A Programming Model for Portable Fault Detection and Diagnosis

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

Gabe Fierro
Colorado School of Mines
gtfierro@mines.edu

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

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

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

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

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

ACM Reference Format:

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

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

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

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

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

\[ R_1 : T_{sa} < T_{ma} + \Delta T_{af} - \epsilon \]  

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

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

2 BACKGROUND AND PRIOR WORK
We discuss recent efforts to enable “portable” applications in smart buildings using recent metadata models.

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

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

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

2.2 Portable Application Platforms
A significant challenge for enabling the deployment of building applications at scale is the time-consuming and site-specific effort of configuring an application to run on a given building. Recent analytics platforms offer several methods for enabling the mass-customization of applications: the (semi-)automated process of configuring an application through the use of semantic metadata [7]. Building Application Stack (BAS) [19] provides a fuzzy query interface over a graph of building components and control interfaces. BuildingDepot [26] adopts a template-based approach which restricts user applications to those that can be expressed using a pre-defined sets of building entities and data sources. This trades expression of arbitrary applications for a simplified configuration experience. Mortar [15] requires tens of lines of code to express queries and application configuration logic; SkyFoundry [6] and Energy [17] use non-standard and purpose-built programming and query languages to reduce lines of code, but still require developer-driven reconfiguration between deployment sites. [8] proposed a query relaxation algorithm to improve the retrieval of building data through SPARQL results and, therefore, increase query portability across different building configurations.

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

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

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

Formally a CQ is a function \( CQ : G \rightarrow \mathbb{R} \) that returns a real-valued quantity when executed against a building’s Brick model; this quantity can be used in any subsequent computation like an FDD rule. Each CQ is defined by a set of possible resolutions; a resolution is an executable plan for producing a value from the building that can be used in an FDD rule calculation. We define three types of CQ resolutions: graph, computational, and default.
A Graph CQ resolution is a SHACL [4] shape that semantically describes the set of ways a CQ could be found in a Brick model. For example, $T_{ma}$ in our running example could be resolved as an instance of the Brick class Mixed Air Temperature Sensor associated with an instance of Brick’s AHU class.

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

A Default or user-defined CQ resolution is a human-provided default value. This accounts for rule parameters ($\epsilon$ in our running example), unknown but assumable quantities ($\Delta T_{id}$ in our running example), and other constants.

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

### 3.2 Expressing Portable Rules

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

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

Above, the function $f(\ldots)$ is the portable site-agnostic expression of the FDD rule. Figure 1 contains an example of such a rule: note that the rule definition is similar to the original mathematical formulation in §1.1, and that the rule definition contains no site-specific logic. A function $f(\ldots)$ is ported to a new building through the process of resolution. Resolving a rule produces a new function $f'(\ldots)$ in which each of the CQs has been resolved to an actual real-valued quantity that can be used for evaluating the rule ($CQ_i : G \rightarrow v_i \in \mathbb{R}$).

In case that a CQ cannot be resolved, the platform produces an error that the rule cannot be executed on $G$.

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

```python
# import provided definitions of common computational quantities
from APAR.computational_quantities import

1 def rule1(G: Graph) -> bool:
2     return Tsa(G) < Tma(G) + Tsf(G) - \epsilon # \epsilon is the error threshold

2 def rule2(G: Graph) -> bool:
3     return Taa(G) > Tsa(G) - Tsf(G) - \epsilon # \epsilon is the error threshold
```

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

### 4 IMPLEMENTATION

Our portable programming model acts as an interface between any rule-based FDD algorithm and its actual implementation in software. In this section, we explain our Graph CQ resolution mechanism, which enables the portable expression of data requirements across many different Brick models and therefore buildings. In contrast to prior mechanisms [8, 15, 17] we enable portability without placing an undue burden on the rule developer. We then explore the proposed CQ resolution algorithm in the context of APAR $R_i$ (251) and demonstrate its self-configuration and execution on two buildings and 24 AHU components.

#### 4.1 Resolution of Graph CQs

All the potential Graph CQ resolutions are implemented as SHACL [4] shapes during the specification of a rule. These incorporate various semantic descriptions of how the Graph CQ may be expressed in a Brick model. For example, the “Mixed Air Temperature” CQ can be expressed in a Brick model either as a point of an AHU or as a semantic description of how the Graph CQ may be expressed in shapes during the specification of a rule. These incorporate various semantic descriptions of how the Graph CQ may be expressed in a Brick model. This allows us to automate the extraction of Graph CQs without model-specific queries.

#### 4.2 Self-configuration

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

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

Figure 3 illustrates how a rule specification is self-configured to produce an executable implementation. The left part of Figure 3 presents an example of a specification of a self-configurable rule. In particular, rule $R_1$ is defined by three computational quantities $CQ_1$, $CQ_2$, $CQ_3$, each of which is expressed as a decision tree of potential resolutions. To elucidate the self-configuration process, we focus on $CQ_1$. If $CQ_1$ cannot be resolved as a Graph CQ, we can then look for its successful resolution as Computational CQ which requires the resolution of $CQ_4$ and $CQ_5$. In the case that...
We evaluate our programming model by determining how well it can perform fault detection in real settings. The right part of Figure 3 illustrates the implementation of rule $R_1$ for a specific building (expressed in a graph). The self-configuration of $R_1$ results in a specific implementation: $R_1 = (CQ_2, CQ_3, CQ_6, CQ_7, CQ_8)$ where $CQ_2$, $CQ_3$, $CQ_7$ are successfully resolved as Graph CQs over the Brick model of a particular building; $CQ_2$, $CQ_3$ are Default CQs.

**4.3 Evaluation**

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

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

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

**4.4 Discussion**

Developing portable applications using Mortar[15], Energon[17], depends on the ability of the rule developer to write successful rules; for space reasons, we elide the implementation of the other rules. The research leading to these results has been partially funded by the CBIM-ETN funded by the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 860555, and supported by the technical team of the Museum of London, who provided access to data from two Museums.