Shepherding Metadata Through the Building Lifecycle

Gabe Fierro
UC Berkeley
gtfierro@cs.berkeley.edu

Anand Krishnan Prakash
Lawrence Berkeley National Laboratory
akprakash@lbl.gov

Cory Mosiman
National Renewable Energy Laboratory
cory.mosiman@nrel.gov

Marco Pritoni
Lawrence Berkeley National Laboratory
mpritoni@lbl.gov

Paul Raftery
UC Berkeley
p.raftery@berkeley.edu

Michael Wetter
Lawrence Berkeley National Laboratory
mwetter@lbl.gov

David E. Culler
UC Berkeley
culler@cs.berkeley.edu

ABSTRACT

Many different digital representations of a building are produced over the course of its lifecycle. These representations contain the metadata required to support different stages of the building, from initial planning and design, to construction and commissioning, through operations, audits, retrofits and maintenance. However, because of differences in the semantics, structure and syntax of these representations, the metadata they contain is not *interoperable*. We present a novel method for leveraging these representations to create a unified, authoritative Brick metadata model for a building that can be continually maintained over the course of the building lifecycle. A simple synchronization protocol relays inferred Brick metadata from existing metadata sources such as gbXML, BuildingSync, Project Haystack and Modelica to a central integration server, which merges the metadata into a valid Brick model.

CCS CONCEPTS

• **Information systems → Ontologies**; *Data encoding and canonicalization*; *Information extraction.*

KEYWORDS

Smart Buildings, Data Integration, Metadata, Ontologies, OWL, RDF, Brick

ACM Reference Format:
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 *The 7th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys '20), November 18–20, 2020, Virtual Event, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3408308.3427627

BuildSys '20, November 18–20, 2020, Virtual Event, Japan

ACM ISBN 978-1-4503-8061-4/20/11.
https://doi.org/10.1145/3408308.3427627

1 INTRODUCTION

Many different digital representations of a building are typically produced over the course of its lifecycle. These representations contain — in some form — the *metadata* required to support different stages of the building, from initial planning and design, to construction and commissioning, through operations, audits, retrofits and repairs. As a result, different representations communicate different perspectives on the same building: the metadata required to support the construction of a building is different than the metadata required to manage schedules and control sequences.

Despite the increasing digitization of building metadata, there is often little reuse or effective communication of this metadata across stages of the building lifecycle. This is due in part to the difficulty of integrating the partially overlapping metadata from different sources, which may use different terminology or names for components, and provide varying levels of detail about the building and its subsystems.

At the same time, a renaissance has occured around the use of metadata to enable the agile deployment of building "applications" such as fault detection and diagnosis, measurement and verification and other analytics. Emerging standards such as ASHRAE Standard 223P [4], Brick [9], Project Haystack [3], BOT [29] and SAREF4BLDG [27] have embraced expressive data models for describing the metadata required to perform these applications.

However, a largely underserved question is how to create and maintain these metadata representations. In this paper we present a method for the *continuous* integration—"shepherding"—of existing metadata sources into a unified metadata model. We implement the method in a proof-of-concept system and demonstrate its operation over several metadata sources used at different stages of the building lifecycle.

1.1 Proposed Approach

We propose a method for creating a Brick-based representation of a building (a *Brick model*) from the different metadata representations across a building's lifecycle, and keeping that Brick model up-to-date with these other representations even as they change. The key insight of our approach is that a unified metadata model of a building does not need to capture *all* relevant metadata for every task of

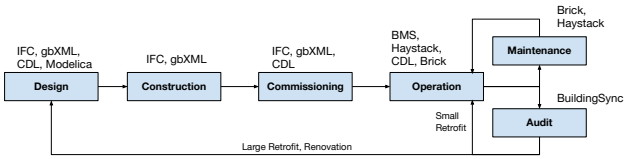

Figure 1: Different tools for different stages: Many different metadata standards and technologies are applied over the course of a building's lifecycle, but are relatively siloed and thus non-interoperable.

every stage of a building's lifecycle. Instead, the unified metadata model should capture the metadata needed to support data-driven applications and preserve investments in existing representations which support traditional applications.

We choose Brick as the unified representation of building metadata for three reasons. First, it has demonstrated success in capturing the semantic requirements of a wide array of building applications [9, 17]. Secondly, its broad vocabulary covers concepts in many different building subsystems, meaning it can express concepts also defined in other metadata representations. Lastly, because Brick is a formal ontology, it is possible to provide well-defined mapping between concepts in Brick and other metadata representations and extend Brick to cover new or unusual concepts [16].

Supporting the continuous integration of metadata into a unified Brick model enables several compelling features. First, applications developed against Brick can automatically reconfigure themselves to changes in the building by re-executing queries against the Brick model [17]. Second, the unified representation provides a means of automatically detecting when external metadata sources are incongruent or out of date, which is typically a tedious and manual process.

1.2 Contributions

An effective solution to maintaining a unified metadata model must import metadata from multiple existing representations, reconcile the differences between the imported metadata, infer missing properties, and keep the model up-to-date as the metadata representations evolve. The contributions of this paper are:

(1) a simple protocol for assembling Brick metadata extracted or inferred from established metadata sources
(2) a merge algorithm for detecting and reconciling differences between overlapping metadata sources
(3) an open-source proof-of-concept implementation[1], demonstrating integration of metadata from four different sources

Together, these contributions enable the creation and maintenance of an *authoritative* metadata model for a building that does not require the adoption of a single universal metadata standard throughout the building lifecycle. Because this authoritative model combines the metadata from a collection of sources, it can present a more comprehensive picture of the building than any individual source.

[1] Available at https://github.com/gtfierro/shepherding

2 BACKGROUND

We define several terms integral to the discussion and provide an overview of prevailing metadata representations used over the building lifecycle.

Definition 2.1 (Metadata Source). A metadata source is a data model, schema, standard or convention for representing building metadata that defines the structure, syntax and/or semantics of that metadata. *Structure* is the organization of metadata into data structures, objects and relationships. *Syntax* is the encoding and rules for how that metadata can be expressed and communicated. *Semantics* is the use of logical rules and statements to encode the "meaning" of metadata. Examples include the IFC 4.1 standard (which defines structure, syntax and semantics), Project Haystack v3.9.7 (which defines structure and syntax), and BACnet point names (which define syntax).

Definition 2.2 (Metadata Model). A metadata model is the content of a metadata representation for a building at a point in time. This reflects the fact that building metadata may change over time via updates made by tools or stakeholders.

Differences in the semantics, structure and syntax of models generally result in a lack of *interoperability* [37]. This limits the extent to which metadata from one stage of a building's lifecycle can contribute to the metadata for another stage. As a result, models are often not maintained after the task for which they were developed has completed [28].

There have been previous attempts to increase information sharing and reuse between stakeholders though a shared knowledge base [22] or by centralizing all data in a BIM model [37]. [30] demonstrates the semi-automated configuration of a building automation system by exporting BACnet objects from the BIM, enabling the exchange of information across the design, construction and operational stages of a building. Rather than adapting an existing representation to other stages of the building lifecycle, our approach translates these existing representations into a unified model. This allows existing representations to coexist with the unified model and encourages maintenance of those existing representations so that they can continue to support their traditional use cases while keeping the unified model in sync.

2.1 Prevailing Metadata Models for Buildings

We describe five representative metadata models from the primary stages of a building's lifecycle.

Green Building XML (gbXML) [1] is an XML schema for the exchange of building information modeling (BIM) data used for the design and construction of a building. This includes the enumeration and 3D geometry of the equipment and spaces in a building and associated sensors. In contrast to other BIM representations like Industry Foundation Classes [5], gbXML provides contextual information about a building's components by grouping related equipment and spaces together.

BuildingSync [24] is an XML-based schema designed to capture energy audit data in line with ASHRAE Standard 211 [6]. The standard requires the reporting of high-level operational parameters of the building (e.g. floor area, occupancy classification, operating hours), primary system information (e.g. heating, cooling, lighting,

process loads), historical energy consumption, benchmarking information, and target performance objectives. Energy auditors use this information to provide the building owner with recommendations for achieving energy reduction goals or mandates. Energy audits are often conducted at various points in a building's lifecycle; e.g. every 10 years for New York City buildings over 50,000 sq ft [38].

Modelica Buildings Library and Control Description Language (Modelica/CDL): Modelica [8] is a declarative, equation-based modeling language used to model engineered systems. Modelica defines modular objects which are coupled to each other through connector objects to form systems. Connectors can represent input/output ports for control signals, or physical ports such as for representing a flange of a valve through which fluid flows. The Modelica Buildings Library contains component and system models for building and district energy and control systems [34].

The Control Description Language (CDL) [32] is a subset of Modelica used to express control sequences for building automation systems in a vendor-independent format. CDL aims at enabling the digitization of the design, specification, deployment and verification of building control sequences such as ASHRAE Guideline 36 [7, 33]. CDL is the basis for the initiation of a new standard, ASHRAE Standard 231P, whose aim is to express building environmental control sequences in a control-vendor neutral computer language.

Project Haystack (Haystack) [3] is a popular tag-based data model that describes equipment and points (data sources) in buildings during the operational stage of a building. Haystack does not formally define how well-known concepts should be described. As a result, tag usage is inconsistent between Haystack models, which limits the interpretability of the resulting model and the extent to which it can be integrated with other metadata sources. Recent work ameliorates these issues by automatically aligning Haystack tags with existing formal definitions [16].

Finally, building management systems (BMS) often contain **vendor-specific and ad-hoc representations** which follow informal and inconsistent naming conventions, but may contain useful metadata for data-driven applications [11].

2.2 Brick: a *Lingua Franca*

Although the above metadata sources provide utility for specific stages of the building lifecycle, they do not carry across stages the information needed to author data-driven applications without losing crucial detail. Prior work has established that certain common sources of metadata alone do not capture the information required to run these applications [10]. Brick [9, 16] is a formal metadata representation designed to address this gap by targeting completeness (capturing the concepts required for applications), expressiveness (capturing the relationships and properties required for applications) and extensibility (the ability to describe new concepts as needed). This combination of features makes Brick a compelling metadata source for data-driven applications [17, 20]. Applications execute queries against a Brick model for the metadata they need to configure and operate.

Brick defines a comprehensive set of concepts for describing virtual, logical and physical "entities" in buildings, and a family of rich semantic relationships for describing the connections and

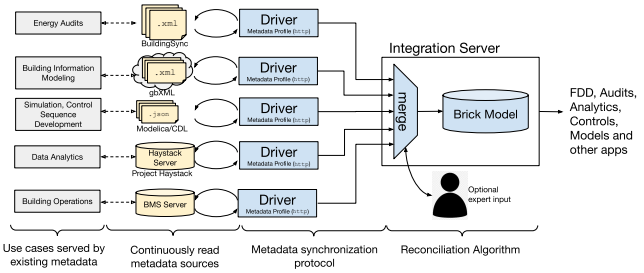

Figure 2: Overview of the proposed approach: *drivers* interface directly with existing metadata sources stored in local file systems, or accessed via file shares or networked services. Drivers continuously publish inferred Brick metadata to a central server, which produces a unified model.

associations between them. Because Brick defines concepts for multiple building subsystems including HVAC, lighting, electrical and spatial, there is often conceptual overlap between Brick and other metadata sources. This means that not only is there a path for creating Brick models from existing metadata sources, but that Brick can evolve to support new concepts and additional metadata without needing to "backport" these changes to other metadata representations. For example, although Brick does not describe geometry, IFC and gbXML descriptions of geometry can be associated with Brick entities.

2.3 Data Integration

The system architecture of the proposed solution takes its inspiration from the wrapper-mediator architecture for data integration [19, 35]. Wrappers — called *drivers* here — provide access to heterogeneous data sources, and mediators — a role fulfilled by the *integration server* — provides access to a unified view of the data reported by the wrappers. Because of the additional complication that our drivers can produce overlapping data, we also incorporate methods from the record linkage literature to determine which data relates to the same entities [36].

3 SHEPHERDING METADATA

We present an overview of the stages and metadata sources of the building lifecycle, and summarize the components of the proposed approach for continuously integrating the metadata sources used throughout the lifecycle.

3.1 Building Lifecycle

Each phase of the building's lifecycle involves the consumption or production of one or more metadata sources (Figure 1).

The design phase of a building, conducted using BIM, may produce IFC or gbXML models that are used during the construction phase of the building. During construction, this metadata may be used in conjunction with CDL descriptions of the building's sequence of operations to configure the BMS. Unstructured BMS metadata may be captured in a Brick or Haystack model to facilitate data analysis and to perform predictive maintenance. Other metadata sources such as BuildingSync may be used to conduct

energy audits before and after retrofits and repairs, which themselves may rely upon CDL, IFC or gbXML representations of the building's control loops and assets.

The differences in structure, syntax and semantics between the above metadata sources limit the extent to which their metadata can be re-used between stages of the building lifecycle. The content of metadata sources may change over time in response to repairs and retrofits or to add additional information to an evolving model. Other metadata sources, such as floor plans or BIM representations, may not be updated to reflect changes in the building. Enabling the continuous integration of disparate metadata sources into a unified and semantically rich model can facilitate and incentivize the re-use of metadata throughout the building lifecycle.

3.2 Continuous Integration Architecture

The proposed approach to continuous metadata integration, represented in Figure 2, has the following components.

A **driver** is a software process that produces Brick metadata from an underlying metadata source. For structured or standardized metadata sources such as BuildingSync and gbXML, this may be accomplished through direct translation of the source's concepts and structures to Brick. For less structured and ad-hoc sources such as Haystack and BMS labels, a driver may infer Brick metadata through a statistical or heuristic-based approach. The driver continually produces Brick metadata from the most recent version or "snapshot" of the underlying metadata source. The driver has no requirements for how the Brick metadata is produced or inferred, but must implement the **synchronization protocol** to transmit this metadata to the integration server.

The **integration server** is a logically centralized process that assembles Brick metadata from a collection of drivers and produces a unified Brick model representing the union of the collected metadata. Because different metadata sources are created at different stages of the building and by independent stakeholders, the Brick metadata produced by the drivers is likely to contain disagreements and inconsistencies, or may simply be out of date. To address this issue, the integration server incorporates a novel **reconciliation algorithm**—analogous to the "merge" operation in `git`—that attempts to resolve the differences between the metadata reported by the drivers. The integration server makes the unified Brick model available to applications and external services such as Mortar [17].

We formally describe the operation of each of these components, implement an end-to-end proof-of-concept, and explore the behavior of the approach on a set of representative metadata models and buildings.

4 METADATA SYNCHRONIZATION

Producing a unified metadata model requires assembling metadata from a variety of sources with differing structure, syntax and semantics. Drivers implement a **metadata synchronization protocol** which updates the integration server with the latest Brick metadata from a specific source. The metadata synchronization protocol decouples the method of inferring or producing Brick metadata from how that metadata becomes integrated into the authoritative model. This allows the proposed system to incorporate new metadata sources and novel methods of inferring Brick metadata.

```
{"id": "RTU-1",
 "raw": {
   "content": "<auc:Delivery ID=\"RTU-1\">
    <auc:DeliveryType>
      <auc:CentralAirDistribution>
        <auc:AirDeliveryType>Central fan</auc:AirDeliveryType>
        <auc:FanBased>
          <auc:CoolingSupplyAirTemperature>73</auc:CoolingSupplyAirTemperature>
        </auc:FanBased>
      </auc:CentralAirDistribution>
    </auc:DeliveryType>",
    "encoding": "XML"},
  "source": "BuildingSyncDriver",
  "source_version": "2.1.0",
  "timestamp": "2020-07-16T20:02:50",
  "protocol_version": "1.0.0",
  "triples": [["http://example.com/building#RTU-1",
               "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
               "https://brickschema.org/schema/1.1/Brick#Rooftop_Unit"]]}
```

Figure 3: Example record published by the BuildingSync driver, showing the original metadata (`raw`) and the inferred Brick metadata (`triples`).

4.1 Formal Definition of Brick Inference

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

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

$$D_i(m_i^t) \rightarrow \{e_1^t, e_2^t, \ldots, e_n^t\} \tag{1}$$

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

$$G_i^t = \bigcup_{j=1}^{n} T(e_j^t) \tag{2}$$

4.2 Metadata Profile

A driver implements a metadata *profile* providing a structured representation of the Brick metadata for the most recent metadata model from a particular source. The profile is a set of HTTP resources: The root resource (`/`) holds a list of entity ids with associated Brick metadata for the current metadata model, and a timestamp representing the version of that metadata model. The record resources (`/record/<id>`) hold the *record* associated with a given entity id. A record contains the following fields:

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

- `triples`: a list of RDF triples defining the Brick metadata for the entity

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

Figure 3 contains an example of a BuildingSync record for an HVAC delivery system named "RTU-1". Note that the original XML element contains additional metadata not conveyed in the produced Brick triples.

4.3 Metadata Synchronization Protocol

Drivers synchronize the contents of the profile with the integration server via the metadata synchronization protocol. This is analogous to the "merge" operation in `git`. The protocol operates over HTTP and consists of two request-response actions: `check` and `sync`.

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

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

The server saves all records in a local database. When the server performs the reconciliation algorithm to produce a unified metadata model (§5), it by default only considers the records corresponding to the most recent timestamp (version) per source. By extension, the server can also produce a unified metadata model for any point in the Brick model's history. This allows applications to access the history of changes in a building, but through the interface of a standardized, unified representation rather than an ad-hoc collection of diverse metadata sources.

The `triples` field can contain arbitrary Brick metadata to be relayed to the server. Typically this involves type information (`vav1` is a `brick:VAV`), system composition information (`vav1` is downstream of `ahu1`), telemetry association (`vav1` has temperature setpoint `temp_sp1`) and location information (`tstat1` is in `Room 410`). The triples may also define extensions to the Brick ontology, such as to describe additional properties of an unusual device or point.

4.4 Producing Brick Metadata with Drivers

We examine the effectiveness of the metadata profile and synchronization protocol by implementing proof-of-concept drivers for four common building metadata representations. For each driver, we explain the structural and semantic mapping between the metadata source and Brick. Our approach permits the development of separate drivers for different versions of each metadata source.

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

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

4.4.2 Project Haystack. The structure of a Haystack model has a straightforward mapping to Brick: each Haystack entity corresponds to one or more Brick entities. The generic links between Haystack entities (called `refs` in Haystack parlance) can be expressed with Brick relationships.

However, because the semantics of a Haystack model are not well-defined, there is no unambiguous and exhaustive mapping of Haystack metadata to Brick. As a result, the types of Haystack entities and relationships between them must be inferred through some external process [16]. The engine described in [16] fulfills this role by using tags associated with Brick concepts to infer the most likely class for a given set of Haystack tags. This engine has been incorporated into the Haystack driver, so that further updates and refinements to the method can be incorporated into the metadata integration process.

4.4.3 Modelica/CDL. Modelica and CDL models consist of a set of connected *objects*. This linked structure clearly identifies entities and the relationships between them, which closely resembles the structure of a Brick model. Modelica objects have classes, which supports the development of a mapping between Modelica classes and Brick classes. For example, every instance of the Modelica class `Buildings.Fluid.Sensor.Temperature` can be translated into a `brick:Temperature_Sensor` entity. The links between objects in a Modelica model can inform the choice of sequential (`brick:feeds`) and compositional (`brick:hasPart`) relationships between their corresponding Brick entities.

There are a few challenges in producing Brick metadata from a Modelica/CDL model. First, because Modelica is a general modeling language, it is possible for models to describe buildings using classes unknown to the driver. The driver establishes mappings for many of the common classes in the Modelica Buildings Library [34], but there is no guarantee that a Modelica model will use these classes. Second, due to Modelica's model encapsulation, contextual properties, i.e., how the objects relate to a larger system, need to be inferred. For example, for an instance of the class `Buildings.Fluid.Sensor.Temperature`, where it is located (e.g. exhaust air, return air, entering water, leaving water) need to be inferred from the system that contains the sensor.

4.4.4 gbXML. Drivers for BIM can produce Brick metadata about individual components, but inferring contextual relationships between those components is more difficult. Because BIM models focus on the geometry and physical connections of spaces and equipment, the representations may lack or obscure the contextual information needed during the operations and maintenance stages of the building. For example, although several versions of the IFC standard enumerate possible physical and contextual properties of fans, these details may not be included in an IFC model or must be inferred by traversing the connections between other elements in the model. gbXML represents equipment and high-level contextual relationships more explicitly than IFC, and the numbers in Figure 8 show promise.

A significant challenge for the production of Brick metadata from a BIM model is variability in how BIM models are expressed. BIM standards are designed to be flexible and extensible, which can result in a lack of consistency and thus interoperability [14, 25]. Representing BIM models using semantic web technology will allow the driver to more easily validate and inspect the BIM metadata, which may result in a more complete Brick metadata model [26].

5 METADATA RECONCILIATION

Differences in the Brick metadata produced by drivers must be reconciled, or *merged*, into a consistent, unified model that supports data-driven applications. Reconciliation must be performed *continuously* to account for changes in metadata sources arising from evolving representations and buildings. We present a reconciliation algorithm that extends existing record linkage techniques to graph-based semantic metadata and enforces the logical and semantic validity of the resulting unified model.

5.1 Problem Definition

Let S be a set of metadata sources which change over time; the content of a metadata source S_i at time t is given by the metadata model m_i^t. Recall that a driver produces a set of entities (Equation 1) and that $T(e_j)$ denotes the Brick triples defining an entity. The input to the reconciliation algorithm is the set of entities E from the *latest* metadata model for each of m sources

$$E = \bigcup_{i=1}^{m} D_i(m_i^{t_{\max}}) \tag{3}$$

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

The first phase of the reconciliation algorithm finds clusters of entities such that (a) all entities in the cluster correspond to the same logical, virtual or physical *instance* (the classic *record linkage* problem [36]), and (b) the union of the associated metadata is logically and semantically sound. Formally, each cluster of entities c_j has an associated metadata graph G_j which is the union of all the associated triples for entities in that cluster:

$$G_j = \bigcup_{e_i \in c_j} T(e_i) \tag{4}$$

The clustering is successful and the algorithm terminates if the metadata graph G_j satisfies a set of constraints determined by formal ontologies such as Brick.

For example, consider two metadata models m_1 and m_2 which propose the following sets of entities:

$$E_1 = \{e_1, e_2, e_3\} \tag{5}$$

$$E_2 = \{e_4, e_5\} \tag{6}$$

along with their associated sets of triples (e.g. $T(e_1)$). The reconciliation algorithm may produce a set of three clusters in which every entity in the same cluster refers to the same individual: $\{e_1, e_5\}, \{e_2, e_4\}, \{e_3\}$.

When the content of a metadata source changes, such as the result of a renovation or retrocommissioning, a driver produces a new metadata model with a newer timestamp t'. The set of entities E_i^t corresponding to the old version are removed from the input set E and are replaced by the new set of entities $E_i^{t'}$. The reconciliation algorithm, described below, must be re-executed.

5.2 Algorithm

The algorithm proceeds in two main phases. The high-level operation is illustrated in Figure 4. The input to the algorithm is the most recent set of entities from each metadata source.

5.2.1 First Phase: Record Linkage. The first phase of the algorithm currently performs two kinds of record linkage on the associated names or labels of each entity: *string matching* and *type alignment*. *String matching* calculates edit distance between entity labels to produce clusters of entities. The name of an entity can be derived from string-valued properties such as `rdfs:label`, or the URI of the entity if no string-valued properties are found. The goal of this step is to use the semantic information sometimes encoded in entity labels as one heuristic for linking [11]. Due to different naming conventions between metadata sources, there can often be greater similarity scores between entities from the same source than between entities of different sources. The algorithm assumes that all entities reported by a metadata source are distinct and only clusters entities from different metadata sources.

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

5.2.2 Second Phase: Graph Union. The second phase of the algorithm takes as input the clusters of entities from the first phase and builds and validates the graphs formed by merging their associated triples. For each cluster, the algorithm produces the graph G using the formula in Equation 4. The algorithm also adds statements to the Brick model to merge the different identifiers for the same entity (this uses the `owl:sameAs` property).

Unlike many other metadata sources, Brick is built over formal logic [16]. This allows continuous validation of a Brick model as metadata is added to it, which allows the algorithm to produce a logically valid model. The logical validation is implemented by a process called a *reasoner*, which also generates logical consequences

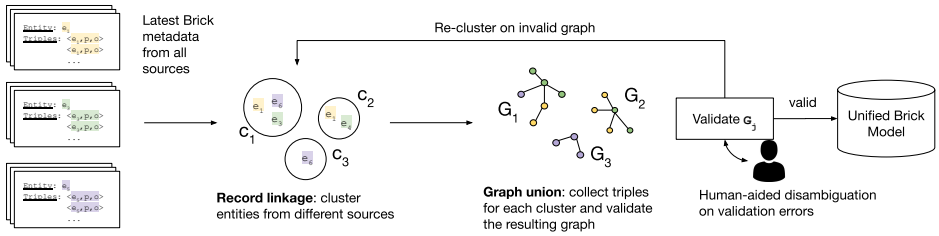

Figure 4: The phases of the reconciliation algorithm. The latest Brick metadata (far left) is stored by the integration server.

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

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

Figure 5: Example Brick metadata produced by BuildingSync and Project Haystack drivers. The `rdfs:label` property denotes the original name or identifier of the entity in the metadata source.

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

Figure 6: The inferred unified metadata model for the triples in Figure 5. The most specific type is chosen for each entity, and that associated properties are carried through.

of the statements in a Brick graph [31]. The reasoner examines the graph G_j for each cluster and produces a set of logical exceptions. These exceptions indicate that either the entities in the cluster are not equivalent, or the metadata associated with those entities is incorrect. Examples of exceptions include *incompatible types* (e.g. if a cluster contains entities with disjoint types), *incompatible relationships* (e.g. if the values of an entity's properties and relationships do not match associated constraints) and *semantic "sniff tests"* which are qualities of the Brick graph that are not logical violations but may indicate deeper issues. The primary example of the latter is an entity's types should all be subclasses or superclasses of each other.

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

5.2.3 Human-aided Disambiguation. When the algorithm logs exceptions for the entities in a given "bad" cluster, the algorithm can ask for external input on how to proceed. First, the algorithm asks if it should split the bad cluster into two or more smaller clusters; this can be performed automatically by adjusting clustering hyperparameters or manually by specifying the new clusters explicitly. If reclustering occurs, then the algorithm begins another iteration of the graph union phase above using the new clusters.

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

The reconciliation process can use human feedback to learn how to automatically cluster, classify and disambiguate entities as well as reduce the amount of manual resolution needed. Although this has not been implemented in the current proof-of-concept, the continuous integration architecture can support active learning techniques such as [11].

5.3 Example

We illustrate the behavior of the algorithm with an example of merging a Haystack and BuildingSync model for a building. The drivers for these two sources produce the Brick metadata listed in Figure 5. The algorithm begins by clustering the entities. The string-matching phase places `bldg:bsync-meter` and `bldg:ph-meter` into the same cluster because their labels are sufficiently similar. The `bldg:bsync-ahu-1` and `bldg:ph-rtu-1` entities are not grouped because the labels are too dissimilar.

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

The algorithm proceeds by unifying the triples for the entities in each cluster and validates the logical and semantic soundness of the resulting graph. In this simple example, the algorithm only needs to verify that the types of each pair of entities are compatible. This is true: `brick:Air_Handling_Unit` is a superclass of `brick:Rooftop_Unit` and `brick:Power_Meter` is a superclass of `brick:Building_Power_Meter`. Finally, the two graphs are merged into a single Brick model (Figure 6).

6 ILLUSTRATION OF USE

We explore the proposed approach by implementing a fully functional prototype and measuring aspects of its execution on a set of real and artificial sites. As structured digital representations of buildings become more standardized and widely available, we envision this framework to serve a vital role in integrating this information over time. The prototype is open-source and is available at https://github.com/gtfierro/shepherding.

6.1 Models and Sites

In order to understand the behavior and performance of the drivers, we assemble a set of publicly available models for each of the targeted metadata sources. These models do not cover the same set of buildings, but the population permits empirical measurement of the availability of Brick metadata. In total we measured 9 Haystack models, 18 BuildingSync models, 16 gbXML models and 3 Modelica/CDL models.

To illustrate the behavior of the reconciliation algorithm, we assemble a collection of metadata models for two sites (Table 1). Carytown is a public reference model for Haystack; we develop a BuildingSync model for the site using available metadata. DOE Medium Office is the reference building for a new construction, medium office in a large U.S. city that has been described in the set of U.S. Department of Energy Commercial Buildings Benchmark [12]. We used the Modelica model that had been developed for a single floor (four perimeter zones and one core zone) of this building as part of the Modelica Buildings library and developed Project Haystack and BuildingSync representations for this building.

6.2 Driver Implementation

We use the implementation of four drivers to explore the issues involved in extracting Brick metadata from existing sources, and how well the metadata synchronization protocol supports that extraction. The prototyped drivers are all implemented using a simple framework which provides an API for constructing metadata profiles and automatically synchronizes the profile with an integration server. The framework also exposes the metadata profile over a built-in HTTP server which permits a user or automated tool to debug the driver's output without the use of an external server.

This set of features reduces the developer overhead of producing a driver by abstracting away common elements of the protocol and implementation. The prototype is built in Python 3, is small (~200 LOC) and uses modules from the standard library; this suggests few technical barriers to implementing the profile and protocol in other languages.

Project Haystack Driver. The Haystack driver is built over the inference engine described in [16], which is part of the open-source `brickschema` Python package[2]. The driver only required a few lines of code to read a JSON export of a Haystack model, feed this to the inference engine, and extract the inferred Brick metadata. The division of a Haystack model into a set of entities is natural: each Haystack document becomes one or more Brick entities, with relationships between them. Due to the high degree of overlap in the modeling domain of Haystack and Brick models, most of the Haystack metadata is translated into its Brick equivalent.

[2]https://pypi.org/project/brickschema/

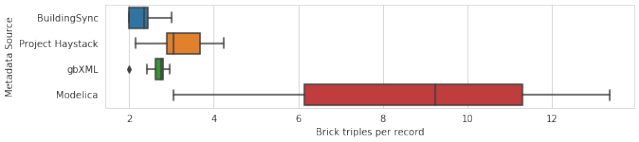

Figure 7: The distribution of the number of triples inferred per entity for each driver.

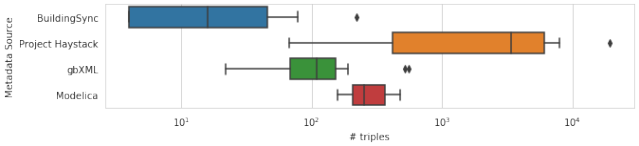

Figure 8: The distribution of the *total* number of triples inferred by each driver. Note the log-scale on the X axis

One exception is the timeseries information embedded in Haystack models (such as the current value and timestamp of a point), which has no direct representation in Brick.

BuildingSync Driver. The BuildingSync driver operates by using XPath expressions to conditionally extract parts of a BuildingSync XML document and translate the information to Brick metadata. At time of writing, the driver defines 27 direct mappings, primarily for locations and equipment types. The amount of Brick metadata obtained from a BuildingSync model is limited compared to what can be inferred from gbXML, Modelica or Project Haystack models (Figure 8). This is due to a difference in scope: BuildingSync describes properties and performance characteristics of building systems, rather than the individual components and relationships found in other metadata sources. As a result, a BuildingSync model may be a better *export* target from a unified Brick model.

gbXML Driver. We chose to implement a gbXML driver instead of an IFC driver because previous work indicates that little Brick metadata can be inferred from IFC models [23]. This is due in part to the complex and generic schema of IFC in which related pieces of information are often separated by many intermediate objects [13]. In contrast, gbXML models contain more explicit contextual information such as the `<AirLoop>` element, which groups related HVAC equipment together. This permits the inference of more metadata such as sequential and compositional relationships.

Modelica/CDL. The driver takes as input a JSON export of a Modelica/CDL model [2]. The driver treats each instance of a Modelica model in the document as a Brick entity, and assigns a Brick class to entities whose class is defined in the Modelica Buildings Library [34]. To infer relationships between these entities, the driver examines the ports for each Modelica instance; these are connected by `connect` statements to other instances of Modelica models. From these statements, the driver can infer the Brick relationships `brick:feeds`, `brick:hasPart` and `brick:hasPoint`.

To compare the behavior of each of the drivers, we executed each driver on a collection of publicly available metadata sources and measured the number of records and triples each produced. Figure 7 shows the distribution of the number of Brick triples that each driver produced *per entity* across all of the buildings. Figure 8 shows the distribution of the total number of triples obtained for each model.

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

Table 1: The results of merging multiple metadata models for two different sites, showing the diversity of the metadata available between the available metadata sources. The *% Contributed* percentages do not add up to 100% because the rest of the graph consists of inferred metadata not contained in any particular model.

Within this population of sites, Modelica models contain the most Brick metadata per entity, but do not contain as many entities as Haystack models. Haystack models contain more entities and more Brick metadata per entity than metadata sources for energy audits and BIM.

6.3 Server Implementation

We explore the reconciliation algorithm's behavior through a Python implementation of the integration server. The server exposes the API endpoints required of the metadata synchronization protocol and logs all `sync` messages received from drivers in a SQLite database. The triples in these messages, which contain the inferred Brick metadata from each driver, are inserted into a dedicated table and indexed by their metadata source and timestamp. This allows the definition of a SQL View that contains the most recent triples for each driver, which is used as input to the reconciliation algorithm. The server incorporates Allegrograph's reasoner implementation to perform the required logical validation of the Brick metadata [18]. The server also embeds an in-memory instance of HodDB [15] to support application queries against Brick metadata.

Evaluation of Reconciliation Algorithm. To develop an understanding of how the algorithm behaves, we execute the system on a collection of sites with more than one metadata source.

Table 1 contains the results of reconciling the Brick metadata from each source for each site. The *% Contributed* column contains the proportion of triples in the unified model that were contributed by each source; this includes redundant triples. The *Unique* column contains the proportion of triples in the unified model that are unique to each source.

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

For all sites, the metadata common to all drivers was very low. This is to some extent due to the completeness of the drivers at time of writing, but is also limited by the different levels of detail and different perspectives of a building that are communicated by different metadata sources. The metadata contributed from each driver was almost completely unique: even though there is some overlap in the entities described by each driver, it is rare for two drivers to produce Brick metadata at the same level of detail or level of completeness. For example, one sensor was identified as a `brick:Flow_Sensor` by the Modelica driver and a `brick:Return_Air_Flow_Sensor` by the Haystack driver.

6.4 Discussion and Future Work

The above exploration of our proof-of-concept demonstrates that integrating metadata from many different sources is not only practical, but also yields a richer and more complete representation than any individual source. The resulting unified metadata model enables the development of portable applications that can use semantic metadata to configure themselves [17].

The metadata synchronization protocol successfully decouples the tasks of inferring Brick metadata from a particular source and integrating multiple sources of Brick metadata into a unified model. This establishes a common platform for Brick metadata inference research; for example, inference methods such as [11] and [21] that operate on ad-hoc metadata representations can be adapted to the protocol. This permits direct comparison of the Brick metadata produced by different methods, and eases the integration of these methods with other drivers. The protocol also offers a clear path for future metadata standards such as ASHRAE's 223P [4] to support or integrate with Brick. We will continue to develop the existing drivers to deliver more complete and accurate Brick metadata, and implement additional drivers that operate on historical telemetry and unstructured data like BMS labels.

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

7 CONCLUSION

We present a method and system for performing the continuous integration of building metadata into a unified model *without* requiring widespread adoption of a single metadata standard throughout all stages of a building's lifecycle. The unified metadata model enables a new generation of portable, data-driven analytics built

over semantic metadata, while preserving investments in existing metadata representations. Future work will explore using the unified metadata model to support the creation or augmentation of other metadata representations from a Brick model. This work also establishes a common and extensible architecture for building metadata inference research. It facilitates future evaluations of the Brick metadata that can be produced from existing metadata representations and provides a path to integrating established and emerging inference methods.

8 ACKNOWLEDGEMENTS

This research is supported in part by Department of Energy grants DE-EE0008681, DE-AC36-08GO28308 and DE-AC02-05CH11231 and the CONIX research center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The opinions expressed belong solely to the authors, and not necessarily to the authors' employers or funding agencies.

REFERENCES

[1] 2020. *Green Building XML.* https://www.gbxml.org
[2] 2020. *Modelica to JSON parser.* https://github.com/lbl-srg/modelica-json
[3] 2020. *Project Haystack.* http://project-haystack.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] 16739-1:2018 2018. *Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries* . Standard. International Organization for Standardization, Geneva, CH.
[6] ASHRAE. 2018. *ANSI/ASHRAE/ACCA Standard 211-2018: Standard for Commercial Building Energy Audits.* Technical Report. ASHRAE, Atlanta, GA.
[7] ASHRAE. 2018. *New Guideline on Standardized Advanced Sequences Of Operation For Common HVAC Systems.* https://www.ashrae.org/news/esociety/new-guideline-on-standardized-advanced-sequences-of-operation-for-common-hvac-systems
[8] Modelica Association. 2020. *Modelica Language.* https://www.modelica.org/modelicalanguage
[9] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, et al. 2016. Brick: Towards a unified metadata schema for buildings. In *Proceedings of the ACM International Conference on Embedded Systems for Energy-Efficient Built Environments (BuildSys). ACM.*
[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.* ACM, 33–34.
[11] 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.* ACM, 3–12.
[12] M Deru, K Field, D Studer, K Benne, B Griffith, P Torcellini, B Liu, M Halverson, D Winiarski, M Rosenberg, M Yazdanian, J Huang, and D Crawley. 2011. U.S. Department of Energy Commercial Reference Building Models of the National Building Stock. (2 2011). https://doi.org/10.2172/1009264
[13] Bing Dong, Khee Lam, Y.C. Huang, and G.M. Dobbs. 2007. A comparative study of the IFC and gbXML informational infrastructures for data exchange in computational design support environments. *IBPSA 2007 - International Building Performance Simulation Association 2007* 3 (01 2007), 1530–1537.
[14] Karim Farghaly, Fonbeyin Henry Abanda, Christos Vidalakis, and Graham Wood. 2018. Taxonomy for BIM and asset management semantic interoperability. *Journal of Management in Engineering* 34, 4 (2018), 04018012.
[15] Gabe Fierro and David E Culler. 2018. Design and analysis of a query processor for brick. *ACM Transactions on Sensor Networks (TOSN)* 14, 3-4 (2018), 1–25.
[16] Gabe Fierro, Jason Koh, Yuvraj Agarwal, Rajesh K Gupta, and David E Culler. 2019. Beyond a House of Sticks: Formalizing Metadata Tags with Brick. In *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation.* 125–134.
[17] Gabe Fierro, Marco Pritoni, Moustafa AbdelBaky, Paul Raftery, Therese Peffer, Greg Thomson, and David E Culler. 2018. Mortar: an open testbed for portable building analytics. In *Proceedings of the 5th Conference on Systems for Built Environments.* ACM, 172–181.
[18] Inc Franz. 2017. *AllegroGraph: Semantic Graph Database.* https://allegrograph.com/allegrograph/
[19] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey Ullman, Vasilis Vassalos, and Jennifer Widom. 1997. The TSIMMIS approach to mediation: Data models and languages. *Journal of intelligent information systems* 8, 2 (1997), 117–132.
[20] Tobias Käfer and Andreas Harth. 2018. Rule-based programming of user agents for linked data. In *Workshop on Linked Data on the Web.*
[21] Jason Koh, Bharathan Balaji, Dhiman Sengupta, Julian McAuley, Rajesh Gupta, and Yuvraj Agarwal. 2018. Scrabble: transferrable semi-automated semantic metadata normalization using intermediate representation. In *Proceedings of the 5th Conference on Systems for Built Environments.* ACM, 11–20.
[22] Matija König, Jaka Dirnbek, and Vlado Stankovski. 2013. Architecture of an open knowledge base for sustainable buildings based on Linked Data technologies. *Automation in Construction* 35 (2013), 542 – 550. https://doi.org/10.1016/j.autcon.2013.07.002
[23] 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.
[24] Nicholas Long, Jason DeGraw, Mark Borkum, Alex Swindler, Kristin Field-Macumber, Edward Ellis, et al. 2018. *BuildingSync®.* Technical Report. National Renewable Energy Lab.(NREL), Golden, CO (United States).
[25] Fiona Moore, David Churcher, and Sarah Davidson. 2020. *BIM Interoperability Expert Group Report.* Report. Center for Digital Built Britain.
[26] Pieter Pauwels and Walter Terkaj. 2016. EXPRESS to OWL for construction industry: Towards a recommendable and usable ifcOWL ontology. *Automation in Construction* 63 (2016), 100–133.
[27] Maria Poveda-Villalón and R García-Castro. 2018. Extending the SAREF ontology for building devices and topology. In *Proceedings of the 6th Linked Data in Architecture and Construction Workshop (LDAC 2018), Vol. CEUR-WS*, Vol. 2159. 16–23.
[28] Viorica Pătrăucean, Iro Armeni, Mohammad Nahangi, Jamie Yeung, Ioannis Brilakis, and Carl Haas. 2015. State of research in automatic as-built modelling. *Advanced Engineering Informatics* 29, 2 (2015), 162 – 171. https://doi.org/10.1016/j.aei.2015.01.001 Infrastructure Computer Vision.
[29] Mads Holten Rasmussen, Maxime Lefrançois, Georg Ferdinand Schneider, and Pieter Pauwels. 2019. "BOT: the Building Topology Ontology of the W3C Linked Building Data Group. *Semantic Web* (2019).
[30] Shu Tang, Dennis R. Shelden, Charles M. Eastman, Pardis Pishdad-Bozorgi, and Xinghua Gao. 2020. BIM assisted Building Automation System information exchange using BACnet and IFC. *Automation in Construction* 110 (2020), 103049. https://doi.org/10.1016/j.autcon.2019.103049
[31] W3C OWL Working Group. 2012. OWL 2 Web Ontology Language Document Overview (Second Edition) - W3C Recommendation 11 December 2012. http://www.w3.org/TR/owl2-overview/
[32] Michael Wetter, Milica Grahovac, and Jianjun Hu. 2019. Control description language. In *Proceedings of The American Modelica Conference 2018, October 9-10, Somberg Conference Center, Cambridge MA, USA.* Linköping University Electronic Press, 17–26.
[33] Michael Wetter, Jianjun Hu, Milica Grahovac, Brent Eubanks, and Philip Haves. 2018. Openbuildingcontrol: Modeling feedback control as a step towards formal design, specification, deployment and verification of building control sequences.. In *Proceedings of Building Performance Modeling Conference and SimBuild co-organized by ASHRAE and IBPSA-USA, , Chicago IL, USA.*
[34] Michael Wetter, Wangda Zuo, Thierry S. Nouidui, and Xiufeng Pang. 2014. Modelica Buildings library. *Journal of Building Performance Simulation* 7, 4 (2014), 253–270. https://doi.org/10.1080/19401493.2013.765506 arXiv:https://doi.org/10.1080/19401493.2013.765506
[35] Gio Wiederhold and Michael Genesereth. 1997. The conceptual basis for mediation services. *IEEE Expert* 12, 5 (1997), 38–47.
[36] William E. Winkler. 1999. *The State of Record Linkage and Current Research Problems.* Technical Report. Statistical Research Division, U.S. Census Bureau.
[37] Q.Z. Yang and Y. Zhang. 2006. Semantic interoperability in building design: Methods and tools. *Computer-Aided Design* 38, 10 (2006), 1099 – 1112. https://doi.org/10.1016/j.cad.2006.06.003
[38] The City Of New York. 2020. *New York Local Law 87.* http://web.archive.org/web/20200531233953/https://www1.nyc.gov/html/gbee/html/plan/ll87.shtml