CHAPTER

12 SERVICE PORTABILITY AND DISCOVERY IN BUILDING OPERATING SYSTEMS USING SEMANTIC MODELING

This chapter is a cosmetic adaptation of the following conference paper: Jakob Hviid, Aslak Johansen, Gabe Fierro and Mikkel Kjærgaard. 'Service Portability and Discovery in Building Operating Systems Using Semantic Modeling'. In: *2020 IEEE International Conference on Pervasive Computing and Communications* (*PerCom*) (*PerCom 2020*). ACM. Austin, USA, Mar. 2020. **Submitted**.

The paper is submitted to the 18th Annual IEEE International Conference on Pervasive Computing and Communications, in Austin, Texas (USA), and if accepted, will be presented on 23 March 2020.

12.1 ABSTRACT

To achieve cost-efficient Building Operating Systems (BOSs), portable building services are needed. This paper presents an ontology-based service discovery mechanism for BOSs. The semantic approach enables interoperability with other semantic models used in the BOS space. The built environment is characterized by extreme heterogeneity; no buildings are entirely alike. Also, Equipment is replaced or updated, control systems and building functionality evolve, as applications, models, forecasters, and controllers improve. Therefore, services deployed in these settings must be robust to change, for them to operate at scale. Describing service interfaces using a semantic model, together with the physical context in a building, enables querying for the services needed, allowing applications to search for the data needed. This change allows applications to depend on an abstract query, instead of specific services. For evaluation, nine services running on models of the service ecosystems of three concrete buildings, are implemented, demonstrated and

evaluated.

12.2 INTRODUCTION

With society's growing concern for the environment, a significant amount of resources is being put into the green energy sector. One of the subfields in this sector seeks to improve the energy efficiency of buildings, but also to support the move to green energy sources through tight integration with the energy grid. This integration is referred to as Demand Response (DR) [19] DR allows for offsetting some of the effects of the inherent unpredictable energy production of green energy sources, thereby facilitating removal of the traditional energy sources that are undesirable for the green energy agenda. DR is achieved by allowing buildings to act as energy storage, by shifting operations, or by turning off non-essential services in response to energy grid demand. In the United States, California [18], DR is additionally used to ensure a more stable energy grid.

Traditionally buildings have been controlled by closed-loop systems, typically produced and maintained by single companies. These systems are called Building Management Systems (BMSs). Over time buildings are expanded and evolve, which means introducing the building changing BMSs and implementations, possibly even introducing multiple of these. These changes make it challenging to integrate these systems into DR operations. Fortunately, one of the growing trends in research on building operations is BOSs [28, 27, 29], that works as a layer on top of the hardware, or their controlling BMS. The presence of such a layer allows applications and services to be abstracted from the specific implementations of the hardware exposed by the BMSs. This abstraction allows the BOS to orchestrate the resources as needed, and on several scales; from a single building to citywide implementations.

Figure 12.1 captures the basic concept of the Hardware Abstraction Layer (HAL) and Service Abstraction Layer (SAL), in the context of a BOS. The components enabling service discovery, effectively implementing the SAL, will henceforth be referred to as just the SAL, as it is a new implementation of the concept introduced by Hviid and Kjærgaard [3]. The architecture is based on microservices and includes a bus for communication and small services that each solve a single well-specified problem area. Actuators and Sensors, generally regarded

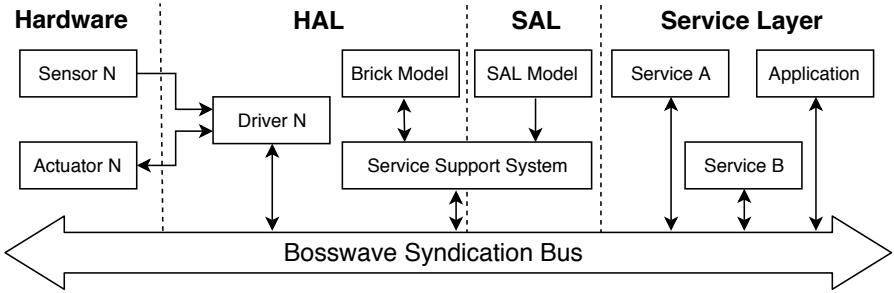

Figure 12.1: BOS Layers overview with SAL.

as the hardware components are abstracted using the HAL. In this case, the HAL is a combination of Brick, a Brick Model, the Service Support System (SSS) to host the model, and finally, the drivers that hide the specific implementation of the hardware.

In buildings it is important to understand the context, so location, direction, placement, etc., of the data being collected or processed. To provide this context, some BOSs are beginning to employ metadata models [106] or semantic models. Brick [74] is an example of a semantic approach to describing buildings, their layout, and how the hardware relates to each other and the building. However, Brick does not describe software services within the building, and how they relate to each other, nor does BOSs HAL provide this. Therefore, the SAL integrates with Brick, and benefits from its strengths. These software services provide a multitude of processed information like fault detection, prediction, occupant modeling, DR and more [2, 78, 79, 20, 80]. The missing abstraction between services, or between services and applications, makes it difficult to integrate a DR service on a larger scale between buildings, as service portability between buildings is dependent on specific implementations. Also, the need for re-implementation or reconfiguring services for each building makes the act of participating in inter-building coordination endeavors like DR infeasible from a cost-benefit perspective.

To help achieve large-scale deployment of applications and services, portability is critical. To enable portability, this paper introduces an ontology for describing services in BOSs, and thereby enable further abstraction from the building implementation. The service description enables applications and services to interface with other services,

without prior knowledge of location, or data structure. In buildings that are maintained over an extended amount of time, multiple changes, and sometimes multiple BMSs are present. This makes it difficult to integrate to only one specific service, as it could end up being different for buildings from a different decade or geographical location. For an application to work successfully in these old and changing buildings, it needs to be assumed that applications do not know what services are present, and vice versa with the services. As a consequence, an application will often be forced to retrieve information from multiple services to achieve a task. Some applications might depend on information not present in the building, which would require the application to degrade gracefully, or an engineer to install the missing service. Furthermore, the ontology, as a minimum, needs to be tied into Brick's semantic model of the building, as the context of what information is being processed by it is paramount for it to be queryable and relevant.

This paper contributes a semantic service discovery implementation, that enables service portability and discovery between buildings, as well as providing improvements to high availability and resilience initiatives in BOSs. The six services are deployed and moved between 3 different case buildings, without any changes to implementation, to evaluate the portability benefits.

12.2.1 RELATED WORK

In the field of BOSs and services, of course, several contributions have been made in a field like Building Operating System Services (BOSS) [27], SMAP [46], eXtensible Building Operating System (XBOS)[28], Bosswave [47], Buildingdepot [29], Sensor Andrew [31], Tridium Niagara [32], Volttron [30] and several others. One of the main contributions of the BOSs above, and the supporting technologies is the addition of a HAL, as described in the introduction. A HAL's role is to adapt the hardware specific interfaces to a standardized interface. This standardized interface enables applications to interact with any vendors hardware in a similar manner, thereby simplifying application implementation. This change allows the application to be written once regardless of which vendor implemented the hardware. Unfortunately, the HAL in itself does not allow for hardware or service discovery, and moving the application to a new building where the hardware structure, besides the vendor, is different without manual configuration

changes. In the intersection of BOS and DR, several contributions have been made, like improved DR decision making by introducing a service that tracks occupant activities [2] and, Nellemann et al. [20] explored automated decision and reasoning concerning DR and BOSs, to enable DR decisions that uphold specific configurable goals. The system controlling these parameters is called Controlium which is presented in the paper. In regards to entire DR systems, EF-Pi [101] is a BOS explicitly designed to serve a single DR-related application. This impacts the value proposition as added functionality is a major selling point for consumers and organizations to install BOSs in the first place and get started towards the path of smart buildings. Other examples can also be found, such as Peffer et al. [22] doing HVAC related DR, and Weng et al. [99] that seeks to include plug-loads into DR.

Service discovery is a broad term, that is used for different purposes at several layers of system architecture. At the lower level, Zeroconf [107], UPnP [108], and SLP [109] exists. Zeroconf's intention is for automatic address configuration and hostname resolution without a DNS server. UPnP is intended as a full system configuration and service discovery protocol suite for computers and devices at a network level. One example of UPnP is an automatic configuration of port forwarding in home routers. SLP is a lightweight service announcement and request protocol, used for locating printers on a local network. SLP supports a limited query language that allows querying for services properties. The three protocols all have in common that they do not support complex service querying for context and that their intended use is network level discovery and not context discovery. Expanding the exploration to higher abstractions of service discovery introduces several other technologies. Among these are Berkeley's Ninja Service Discovery [110] and Apache River [111]. With the above technologies, applications require prior knowledge of a service interface, and the context of the service provided. When exploring the above service discovery techniques, we discover they are not centered around the notion of zero-knowledge between the application and service, but about detecting instances of a service solving distinct tasks such as High Availability (HA), failover functionality, or finding the location of devices with a specific service running. Also, these services have distinct interfaces that the application is still integrating to directly, and has been specifically built around.

12.2.2 RELATED WORK

Table 12.1 compares the related technologies with the SAL approach. Three types of related work is compared:

1. Two BOS specific implementations are detailed: A combination of SMAP [46] and Metafier [63], and Brick [74].
2. Two different service description ontologies: OWL-S [71], and WSMO [72].
3. Two service interface description methods: WSDL [81], and gRPC Remote Procedure Calls (gRPC) [82].

The SAL is designed to function in the BOS space; therefore, exploring existing technologies in this space is relevant. First, the combination of SMAP [46] and Metafier [63] is discussed. Metafier adds metadata to streams of information, thereby allowing the creation of a logical hierarchical structure, but does not explicitly address discovery needs of services and their context. Metafier seeks to add metadata related to hardware devices in a building, by having an expert tell Metafier where resources are located, and what kind of properties the streams have. These annotations are associated with single streams of information, not distinguishing between hardware or services. The metadata allows Metafier to build up a logical hierarchical structure, that signifies the composition of a building, and the relation that sensors and actuators have to each other in the spatial dimension. Metafier has a limited ability to express the physical context, but only by non-related strings that, for example, can be used to distinguish between the buildings or rooms to which the stream belongs. However, these descriptions do not allow for any description of how one room relates to a building, or if a room is adjacent to another. The SMAP protocol allows for querying for streams and can take into account the metadata created by Metafier. However, these queries are limited by the limited expression of Metafier itself. Also, SMAP is not created with elaborate ontology descriptions in mind and does therefore not support these types of abstractions. Because SMAP is a protocol for retrieval of time series data, it has a precise and detailed query for the temporal aspect of the data. However, this temporal aspect is in the form of a query for data, and not a description of what can be expected of the endpoint. This is due to SMAP also acting as a HAL, and that all requests for data are expected to go through SMAP. Also, the temporal query parameters

	SAL	SMAP + Metafier	Brick	OWL-S	WSMO	WSDL	gRPC
Type	Ontology	Stream Metadata	Ontology	Ontology	Ontology	Interface Desc.	Interface Desc.
Service Descriptions	Yes	No	No	Yes	Yes	Yes	Yes
Physical Context Description	Yes	Limited	Yes	No	No	No	No
Queryable	Yes	Limited	Yes	Yes	Yes	No	No
Modality Description	Yes	Limited	Limited	No	No	No	No
Unit Description	Yes	Limited	Limited	Primitives	Primitives	Primitives	Primitives
Temporal Aspect Description	Yes	Yes	No	No	No	No	No
Organizational Description	Yes	No	No	No	Yes	No	No
Multipath Options	Yes	No	No	No	No	No	No
Multipath Priority Options	Yes	No	No	No	No	No	No

Table 12.1: Related Work Comparison.

are describing the time series data retrieved, and not the context of a single property within. Brick [74] is an ontology, that also seeks to enable service and application portability and uses ontologies to describe the hardware and how it relates to locations, and other hardware. The ontology approach has proven itself to be significantly more descriptive than the Metafier approach, capturing significant aspects of a building and how it relates to other components. Also, SPARQL Protocol and RDF Query Language (SPARQL) is versatile, allowing for complex questions about the structure of the building to be answered. Brick is specifically designed to describe the physical context of sensors and actuators, as well as how the hardware and rooms inside of a building relate to each other. It therefore not only describes the physical context exceptionally well but is also used as the description of this in the SAL. Brick, however, does not concern itself with services, interfaces, or high availability. Both Brick and Metafier does allow for Modality and Unit descriptions but is currently limited to the types existing in the hardware space, neglecting the more complex types found in the services space. Neither Metafier or Brick, models services, but only hardware (by design), and therefore fails to capture the output of services, their properties, and context. Moving on to the ontologies for service descriptions, OWL-S [71] and WSMO [72], they both describe similar perspectives on services. Both are more concerned with control logic, and interfaces, but does not describe the context of the dataset retrieved from modality, unit, temporal aspects, or physical context. While they do have some unit descriptions, these are limited to more primitive data types like integers, doubles, or strings. WSMO does describe organizational relationships of the service in the form of who created the service, and who owns it. Also, while WSMO does not support multipath HA descriptions, it does have several parameters like Robustness, and Scalability, though these describe more about the properties of the service than actually providing multiple paths. The case could be argued that either of these ontologies could be extended to support the contextual descriptions the SAL sets forth to solve. However, both ontologies are complex and do not describe anything about the content itself and how to read it, but more the interfaces to the services. Also, the ontologies complexity adds overhead with concepts like atomic processes and more, while the SAL sets out to present the core relations needed to solve the issue at hand. WSDL [81] and gRPC [82] describes the interface, input, and output of a resource.

They describe the parameters an RPC endpoint needs to function and the returned values. These technologies describe how to interface with the service, but they do not describe the context of the services they expose, nor do they allow for querying for them, as Table 12.1 states. None of the alternatives to the SAL presented in Table 12.1 describes the properties needed to support services in a BOS context.

This paper builds on the experiences made by Hviid and Kjærgaard [3], who introduced the term Service Abstraction Layer, or SAL. The paper proposes a SAL based on an ontology but unfortunately has several shortcomings. **1**) The implementation of the ontology relies on a simple inheritance strategy of a service endpoint, which is restricted by its limited expressiveness. Specifically, it does not deliver the nuance needed to locate endpoints with specific types of information successfully. **2**) As previous work points out, a service is not able to successfully find the specific information it needs from a transmitted object. This forces the service using the service endpoint to know the specific implementation of the providing service, or for there to exist a detailed specification for how these service endpoints interfaces, based on the service endpoint's inherited class. **3**) previous work also support failover functionality but does not allow for a mechanism to prefer one over another, apart from just choosing the first that was entered into the model. This is a significant shortcoming for a failover implementation. This paper builds on their work, addresses the above papers shortcomings, and add upon the functionality.

12.2.3 Approach

Solving the described issue first requires framing and boundaries. So, how can application portability and service discovery be achieved given the following restrictions? **1**) The classes of services available at this time, including ventilation usage prediction, energy benchmarking, presence prediction, weather prediction and finally solar battery storage prediction. **2**) No prior knowledge of service interfaces specifications are given, so services are built as general services, as opposed to a specific application needs, or an application built directly on a specific service. **3**) Neither application or service has prior knowledge of each other.

First, given the above restrictions, what relations need to be modeled? Second, what requirements does our class of services need to satisfy

to express their interfaces successfully? Third, how do these expressions relate to the physical aspect of the building the service relates to? Moreover, how do we restructure building services to be general services, and not to be built for specific applications? Currently, most researchers approach building applications as one problem that needs to be solved for only that application. This approach results in large monolithic applications that include all functionality needed to achieve a goal, but as a consequence, it also encloses general solutions to problems, leaving them unreachable for other services in the BOS.

To ensure service portability, an ontology approach was chosen, as prior implementations in related areas, using ontologies has been proven to be successful. These expectations are due to the fact that services and applications have no prior knowledge of each other.

12.3 CONCEPT

The SAL is the product of the simple idea of having service and interface discovery in BOSs, as an application will not have prior knowledge of what services are available, or what interfaces they expose. When applying service discovery to a building, identifying the context of the information exposed is necessary. The first need is to know where the service is located, and what properties are present in the information you can gather from that location. The SAL seeks to expose and describe these properties, with modality, unit, and spatiotemporal aspects. Also, due to the recent initiatives such as EU General Data Protection Regulation (GDPR), an ever-growing interest in ownership of data, the SAL also models ownership of an endpoint's exposed data. This is especially useful in mixed environment buildings encompassing multiple companies or private occupants. Like Brick, the service interfaces are described using an ontology, resulting in a model, which can be queried by other services that have dependencies on information, instead of specific services. The SAL allows for this kind decoupling from other services, because of the expressiveness of the RDF modeling method.

12.3.1 TERMINOLOGY

The SAL consists of several concepts and elements described below. The *SAL Ontology* is the description of SAL related concepts and their

relationships to each other. It is based on RDF [62] and OWL [83], with references to the Brick Ontology and Schema.org Ontology [68]. The *SAL Instances*, or *SALI*, is a collection of instances of modalities, units, and more, ensuring only one instance of a particular concept exists. The *SAL Model* is the Turtle file containing the modeling specific to the service implementation of a building.

12.3.2 SAL BENEFITS AND IMPLICATIONS

The SAL facilitates several functionalities and benefits. First, the primary purpose of the SAL is the provide a facility to support service discovery, thereby allowing applications to integrate with services without prior knowledge of their specifics. The service discovery allows the services consumers, that can be both services or applications, to be portable between buildings without changes in implementation. Second, the change in architecture allows developers to change their services over time, for example, splitting services into multiple interfaces or merging them. This can be done without breaking compatibility with applications that depend on the exposed information, as long as it exposes the same types of information over the new interfaces, with the same context. The above change allows for service developers to move from mostly monolithic service structures to microservices over time or vice versa.

Another area where the SAL contributes is in the area of describing information ownership. This description is especially useful in the case of mixed environment buildings, with multiple companies inhabiting the building, but also if the BOS is serving a larger area with several buildings, or even on a city scale. This usefulness is derived from the SAL's support of ownership description of service endpoints. The SSS mentioned earlier, allows the SAL to be dynamically updated. This system allows services to be installed, query the HAL and SAL, configure itself, and publish its functionality into the SAL. Effectively, the SSS enables plug and play services. Due to the nature of how services are described in the SAL, several paths to similar information can be described. This type of over-provisioning of information, enables services to implement its own HA functionality or load balancing.

Using a microservice-oriented architecture, and the SAL introduces several complexities. Debugging a service architecture where dependencies are loosely defined can be difficult, and generally increases the

focus of observability. This change will require developers to be aware of how they expose errors in the system and tell the technician why a service or application is not working. Failing to do so, or as a minimum give examples of what services an application needs, could make it difficult for a technician to diagnose problems. If error messaging is made correctly though, debugging should be relatively easy even with a service architecture this loosely coupled. From the perspective of the developer though, it is easier to detect what component is broken, and fix only that, compared to a monolithic application. This could increase correctness and robustness, as connections between program components are formalized.

12.3.3 SAL ONTOLOGY DESCRIPTION AND SALI

This section describes the anatomy of the SAL Ontology, based on the requirements described earlier in this paper. Figure 12.2 illustrates an overview of the parent classes in the SAL ontology. Subclasses are not present, as the diagram would become too large if it had to encompass all the subclasses. The implemented specific modalities and units are representative of the data streams collected across three buildings by the author's group. On top of this, several additional modalities and units were added, based on the needs of the services developed for the evaluation section.

Service describes a single service and has an object property called *provides*, which defines all ServiceEndpoints created by the service. The service class acts as a starting point for all of its endpoints and is a representation describing an actual service existing in the BOS. It has one data property, *name*, that helps give a meaning to a developer browsing the instance of the class. The *Service Endpoint* class is typically what applications are trying to locate when querying, as it encapsulates the Uniform Resource Identifier (URI) for locating the desired information. It has four data properties. First, *Read* and *Write* properties define whether the endpoint expects parameters to function, or if it provides information, or both. The data property *Priority* defines the service's priority compared to other services that provide the same kind of information. This is an indicator for depending services of which order to contact a dependency provider in case of multiple options. Finally, the data property *Uri* defines the actual location where a service will be able to locate the information. This URI can refer to the Bosswave

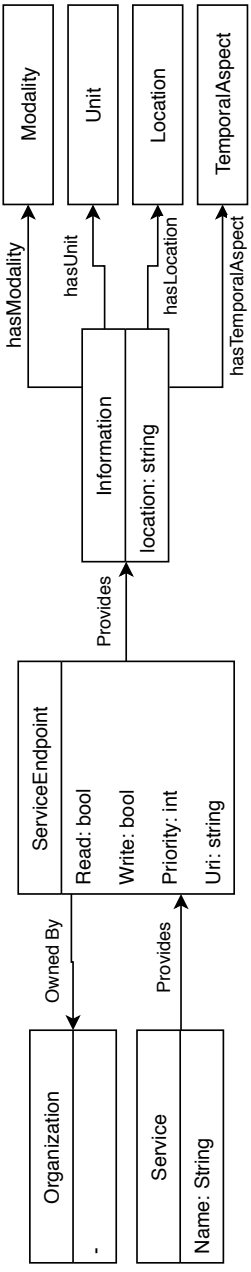

Figure 12.2: SAL Ontology Parent Classes - Relationships and Properties.

Syndication Bus, REST or similar technologies.

The Service endpoint has two object properties, *OwnedBy* and *provides*. *OwnedBy* defines the organizational owner of the information provided through the endpoint, while *provides* links to the information parent class. *Information* represents a property inside of the information obtained from the URI residing in the Service Endpoint. One endpoint will typically have a multitude of *information* object properties, that point to separate instances of information. It is important to mention that this structure assumes that the returned object is a fixed structure, and does not change. One data property is present on Information, *location*, which is a descriptor of where this single piece of information can be found within the returned object that can be retrieved from the URI described in the ServiceEndpoint. The format of the data property is not formally defined as it depends on the architecture of a given BOS. For example, if JavaScript Object Notation (JSON) is used, one possible value could be "[].MyProp" to describe that the property is found in MyProp in each object in the array received. Alternative implementations could be used here using gRPC or other alternatives that suit the specific implementation of the BOS. The information is described by its object properties, *hasModality*, *hasUnit*, *hasLocation*, and *hasTemporalAspect*. Each of these object properties describes a small but significant part of the context of that piece of information. This is an essential part of querying the SAL, as it enables the querying for the context of the information.

The *Modality*, *Unit* and *TemporalAspect* parent classes each consist of several subclasses, none of which have any object or data properties. The subclasses are shown in Table 12.2. These subclasses used in conjunction, add context to the information being described. An example could be the combination of the modality Wind, the Unit MetersPerSecond, and the temporal aspect Prediction. Each has little meaning by itself, but the combination provides an added insight. The lists of modalities and units are on what needed types where observed, but more will need to be added over time. To save the developer time, each descriptive type have instances defined beforehand in the SAL Instances file. This file is not a requirement, but a convenience. Creating these instances for the developer beforehand ensures cleaner models, as well as only one type of each existing at any given time. All instances are subclass of Modality, Unit, and TemporalAspect.

Imported from Brick, *Location* adds a spatial dimension, describing

Modality	Unit	TemporalAspect
Angle, CO_2, Presence, Flow, Illuminance, Power, Pressure, Rain, Humidity, Temperature, Wind, AbsoluteTime, RelativeTime, PowerFlexibility, Performance, Energy, Certainty, Time	Boolean, Count, CubicMeters, CubicMetersPerHour, DegreeCelsius, DegreeFahrenheit, Degrees, GigaByte, KiloByte, Hours, Hertz, Joules, JoulesPerCubicMeter, Kelvin, KiloJoulesPerSquareMeter, KiloMeters, KiloWatts, KiloWattHours, Lux, CubicMeters, CubicMetersPerHour, CubicMetersPerSecond, MilliAmperes, Minutes, MilliMeters, MilliSeconds, MetersPerSecond, MilliVolts, MilliWatts, MilliWattHours, Pascal, Percent, PartsPerMillion, RotationsPerMinute, Volts, Watts, Unitless, Time, DateTime, Date	Prediction RealTime Archival

Table 12.2: Subclasses for information annotation.

the concept of building, room, zone, and more. As Brick is already describing the physical properties of a building and uses the same types of descriptive technologies, we consider it well suited for describing the physical context.

Returning to one of the *Organization* class, this class represents a company and its ownership of the information contained at the URI residing in the service endpoint. To not re-engineer a concept already thoroughly explored, the *Organization* parent class is imported from the schema.org [68] ontology. Schema.org has already contributed a significant amount of work modeling organizations, including a multitude of subclasses, as well as the possibility of describing divisions, owners, contact information and more. It is up to the BOS developer to choose the level of detail one wishes to have in these models. This parent class is particularly interesting in buildings with shared spaces between several companies, where a service might only service a single company, and not have permission to process data gathered from other sources. The class does not have any data properties in the figure, as everything is inherited from schema.org.

Figure 12.3 visualizes a limited sample of a SAL Model for a weather prediction service. The example is limited by showing one service

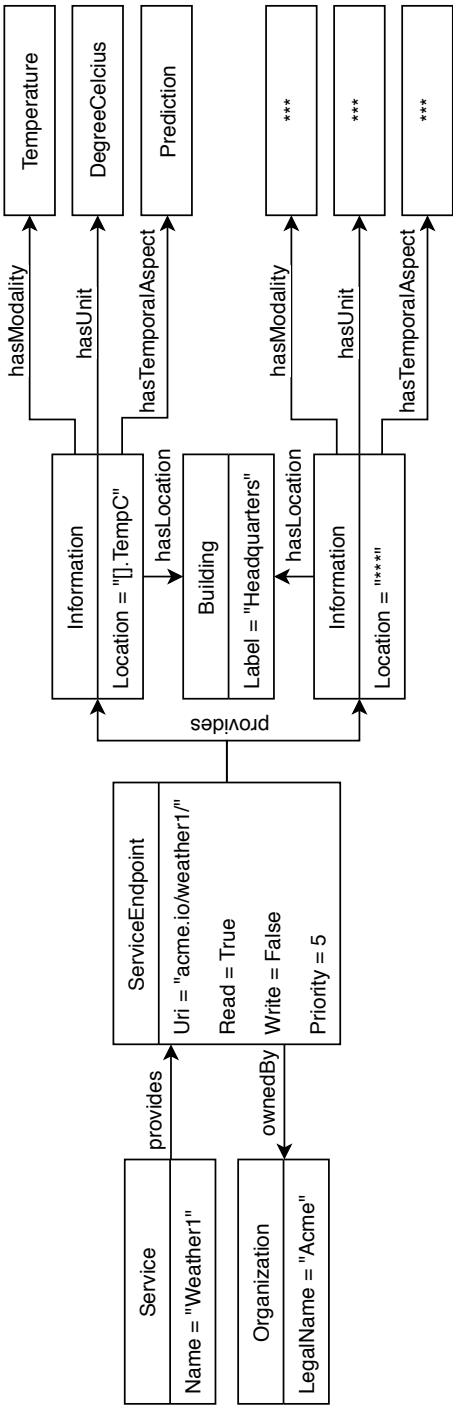

Figure 12.3: SAL Model; Weather Example.

endpoint, and one describing property. The service endpoint has a priority set, giving it a weight if more than one weather prediction service is present. The information modeled is owned by the Organization "Acme". The information found at "acme.io/weather/1", is described to be an array of objects, where each array has a property called "TempC", which is a temperature measured in degree Celsius. The value is a predictions about the weather around the "Headquarters" building. Additional information could be described, which is indicated by the last information box.

12.4 EVALUATION

To evaluate the SAL and the components, an evaluation setup is built. The BOS selected and used for the verification implementation is based on XBOS [28] combined with Bosswave [47] and Brick [74]. The ontology is built using, Protégé using Web Ontology Language (OWL) and Resource Description Framework (RDF), while the instances of the ontology and the models of the buildings are generated using Python and RDFLib. The SAL Ontology, Instances, and Model files are all saved in the Turtle [84] file format. The SSS is used for hosting the SAL ontology, SAL Instances, and SAL Model, but also Brick and the Brick Model. All queries are in the SPARQL format. The experimental setup is described in Figure 13.2. The hardware and HAL are included in the diagram, to support the understanding of how the entire setup is working. Bosswave is providing integrated communication, authentication and permission functionality, which is implemented using the Ethereum Blockchain. All services for a particular building exists in the service layer and communicates through the Bosswave Syndication Bus. An application would query the SAL, and then contacts the resolved provider through Bosswave.

The evaluation is performed on three different SAL Models inspired by actual buildings and their hardware. These represent different types of buildings, namely an office building, a retail store, and an educational building. The services are structured differently for each building, but are using the same implementation of the services, to as the portability of these services are critical.

Figure 12.5 shows the dependency between services for each of the buildings. There are six different types of services in the setup, all

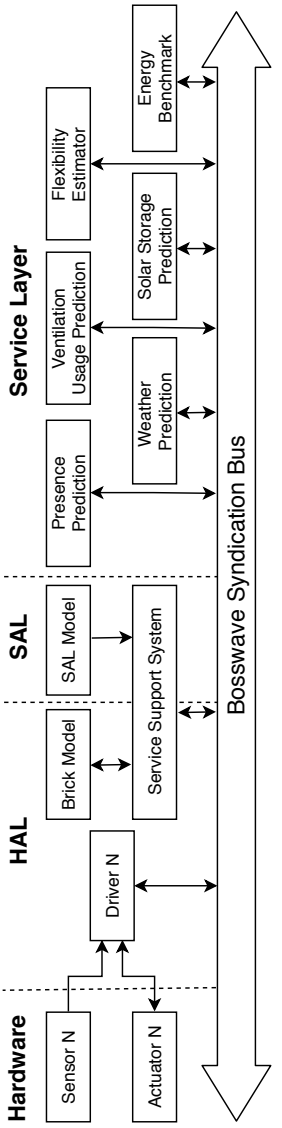

Figure 12.4: Evaluation Microservice Ecosystem.

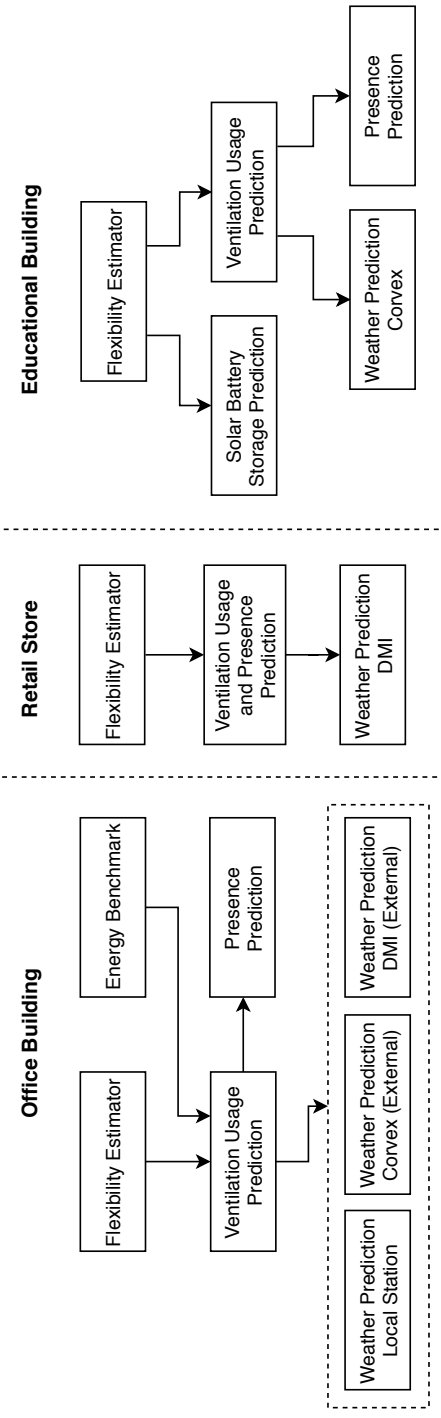

Figure 12.5: Service Dependencies For Each Case Building.

providing predictions. The flexibility estimator uses information from the solar battery storage prediction service, and the ventilation usage prediction service, to estimate the available flexibility potential. This functionality is used for DR purposes. The energy benchmark service evaluates the performance of the building, and how it changes over time. The ventilation usage prediction service uses the presence prediction and weather service prediction to estimate the need for ventilation in a room. The presence prediction service is based on the OccuRE framework [80] for predicting occupancy. Note that while some rooms are measured as a Boolean indicating presence, others have better sensors allowing for prediction of occupancy counts. This nuance is included in the SAL model. The solar battery prediction is only present in the educational building as that is the only building to have solar panels and battery storage. The service predicts the given capacity of the battery storage at any given time. For the retail store SAL Model, the ventilation usage service, and the presence prediction services are contained in the same service. This is to evaluate if service merges have an impact on the queries needed to locate the information needed.

Each BOS service in this experiment is self-configured through queries on the relevant SAL model. The office building has an extra service depending on the ventilation usage prediction service and has several weather prediction services. Two of these are external to the building instrumentation, existing in the cloud, and one that is internal in the form of a service interpreting output from a local weather station. The existence of several weather prediction services allows testing of failover functionality. The retail store SAL Model is primarily chosen to allow testing the merging of service functionality, and the impact on the queries needed to find the information. The educational building SAL Model primarily acts as the ideal setup for the flexibility estimator service. So, when the service gets moved to the other SAL Models, it is missing this service and needs to adapt to the sub-optimal environment. Of course, all of the models contribute to testing the implemented services, and thereby queries, across different environments.

The portability of services is demonstrated by using the same queries on all the models for their respective services. As expected, even though the setups are different, the queries and service implementations adapt and work on each building without changes to the code. All data transmitted between services is synthetic data, as real-time data does not benefit the evaluation of the ontology and SAL layer.

The infrastructure for high availability is tested using the office SAL Model, as it has multiple weather prediction services available. The model supplies the infrastructure for implementing failover. The ventilation usage prediction service is forced to adapt to changes in the infrastructure when a service is forced offline.

The expectations of the evaluation are as follows: **1**) *Service Portability:* Move the services between the three different SAL Models. *Expected result:* Services are working, and adapts to the changing models. No code changes needed. **2**) *Merging of Services:* Move to a SAL model where one service fulfills two service roles, instead of separate services. *Expected result:* Depend services keep running without code changes. **3**) *System Resilience:* Several services expose the same information type, and have depending services. The used service is then removed. *Expected result:* The depending service keeps functioning correctly.

12.4.1 RESULTS

Table 12.3 shows the results of the evaluation setup, with results split into the three evaluation areas. Section **A** refers to the portability evaluation, where the requirement for a ✓is that the service ran on the SAL model for the specified building, and could successfully gather the information it needed from the services it depended on. Dashes refer to a situation where the service is not present in the model, meaning there is no evaluation. As the table shows, all portability tests ran successfully, as expected. Section **B** refers to the merging and splitting of service roles, and here a ✓means to a successful merge or split, without impact on services. The table shows all merge or splits of services worked as expected. Section **C** refers to the system resilience aspect of the evaluation, where a ✓means the failover test for the Office Building worked as expected, by arbitrating similar or competing services. Dashes mean that that type of evaluation was not available for that SAL model. All testing was performed at runtime. Based on the observations during the evaluation, the SAL should scale comfortably to several thousand services. This number depends significantly on the implementation of the SSS, and query complexity.

	Flexibility Est.	Energy Bench.	Vent. Usage Pred.	Presence Pred.	Weather Pred.	Solar Storage Pred.	Merge/Split Services	System Resilience
Office Building	✓	✓	✓	✓	✓	-	✓	✓
Retail Store	✓	-	-	-	✓	-	✓	-
Educational Building	✓	-	✓	✓	✓	✓	✓	-
	A						B	C

Table 12.3: Service Portability Evaluation Results.

12.5 DISCUSSION

The results show service portability, merging and splitting of services, and system resilience, are all functioning as expected. This means the interfaces was successfully expressed using the SAL ontology, as well as the physical context, also allowing for a restructuring of building services, without impact on the depending services.

Several factors could be improved, or explored further. First, BOSs has a steep learning curve. Several technologies, terms, and abstractions that are introduced require the developer to explore technologies like blockchain (due to the Bosswave Syndication Bus in Figure 12.1), ontologies, SPARQL, RDF, turtle and many more. These BOSs and abstractions layers, could benefit from frameworks that encapsulate these in tools that are familiar for developers. A framework that abstracts some of this into a simple package could severely reduce the initial friction when moving into the field of BOSs, and require less prior knowledge. As familiarity increases, the developer can move into the more obscure parts of the systems.

In regards to the SAL Ontology, services requiring input parameters still require some prior knowledge of the service. For example, if a service changes its prediction regularity, the endpoint facilitating these changes cannot currently be described, as the SAL Ontology is created from the needs of the service examples in this case. One solution would be to let the service accept input defining temporal granularity for the output of the service. To alleviate the above restrictions, the ontology needs further expansion. Another area where the SAL ontology needs

expansion is in regards to modalities and unit types. The given types in this paper are not exhaustive, and only by working more with the SAL Ontology and services will these be explored. Fortunately, by nature, RDF ontologies are accessible for a user to expand upon, and will not be restricted by the current modalities and units.

12.6 CONCLUSION

This paper set out to create an ontology that support service discovery, and enable portability of BOS applications and services, in an ecosystem with no prior knowledge between an application, and the services it uses. This goal has been achieved. The evaluation shows the SAL gives tangible benefits to the goal of enabling portable services. Also, introducing the SAL gives the added benefits of applications being dependent on information, and not specific services, enabling a more loosely coupled and adaptive service landscape. The SAL also enables potential resilience benefits to the BOS as services provide information, and this information can be provided redundantly. This redundancy enables services to implement failover functionality if services are sensitive to downtime. As the evaluation shows, failover functionality is successfully achieved by arbitrating similar or competing services.

The SAL enables large-scale deployment of services across different types of buildings that change over time, with the same codebase. Also, it allows developers to make standard products, instead of custom implementations for each customer, thereby reducing development costs per customer. For future work, the SAL should be evaluated by running on multiple buildings over a prolonged period of time, to validate it in a running, evolving, and expanding setting with multiple service developers using the ontology. However, the SAL paired with the HAL can have a significant impact on a BOS ecosystem and has the potential to change the return of investment calculations when deciding if a service is going to be profitable to develop.

REFERENCES

[2] Jakob Hviid and Mikkel Baun Kjærgaard.
'Activity-Tracking Service for Building Operating Systems'.
In: *2018 IEEE International Conference on Pervasive Computing*

and Communications Workshops (*PerCom Workshops*). IEEE. 2018, pp. 854–859. DOI: 10.1109/PERCOMW.2018.8480362. **Published**.

[3] Jakob Hviid and Mikkel Baun Kjærgaard. 'Service Abstraction Layer for Building Operating Systems: Enabling portable applications and improving system resilience'. In: *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids* (*SmartGridComm*). IEEE. Aalborg, Denmark, Oct. 2018, pp. 1–6. DOI: 10.1109/SmartGridComm.2018.8587543. **Published**.

[4] Jakob Hviid, Aslak Johansen, Gabe Fierro and Mikkel Kjærgaard. 'Service Portability and Discovery in Building Operating Systems Using Semantic Modeling'. In: *2020 IEEE International Conference on Pervasive Computing and Communications* (*PerCom*) (*PerCom 2020*). ACM. Austin, USA, Mar. 2020. **Submitted**.

[18] Mary Ann Piette, Sila Kiliccote and Girish Ghatikar. 'Field Experience with and Potential for Multi-time Scale Grid Transactions from Responsive Commercial Buildings'. In: *ACEEE Summer Study on Energy Efficency in Buildings*. 2014.

[19] Mikkel Baun Kjærgaard, Krzysztof Arendt, Anders Clausen, Aslak Johansen, Muhyiddine Jradi, Bo Nørregaard Jørgensen, Peter Nelleman, Fisayo Caleb Sangogboye, Christian T. Veje and Morten Gill Wollsen. 'Demand response in commercial buildings with an Assessable impact on occupant comfort'. In: *SmartGridComm 2016*. 2016, pp. 447–452.

[20] Peter Nellemann, Mikkel Baun Kjærgaard, Emil Holmegaard, Krzysztof Arendt, Aslak Johansen, Fisayo Caleb Sangogboye and Bo Nørregaard Jørgensen. 'Demand Response with Model Predictive Comfort Compliance in an Office Building'. In: *SmartGridComm'17*. IEEE. 2017.

[22] Therese Peffer, David Auslander, Domenico Caramagno, David Culler, Tyler Jones, Andrew Krioukov, Michael Sankur, Jay Taneja, Jason Trager, Sila Kiliccote et al. 'Deep demand response: The case study of the CITRIS building at the university of California-Berkeley'. In: (2012).

[27] Stephen Dawson-haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev and David Culler. 'BOSS: building operating system services'. In: *NSDI '13* (2013), pp. 1–15.

[28] Gabriel Fierro and David E Culler. 'XBOS: An Extensible Building Operating System'. In: *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*. ACM. 2015, pp. 119–120.

[29] Thomas Weng, Anthony Nwokafor and Yuvraj Agarwal. 'Buildingdepot 2.0: An integrated management system for building analysis and control'. In: *BuildSys'13*. ACM. 2013.

[30] Bora Akyol, Jereme Haack, Brandon Carpenter, Selim Ciraci, Maria Vlachopoulou and Cody Tews. 'Volttron: An agent execution platform for the electric power system'. In: *Third international workshop on agent technologies for energy systems valencia, spain*. 2012.

[31] Anthony Rowe, Mario E Berges, Gaurav Bhatia, Ethan Goldman, Ragunathan Rajkumar, James H Garrett, José MF Moura and Lucio Soibelman. 'Sensor Andrew: Large-scale campus-wide sensing and actuation'. In: *IBM Journal of Research and Development* 55.1.2 (2011), pp. 6–1.

[32] Vykon by Tridium. 'Niagara Networking & Connectivity Guide'. In: *Niagara Release* 2 (), p. 245.

[46] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz and David Culler. 'sMAP: a simple measurement and actuation profile for physical information'. In: *SenSys'10* (2010). DOI: 10.1145/1869983.1870003.

[47] Michael P Anderson, John Kolb, Kaifei Chen, David E Culler, Randy Katz, Michael Andersen, John Kolb, Kaifei Chen, David E Culler and Randy Katz. 'Democratizing Authority in the Built Environment'. In: *BuildSys*. 2017.

[62] Graham Klyne and Jeremy J Carroll. 'Resource Description Framework (RDF): Concepts and Abstract Syntax'. In: *W3C Recommendation* 10.October (2004), pp. 1–20. URL: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[63] Emil Holmegaard, Aslak Johansen and Mikkel Baun Kjærgaard. 'Metafier - a Tool for Annotating and Structuring Building Metadata'. In: *SmartWorld'17*. 2017.

[68] Open Community. *Schema Org*. https://schema.org/.

[71] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne et al. 'OWL-S: Semantic markup for web services'. In: *W3C member submission* 22.4 (2004).

[72] Jos de Bruijn, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, Uwe Keller, Michael Kifer, Birgitta Konig-Ries, Jacek Kopecky, Ruben Lara, Holger Lausen, Eyal Oren, Axel Polleres, Dumitru Roman, James Scicluna and Michael Stollberg. *Web Service Modeling Ontology (WSMO)*. [Online; accessed 22. Jul. 2019]. Mar. 2014. URL: https://www.w3.org/Submission/WSMO.

[74] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Bergés, David Culler, Rajesh K. Gupta, Mikkel Baun Kjærgaard, Mani Srivastava and Kamin Whitehouse. 'Brick : Metadata schema for portable smart building applications'. In: *Applied Energy* (2018). DOI: 10.1016/J.APENERGY.2018.02.091.

[78] Elena Markoska, Muhyiddine Jradi and Bo Nørregaard Jørgensen. 'Continuous commissioning of buildings: A case study of a campus building in Denmark'. In: *iThings, GreenCom, CPSCom, and SmartData*. IEEE. 2016, pp. 584–589.

[79] Krzysztof Arendt, Ana Ionesi, Muhyiddine Jradi, Ashok Kumar Singh, Mikkel Baun Kjærgaard, Christian Veje and Bo Nørregaard Jørgensen. 'A building model framework for a genetic algorithm multi-objective model predictive control'. In: *REHVA World Congress*. 2016.

[80] Mikkel Baun Kjaergaard, Aslak Johansen, Fisayo Sangogboye and Emil Holmegaard. 'OccuRE: An Occupancy REasoning Platform for Occupancy-Driven Applications'. In: *CBSE'16*. IEEE, 2016. DOI: 10.1109/CBSE.2016.14.

[81] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana et al. *Web services description language (WSDL) 1.1*. 2001.

[82] Google Inc. and WeWork Companies Inc. *gRPC: A high performance, open-source universal RPC framework*. https://grpc.io/.

[83] W3C. *OWL*. https://www.w3.org/OWL/. 2012.

[84] W3C. *Turtle*. https://www.w3.org/TR/turtle/. 2014.

[99] Thomas Weng, Bharathan Balaji, Seemanta Dutta, Rajesh Gupta and Yuvraj Agarwal. 'Managing plug-loads for demand response within buildings'. In: *BuildSys'11*. ACM. 2011.

[101] Flexiblepower-Alliance-Network. *EF-Pi*. https://flexible-energy.eu/ef-pi/.

[106] David Huynh, David R Karger, Dennis Quan et al. 'Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF.' In: *Semantic Web Workshop*. Vol. 52. 2002.

[107] Erik Guttman. 'Autoconfiguration for ip networking: Enabling local communication'. In: *IEEE Internet computing* 5.3 (2001), pp. 81–86.

[108] Alan Presser, Lee Farrell, Devon Kemp and W Lupton. 'Upnp device architecture 1.1'. In: *UPnP Forum*. Vol. 22. 2008.

[109] Erik Guttman. 'Service location protocol: Automatic discovery of IP network services'. In: *IEEE Internet Computing* 3.4 (1999), pp. 71–80.

[110] Steven D Gribble, Matt Welsh, Rob Von Behren, Eric A Brewer, David Culler, Nikita Borisov, Steve Czerwinski, Ramakrishna Gummadi, Jason Hill, Anthony Joseph et al. 'The Ninja architecture for robust Internet-scale systems and services'. In: *Computer Networks* 35.4 (2001), pp. 473–497.

[111] Jim Waldo. 'The Jini architecture for network-centric computing'. In: *Communications of the ACM* 42.7 (1999), pp. 76–76.