne, B Griffith, P Torcellini, M Halverson, D Winiarski, B Liu, M Rosenberg, et al. 2010. DOE Commercial Reference Building Models for Energy Simulation–Technical Report. *National Renewable Energy Laboratory: Golden, CO, USA* (2010).

[13] US EIA. 2082. Commercial buildings energy consumption survey. *United Sates Department of Energy* (2082).

[14] Gabe Fierro and Pieter Pauwels. 2022. *Survey of metadata schemas for datadriven smart buildings (Annex 81).* CSIRO, Australia.

[15] 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.

[16] Gabe Fierro, Marco Pritoni, Moustafa AbdelBaky, Daniel Lengyel, John Leyden, Anand Prakash, Pranav Gupta, Paul Raftery, Therese Peffer, Greg Thomson, et al. 2019. Mortar: an open testbed for portable building analytics. *ACM Transactions on Sensor Networks (TOSN)* 16, 1 (2019), 1–31.

[17] GUIDELINE 36-2021 2021. *High-Performance Sequences Of Operation For HVAC Systems.* Standard. American Society of Heating, Refrigerating and Air-Conditioning Engineers.

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

[19] 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*. 15–26.

[20] 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*. 1–10.

[21] 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*. 72–79.

[22] Henrik Lange, Aslak Johansen, and Mikkel Baun Kjærgaard. 2018. Evaluation of the opportunities and limitations of using IFC models as source of building metadata. In *Proceedings of the 5th Conference on Systems for Built Environments*. 21–24.

[23] Ora Lassila, Ralph R Swick, et al. 1998. Resource description framework (RDF) model and syntax specification. (1998).

[24] Shuheng Li, Dezhi Hong, and Hongning Wang. 2020. Relation Inference among Sensor Time Series in Smart Buildings with Metric Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 04 (Apr. 2020), 4683–4690. https://doi.org/10.1609/aaai.v34i04.5900

[25] Nicholas Long, Katherine Fleming, Christopher CaraDonna, and Cory Mosiman. 2021. BuildingSync: A schema for commercial building energy audit data exchange. *Developments in the Built Environment* 7 (2021), 100054.

[26] James P McCusker. 2015. *WebSig: a digital signature framework for the web.* Rensselaer Polytechnic Institute.

[27] Jyrki Oraskari, Madhumitha Senthilvel, and Jakob Beetz. 2021. SHACL is for LBD what mvdXML is for IFC. In *Proc. of the Conference CIB W78*, Vol. 2021. 11–15.

[28] 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 (Apr 2021), 2024. https://doi.org/10.3390/en14072024

[29] Eric Prud'hommeaux and Andy Seaborne. 2008. SPARQL Query Language for RDF. W3C Recommendation. http://www.w3.org/TR/rdf-sparql-query/ http://www.w3.org/TR/rdf-sparql-query/.

[30] Rasmussen, Lefrançois, Schneider, and others. 2021. BOT: the building topology ontology of the W3C linked building data group. *Semant. Web* (2021).

[31] Zixiao Shi, Guy R Newsham, Long Chen, and H Burak Gunay. 2019. 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*. 111–120.

[32] Martin G Skjæveland, Henrik Forssell, Johan W Klüwer, Daniel Lupp, Evgenij Thorstensen, and Arild Waaler. 2019. Pattern-based ontology design and instantiation with reasonable ontology templates. *A Higher-Level View of Ontological Modeling* 69 (2019).

[33] 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*. 1–8.