Abstracting Interactions with IoT Devices Towards a Semantic Vision of Smart Spaces

Roberto Yus, Georgios Bouloukakis, Sharad Mehrotra, Nalini Venkatasubramanian
UC Irvine, {ryuspeir,gboulouk}@uci.edu, {sharad,nalini}@ics.uci.edu

ABSTRACT

This paper describes a middleware framework for IoT smart spaces, SemIoTic, that provides application developers and end-users with the semantic domain-relevant view of the smart space, hiding the complexity of having to deal with/understand lower-level information generated by sensors and actuators. SemIoTic uses a meta-model, based on the popular SOSA/SSN ontology with some extensions, to represent relationships between the low-level IoT devices' world (i.e., devices, observations) and semantic concepts (i.e., users and spaces and their observable attributes). It supports a language using which users can express their action requirements (i.e., requests for sensor data, commands for actuators, and privacy preferences) in terms of user-friendly high-level concepts. We present an ontology-based algorithmic approach to translate user-defined actions into sensor/actuators commands. Finally, our end-to-end approach includes a cross-layer solution to provide interoperability with diverse IoT devices and their data exchange protocols.

CCS CONCEPTS

• **Software and its engineering → Interoperability**; • **Hardware → Sensor applications and deployments**.

KEYWORDS

Internet-of-Things, Semantic interoperability, Open APIs

ACM Reference Format:
Roberto Yus, Georgios Bouloukakis, Sharad Mehrotra, Nalini Venkatasubramanian . 2019. Abstracting Interactions with IoT Devices Towards a Semantic Vision of Smart Spaces. In *The 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys '19), November 13–14, 2019, New York, NY, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3360322.3360859

1 INTRODUCTION

The emerging IoT ecosystem has the ability to impact our daily lives with a novel set of capabilities and services that brings with it new benefits (efficiencies, comfort) as well as a host of potential risks [4]. Smartspace technologies are being deployed today in homes, offices, and communities at large – numerous devices capture and sense the environment, detect events and activities around us, and communicate this information to other devices, users, and the cloud. Typical IoT devices on the market are often one-off and special purpose, e.g., smart TV's, thermostats, and smart assistants such as Amazon's Alexa or Google Home. Typical applications range from building security and zero-energy sustainable buildings to continuous health and wellness monitoring [2].

Key challenges arise in the operation of such smart infrastructure [5]. The first challenge is that of interoperability. Due to the heterogeneity of the devices and lack of a single standard interaction protocol, communicating with the devices is in most situations ad hoc and often proprietary. For the most part, manufacturers of IoT devices use diverse mechanisms to enable communication with their platforms (from Application Programming Interfaces –APIs–, socket communication, to messaging protocols –MQTT, CoAP–). There have been efforts to deal with these issues, for example regarding annotation of IoT devices [9, 11, 16] or dealing with the heterogeneity of exchange protocol semantics [6, 13, 19]. However, there are still open interoperability challenges specially w.r.t. the holistic end-to-end vision of smartspaces from applications to devices. The next challenge is that of facilitating reusability. Ideally, developers should create smart applications that can be reused across multiple contexts (e.g., smart homes/offices/cities) regardless of the underlying device infrastructure. Overall, this is challenging as developers today need to encode the communication with each specific device in their applications as well as the domain in which the application will run.

Additional challenges arise due to the low level nature of the data captured by IoT devices and the "semantic gap" between such device world and the higher-level concepts that users are interested in. For instance, users might want to know room occupancy regardless on how this information is obtained. Indeed, this could be done through audiovisual data from IoT cameras and microphones after a further analysis process to extract meaningful semantics or after analyzing the connectivity packages of the WiFi Access Point that covers that room. Additionally, recently there has been a significant legislative support for user privacy (e.g., the European General Data Protection Regulation –GDPR– or the California Consumer Privacy Act –CCPA–). The semantic gap in smartspaces makes particularly challenging for users to understand what data about them is being collected/inferred and thus, expressing their preferences about it. This means that the developer bears the onus of bridging the semantic gap to make IoT spaces amenable for users while addressing tradeoffs between computation/communication cost, interaction with devices, and even concomitant privacy implications [20].

We propose a middleware solution, SemIoTic, to address the aforementioned challenges, facilitating the development and provisioning of IoT smartspace applications. To deal with issues of interoperability at the semantic-layer, we present an extensible and

BuildSys '19, November 13–14, 2019, New York, NY, USA

ACM ISBN 978-1-4503-7005-9/19/11…$15.00
https://doi.org/10.1145/3360322.3360859

general metamodel, based on the popular SOSA/SSN ontology [14], which is used to define static and dynamic aspects of a smartspace including the domain (spatial aspects and users), insitu and mobile IoT devices (i.e., sensors and actuators), and the dynamic data captured. SemIoTic provides programmatic support and algorithms to specify and translate user-defined actions based on semantically meaningful concepts represented in the metamodel to the specific services and the low-level sensor data required to make inferences. To deal with issues to achieve interoperability at the data exchange layer, we define wrappers for IoT devices which consist of a common interface to enable SemIoTic to communicate with them and a device/manufacturer/model-specific code that encapsulates the low-level interaction. Also, SemIoTic defines a specification methodology for virtual sensors which enable a semantic interpretation of low-level sensor data and provide a application-oriented access to the smartspace with clear definitions of input and output datatypes. The main contributions of this paper are:

- Metamodel based on the SOSA/SSN ontology to connect IoT devices to high-level more semantically meaningful concepts in a smart space.
- Language to enable users to define their requirements for actions based on high-level concepts defined in the metamodel.
- Ontology-driven mechanism to automatically translate user actions into the appropriate IoT device actions.
- Approach to abstract low-level data exchange protocols employed by sensors.

The specification of SemIoTic is used to develop a reference implementation through which we showcase how SemIoTic can deal with a motivating use case and the benefits for developers.

2 RELATED WORK

In the following, we review works focused on the two main goals of our approach: facilitating IoT smart application development and providing interoperability between different IoT protocols.

Multiple proposals of IoT frameworks to facilitate management of devices and development of applications have been presented in both the industry (e.g., EvryThng, Nod-RED, Google Cloud IoT), as well as in academia [22]. For instance, the IDeA framework [10], based on the IoT-A reference model, provides an abstraction of the IoT devices and a tool to define applications based on them. Similarly to our approach, they envision different stakeholders defining different components needed to create IoT applications (e.g., device and domain experts and IoT application developers). IoTLink [18] provides a visual interface for developers to define applications in terms of connections between devices and software components. It has a layered architecture to handle the communication with different IoT technologies and to expose domain objects to developers. The framework at [15] also provides a visual interface to prototype applications by defining connections between heterogeneous IoT devices. It uses WebRTC data channels to enable communication between devices that support that protocol and a proxy for those that do not support it. The main difference between these approaches and the one presented in this paper is that, in general, their focus is on facilitating the understanding of what the underlying device infrastructure is and easing the process to develop application interconnecting them. In contrast, in our approach we aim at enabling the creation of applications based on high-level entities completely abstracting out completely the existence of IoT devices.

Multiple approaches enable protocol-to-protocol bridging such as the following: QEST [8] broker for CoAP and REST protocols, HTTP-CoAP proxy [7], and Ponte for REST, CoAP, and MQTT. These approaches require the implementation of one protocol to all existing ones. This is highly inefficient due to the vast development of IoT protocols. To avoid such an issue, other works propose the use of software abstractions. XWARE [19] implements mediators to translate messages of IoT protocols using an intermediate format. This is designed based on common interaction paradigms described in [13] for SOA. Then, authors of [13] extended their work in [6] to deal IoT heterogeneity using software abstractions and code generation. While this approach reduces the development effort considerably, the above works do not take into account semantic layer incompatibilities that are very common issues in the IoT.

In summary, the main contribution of SemIoTic in comparison with the above works is to provide a holistic end-to-end approach for providing inteoperability: at the application layer (by automatically translating high-level user requirements into device actions), and at the device layer (by abstracting the interaction with heterogeneous sensors regardless of their specific protocols/formats).

3 OVERVIEW

Motivating Use Case. Consider a developer that wants to implement a smart application to provide thermal comfort to the inhabitants of a space. First, she will have to decide the context of the application, as depending on the specific space the deployment of IoT devices will be different. For instance, in an office space this would involve HVAC sensors/actuators (e.g., through BACnet protocol) while in a smart home this could involve smart thermostats (e.g., Nest through MQTT) or even a fan connected to a smart switch (e.g., a WeMo switch through RESTful APIs). Ideally, the developer would like to build a single thermal comfort application that runs everywhere but that would require developing and maintaining updated code that can consider the large variety of current and future IoT devices.

Let's imagine that the developer wants to make the application *smarter* to automatically control the temperature of the space depending on how many people are inside or who is among them. Now the developer has to understand what other sensors in the space can be used to retrieve such information (e.g., beacons, WiFi Access Points –APs–, entry detectors, video cameras, etc.). Additionally, she will need to develop the (sometimes sophisticated) logic that translates raw sensor data into, for instance, occupancy data or reuse libraries created by others. Indeed, there might be different algorithms available to perform such analysis (e.g., based on extraction of faces from images or other sensor fusion techniques) and she will need to choose one (e.g., taking into account their accuracy, resilience, etc.). If the application has to run everywhere, different codes will have to be included that might need to be adapted if the configuration of a space changes.

Finally, let's imagine that the developer wants (or is forced) to follow a privacy-by-design [17] approach in the development process. For example, by taking into account the privacy preferences/policies of the inhabitants of the space when capturing sensor data. A traditional privacy challenge is that of translating people's privacy

preferences into actual enforceable settings. For instance, a person's preference could be not to be tracked/located but the application needs to know which sensor data can it collect or not.

SemIoTic would facilitate the development of such an application by enabling users/developers to focus on expressing what are their requirements in high-level terms, and agnostic of the type of smart space and device deployment, and hiding the complexity of dealing with IoT devices. In fact, the goal of SemIoTic is to make the development of such an application as easy as to just express the need to "decrease the temperature of rooms where the occupancy is at least half of their capacity and the temperature is above 75F".

High-Level Architecture. There are three main stakeholders involved in the SemIoTic ecosystem: Administrators of a smartspace who describe it into the system (e.g., what types of sensors are deployed, what information do they collect, etc.); Developers who utilize SemIoTic to develop applications that interact with the smartspace; Users of the applications who express through them actions they want to perform in the space. SemIoTic handles such *user-defined actions* and manages the IoT devices defined in the smartspace to perform them. User-defined actions are of three types: 1) Requests for dynamic or static information about the space (e.g., to obtain the current location of a person or to monitor the occupancy of a specific room every five minutes for the next two hours). 2) Commands related to such entities (e.g., to switch on the AC if the occupancy of the room is above its capacity). 3) Privacy preferences/policies regarding the handling of information (e.g., to deny the capture of any information that can lead to determining the location of a person). The architecture of SemIoTic to handle such interaction is based on three main components (see Fig. 1):

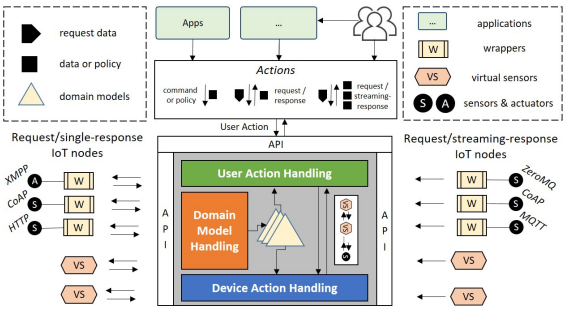

Figure 1: High-level architecture of SemIoTic.

Model Handling (see Section 4), which enables administrators to describe the smartspace in terms of types of *spaces*, *users*, and *devices*, as well as specific instances of those types.

User Action Handling (see Section 5), which takes as input user actions, based on high-level concepts defined in the model, and translates them into an appropriate and feasible plan of actions on the devices deployed in the smartspace.

Device Action Handling (see Section 6), which is responsible to access the devices assigned to execute the plan through *wrappers*, that encapsulate the interaction, and/or *virtual sensors*, that process raw sensor data to produce semantically meaningful information.

4 DATA MODEL HANDLING

SemIoTic bases its processing on a model (semic), that describes the smartspace, and a language (SemIoTic Action Language –sal–), to enable users to define complex actions related to the entities in the model. See [1] for the complete specification of semic and sal. The model, managed by the administrator of the SemIoTic-enabled space, is built on concepts of the semic metaontology. semic is an extension on the SSN/SOSA ontology [14] to support the automatic translation of user actions defined around higher level concepts into device actions at a lower level. Fig. 2 shows a snippet of semic along with the required elements of SSN/SOSA.

Entities of an IoT Smart Space. semic supports the definition of the higher-level concepts of the IoT space through the semic:`Entity` class (a subclass of the sosa:`FeautureOfInterest`) and its specialization the semic:`Person` and semic:`Space` concepts, which are intrinsic to smart spaces. We advocate the creation of subclasses of such concepts to represent different types of entities in a smart space. Each of those entities can be related to properties of interest, semic:`Properties`, which can be either semic:`StaticProperty` (whose value is a literal –e.g., string or integer – and does not depend on any IoT device), semic:`ObservableProperty` (whose value can be captured by sensors), or semic:`ActuatableProperty` (which can be actuated through an actuator). The main difference between these properties and their corresponding SSN/SOSA counterparts is that instead of just assigning a specific device to each, we include an attribute (semic:`observationType`/semic:`actionType`) to describe what is the expected value type of such property. For example, one could define the property "TemperatureProperty" of a room and then describe that the expected observation type of such property is "Temperature". This will enable SemIoTic to automatically infer which sensors could capture that value (e.g., any thermometer inside of the room including those integrated into smartphones that happen to be there). To support that functionality, semic includes a predefined type of static property (semic:`Extent`) related to spaces which is used to describe their geographic extent in an X-Y-Z coordinate system.

IoT Devices, Observations and Actuations. semic follows the SSN/SOSA definition of IoT devices extended to introduce two subclasses of the sosa:`Sensor` concept which we use to represent *physical sensors*, that sense the environment, and *virtual sensors*, that are software components that use data from other sensors to generate their observations about higher-level phenomena. Similarly, we specialize the concept of observation to further divide them into *raw observations*, captured by physical sensors, and *semantic observations*, captured by virtual sensors. As before, we expect the administrator to define appropriate subclasses of the sensor or actuator concept and then associate instances to them. For instance, one could define the subclass of sensor "Thermomether" which observes the subclass of raw observation "Temperature". We use the concept of observation, connected to sensors, along with the aforementioned attribute semic:`observationType`, connected to properties of entities, to bridge the gap between high-level and low-level concepts (this is the similar for actuators).

semic also introduces two attributes to devices which are used to represent their location, in the same coordinate system used to describe the space as well as the coverage of the device. This way, we can represent, for example, that a thermometer sensor is inside of a room and it covers (i.e., can observe) a radius around its location. Finally, semic supports the definition of *Quality-of-Service* features for IoT devices. We reuse the QoS ontology presented in [12] to

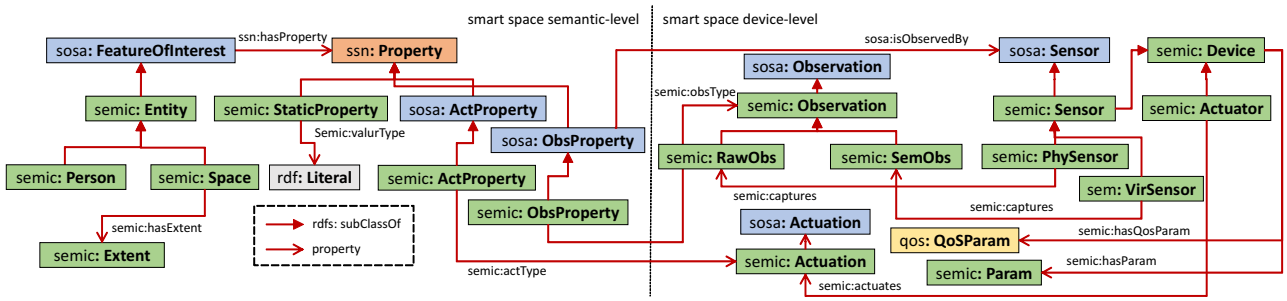

Figure 2: Overview of the SEMIC metaontology to support the description of a smartspace.

represent metrics related to devices such as their response time, latency, error rate, reliability, cost (e.g., dollars per observation).

Action Language. A SAL user action (denoted in the rest of the paper as UA) can be either a request for data (UR), a command (UC), or a policy (UP). The general format of such actions includes the following definitions: 1) Entities of interest, E, which is a set of one or more entities $\varepsilon_i \in E$ such that each ε_i is either an entity class (i.e., $\langle \varepsilon_i$, `rdfs:subClassOf`, `SEMIC:Entity`$\rangle$) or an entity instance (i.e., $\langle \varepsilon_i$, `rdf:type`, `SEMIC:Entity`$\rangle$). For example, the action can be related to either a general concept such as "Meeting Rooms" or a specific instance such as "Room 111". 2) Properties of interest, P, which is a set of properties $\rho_i \in P$ for which values have to be obtained or actions have to be performed (i.e., $\langle \rho_i$, `rdf:type`, `SEMIC:Property`$\rangle$). For example, "occupancy", "capacity", and "control temperature" in the case of a room. 3) Conditions, C, which is an expression that has to be satisfied as a condition to perform the actions on the entities. We assume that the condition expression contains one or more properties (e.g., the occupancy of the room has to be greater than the capacity). 4) Parameters, which is a set of parameters (involving both QoS and/or parameters related to the observation/action to obtain/perform). For example, this could be the definition of the measure unit for the temperature values to obtain (e.g., Fahrenheit or Celsius). Additionally, privacy policies contain two more attributes: 5) Interaction to control (i.e., capture, store, share). 6) Specific preferred action (i.e., accept or deny).
Given the above SAL definition, we can express actions such as: "retrieve the current location of John and Mary" (⟨`John, Mary, LocationProp, ∅`⟩), "decrease the temperature of those rooms with occupancy above 50% of their capacity" (⟨`Room, ControlTempProp, OccupancyProp>0.5xCapacityProp`⟩), or "do not capture the location of Mary when she is in a private space" (⟨`Mary, LocationProp, LocationProp=PrivateSpace, capture, deny`⟩).

SAL supports also the definition of device actions which are used internally by SEMIOTIC as a result of the translation of user actions. A device action DA, which can be either a sensor request SR or an actuator command AC, contains the following definitions: 1) Device to perform the action, D, (this could be a class of devices or a specific instance); 2) Entity of interest that the device should observe/actuate upon, ε, (this has to be an instance of an entity), and 3) Type of observation/action to request/command the device, a; 4) Parameters, as in the UA. As an example, we could define a request to capture temperature data from a specific thermometer as ⟨`thermometer111, room111, TemperatureObs`⟩.

5 USER ACTION HANDLING

We explain the steps involved in the handling of high-level user actions illustrated using the BPMN inspired data structure in Fig. 3.

Flattening. Complex user actions, UAs, e.g., containing conditions, require the processing of other internal actions to resolve them. For instance, to handle the action to control the temperature of rooms where the occupancy is above 50% of their capacity, we need to execute user requests to obtain the capacity as well as the occupancy of the different rooms. We refer to this process as *flattening* (borrowing the terminology used in databases to refer to the process to convert a nested query into a non-nested one).

The flattening process takes a user action, $UA =< E, P, C >$, and generates a tree structure, $\mathcal{T}_{UA}$, that contains the high-level plan required to process it in terms of other UAs. Fig. 3(a) shows the resulting data structure after the flattening of a UA. $\mathcal{T}_{UA}$ generated in this step fulfills the following: 1) The first level of the tree flattens UA by extracting the entities of interest from E (e.g., all the instances of the class "Meeting Room" in our running example). Thus, this level contains a set of UAs such that $UA_i =< \varepsilon_i, P, C >$ where $\varepsilon_i \in E$ and $\langle \varepsilon_i$, `rdf:type`, `SEMIC:Entity`$\rangle$. 2) For each UA_i, the next level flattens the set of internal UAs that need to be processed to compute C (e.g., the requests to get the occupancy and the capacity of each meeting room in our example). This is a set of URs such that $UR_{ij} =< \varepsilon_i, \rho_j >$ where ρ_j refers to the j-th property needed to compute a $c \in C$. Notice that we consider that conditions require data obtained through requests and not commands. 3) The last level of the tree flattens the UAs that need to be processed to perform the user action on each property in P (e.g., the actuable property to control the temperature in our example). Thus, it contains the UAs leaf nodes such that $UA_j =< \varepsilon_i, \rho_j >$ where $\rho_j \in P$.

The flattening algorithm takes as input the user action, UA, and the domain model M, and outputs the tree explained above. First, the algorithm extracts the list of entities, properties, and conditions associated to UA. Notice that if E contains a set of entities such that $\langle \varepsilon_i$, `rdf:type`, `SEMIC:Entity`$\rangle$ $\forall \varepsilon_i \in E$ then the method $ExtractEnt(UA, \mathsf{M})$ returns that same set. If $\langle \varepsilon_i$, `rdfs:subClassOf`, `SEMIC:Entity`$\rangle$ $\forall \varepsilon_i \in E$ then the method uses a *Description Logic reasoner* [21] to obtain any instance $m_j \in \mathsf{M}$ such that $\langle m_j$, `rdf:type`, $\varepsilon_i\rangle$. This means that the hierarchical nature of the representation is taken into account and if m_j is an instance of a ε_k such that $\langle \varepsilon_k$, `rdfs:subClassOf`, $\varepsilon_i\rangle$ then m_j will be returned. For instance, if ε_i=⟨`Room, rdfs:subClassOf, SEMIC:Entity`⟩, ε_k=⟨`Meeting Room, rdfs:subClassOf, Room`⟩, and m_j=⟨`111, rdf:type, MeetingRoom`⟩, then m_j will be returned by the $ExtractEntities(UA, \mathsf{M})$ method for a UA where ε_i ="Room"$\in E$.

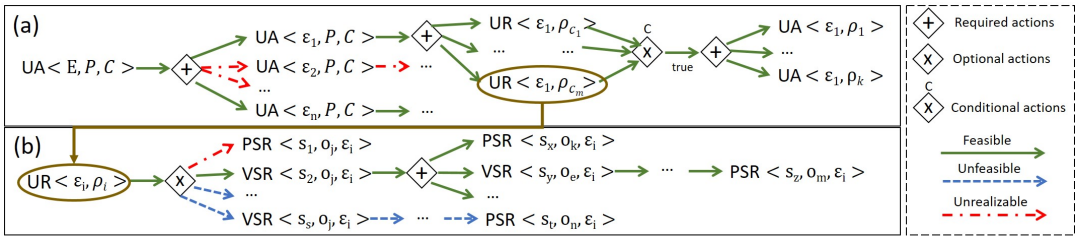

Figure 3: Structures generated to handle a User Action: (a) Flattened tree for UA and (b) Execution plans generated for a UR.

Execution Plans Generation. After flattening, $\mathcal{T}_{UA}$ contains the set of internal UAs that have to be processed to handle the user action (see for instance, the different UA_i in each level of the flattened tree of Fig. 3(a)). Each $UA_i \in \mathcal{T}_{UA}$ will require a set of device actions, DA, to be executed. Notice that more than one type of device could be able to perform such action so all the possible options have to be included as different plans. For instance, a particular UA_i representing a user request to obtain the occupancy of "Room 111" will result into a DA set to different occupancy counter sensors (either virtual or physical).

The execution plans generation step expands $\mathcal{T}_{UA}$ by extending each UA_i with a set of execution plans as $\mathcal{T}_{UA_i}$. Fig. 3(b) shows the structure of the execution plans for a particular $UR =< \varepsilon_i, \rho_j >$ that is used to expand the highlighted node in the Fig. 3(a). The constraints of $\mathcal{T}_{UA_i}$ are as follows: 1) Each level of the tree contains a $SR =< s_k, o_j, \varepsilon_i >$ which can be either for a virtual sensor, notated by VSR and such that $\langle s_k$, rdfs:subClassOf, SEMIC:VirtualSensor$\rangle$, or for a physical sensor, PSR and such that $\langle s_k$, rdfs:subClassOf, SEMIC:PhysicalSensor$\rangle$. 2) For each VSR there will be an additional level with requests for those (physical or virtual) sensor requests that need to be executed in order to obtain the input required by s_k. 3) The leaf nodes of the tree contain only PSR.

Thus, this step iterates for each $UA_i =< \varepsilon_i, \rho_j >$ node in $\mathcal{T}_{UA}$ and performs an iterative process to extract the different execution plans possible. First, the algorithm determines which device classes can execute the required action (i.e., which sensors can capture observations of the type associated with the property of interest or which actuators can perform actions of the type associated with the property of interest). To this end, it queries M to retrieve any device d such that:

$$d = \begin{cases} s & \text{if } \langle \rho_j, \text{rdfs:subClassOf, SEMIC:ObsProperty}\rangle, \langle s, \text{rdfs:subClassOf, SEMIC:Sensor}\rangle, \langle s, \text{SEMIC:captures}, o_k\rangle, \langle \rho_j, \text{SEMIC:obsType}, o_k\rangle \\ a & \text{if } \langle \rho_j, \text{rdfs:subClassOf, SEMIC:ActProperty}\rangle, \langle a, \text{rdfs:subClassOf, SEMIC:Actuator}\rangle, \langle s, \text{SEMIC:actuates}, o_k\rangle, \langle \rho_j, \text{SEMIC:actionType}, a_k\rangle \end{cases}$$

Notice that more than one device d can be obtained for the same property. For instance, different virtual sensors could retrieve the occupancy values using different inputs (e.g., WiFi connectivity data or video camera feeds). For each d retrieved, the algorithm creates a device action, DA_d where $DA_d = SR_d$ if $d = s$ or $DA_d = AC_d$ if $d = a$, and appends it as a node under the corresponding UA_i node. If $\langle d$, rdfs:subClassOf, SEMIC:VirtualSensor$\rangle$ then such virtual sensor might need additional input sensor data. For each of the input observation types defined for the virtual sensor, the algorithm similarly retrieves devices that can capture such data and appends them to that node. This process is performed iteratively until all the leaf nodes of the tree are PSR or AC.

Plan Realizability and Feasibility Checking. A plan could be *unrealizable* given the deployment of devices in the scenario or the policies defined by users. For example, consider a plan that points out that occupancy of a room can be obtained by a virtual sensor that analyzes images from cameras and by another virtual sensor that uses a movement detector sensor in the room. Those plans will be unrealizable for a specific room that does not contain any movement detector or that is not in the view frustum of any video camera. Similarly, the plan will be unrealizable if there is policy that dictates that the camera cannot be used at that moment (e.g., because a user is in the space who does not want to be tracked). In addition, some plans can be realizable but *unfeasible*. For instance, for rooms that have both cameras and motion detectors, capturing video data and analyzing it to detect occupancy might be unfeasible depending on the cost constraints of the user.

The plan realizability checking step prunes down branches of the extended $\mathcal{T}_{UA}$ (i.e., $\mathcal{T}_{UA}$ containing all the possible execution plans) that are unrealizable and classify the remaining regarding their feasibility. In the example tree of Fig. 3, we have marked some of the plans according to their realizability and feasibility as an output of this step. Notice that the result could be an empty tree if the whole UA is unrealizable because of a lack of devices that can capture raw observations or perform the required commands. In this step, SemIoTic performs a reverse level order traversal of the tree starting with the leaf nodes, which by definition contain a $DA =< D, \varepsilon_i, o_j >$ which is either a PSR or a AC. Given such node, $\mathcal{N}_{DA}$, the method `getAptDevices(`DA_i`,`M`)` obtains the set of those specific instances of physical sensors/actuators deployed in the space that can perform such action, $\mathcal{D}$, by using the function `checkCoverage(`d`,`ε_i`)` for all d such that $\langle d$, SEMIC:captures, $o_j\rangle$ (i.e., d is a device that can capture observations or actuate actions of the type related to the property of interest). The `checkCoverage` function returns true if a specific device d can cover the entity ε_i by using the SEMIC:Extent associate with ε_i and the SEMIC:location and SEMIC:coverage property associated with d. This way, if the observation type to capture is "image" and the entity of interest is "room 111", for a camera "cam111" that has the room in its view frustum `checkCoverage`("cam111","room 111") will return true.

For those devices that can cover the space, the `getAptDevices()` method performs an additional call to the `checkAccess(`d`,`$\mathcal{P}$`)` method where $\mathcal{P}$ is the set of all UPs defined by users of SemIoTic. The goal is to check whether there exists a policy that restricts access to d. Given a user defined policy $UP_n \in \mathcal{P}$ the same process described so far is applied to generate possible execution plans. Thus, SemIoTic generates a $\mathcal{T}_{UP_n}$ that contains all the different

devices involved in the processing of the policy. If there exists a node $\mathcal{N}_m \in \mathcal{T}_{UP_n}$ such that $\mathcal{N}_m = DA =<\mathrm{d}, \varepsilon_j, o_k >$ and the preferred action defined for the UP_n is to deny the access, then `checkAccess()` returns false.

If $\mathcal{D} = \emptyset$ then $\mathcal{N}_{DA}$ is removed from $\mathcal{T}_{UA}$. In such a case, SemIoTic checks $\mathcal{N}_{DA}$ parent nodes and those are also removed if they require the processing of DA_i (e.g., if the parent is a VSR that takes as input the observations of DA_i). This is done recursively and nodes are removed until a parent of an unrealizable node does not require such node (e.g., because it can obtain its input data from another child). If $\mathcal{D} \neq \emptyset$ then the node $\mathcal{N}_{DA}$ that specified an action on a general device type D has to be replaced by specific actions on the devices in $\mathcal{D}$. For each $d_k \in \mathcal{D}$ SemIoTic creates a $DA_k =< d_k, \varepsilon_i, o_j >$ and this action gets added as child to $\mathcal{N}_{DA}$. At the same time, SemIoTic computes a *feasibility score* for DA_k which can be used to compare whether a plan is more feasible than others. This score gets added to the metadata of $\mathcal{N}_{DA_k}$.

To compute the feasibility score of a DA_k (represented as a cost $C(DA_k)$), SemIoTic uses the different cost metrics defined in M for that specific device (see Section 4). These metrics (e.g., processing time, quality of the answer, economical cost, privacy grade, etc.) get aggregated by using a cost model based on weights $C(DA_k) = \sum_{j=1} w_k c_{jk}$ where w_k is the weight assigned to the $k-th$ cost, c_{jk}, associated with the device d_k in DA_k. In cases where the user explicitly requires some preferences regarding the cost or quality of the answer, this information can be leveraged to assign weights to each objective/cost. By default we consider that the value for the weights are assigned uniformly.

Plan Selection and Execution. Given a UA, several execution plans can be feasible and, if the goal is to maximize the chances of carrying out the action (as devices might fail or produce noisy results), SemIoTic can execute them all. However, in general this might result in a waste of resources as we would be duplicating efforts to obtain the same result. Thus, SemIoTic chooses the most feasible plan according to the score computed in the previous step. This way, it computes the feasibility of each plan, by recursively aggregating the cost of its nodes, and removes all the branches of $\mathcal{T}_{UA}$ which do not have minimal cost. Notice that, the flexible and modular design of SemIoTic makes it possible to use other more sophisticated optimization functions to assign costs and select plans.

Once a single plan has been selected, the next step is to execute it. The execution engine executes first each $UA_i \in \mathcal{T}_{UA}$ that is needed to compute the condition component of UA (if any). Then, after checking whether the condition is satisfied, it executes each $UA_j \in \mathcal{T}_{UA}$ related to the properties of interests in UA. Given a $UA_k \in \mathcal{T}_{UA}$ the execution engine performs a reverse level order traversal of the subtree $\mathcal{T}_{UA_k}$. Each node $\mathcal{N}_m \in \mathcal{T}_{UA_k}$ is handled as follows depending on its type. If $m = AC$, the appropriate wrapper is notified for actuating a device. Then, if $m = VSR$, the engine setups the required data inputs based on its children nodes. This is performed by creating consumers that subscribe to receive data from a virtual/physical sensors (more details in Section 6). Note that if $m = PSR$, the communication with its corresponding wrapper is handled by the VSR sensor parent node. Finally, if m is the root VSR/PSR node, the engine calls the corresponding virtual sensor or wrapper, respectively. In the case of a VSR root that call will trigger the chain of calls to predecessor nodes.

6 DEVICE ACTION HANDLING

A device action $DA =\langle d_i, a_j, \varepsilon_k \rangle$, can be a PSR or AC, or VSR handled by *wrappers* and *virtual sensors*, respectively.

Device Wrapper Design. Accessing physical sensors or actuators is challenging as they introduce multiple levels of heterogeneity. Any IoT standard protocol can be utilized by a device to push/pull data. These protocols differ significantly in terms of interaction paradigms – i.e., CoAP based on *Client/Server* interactions, MQTT following *Publish/Subscribe* and WebSocket based on the *Streaming* interactions [6, 13]. Input and output data of these protocols are defined in multiple data-serialization formats (e.g., JSON, XML, protobuf, etc.). Finally, to access these data, protocols require to use a *scope parameter* that corresponds to an operation, resource, filter or stream identifier. These may differ from device to device even if they observe the same type of data (e.g., temperature).

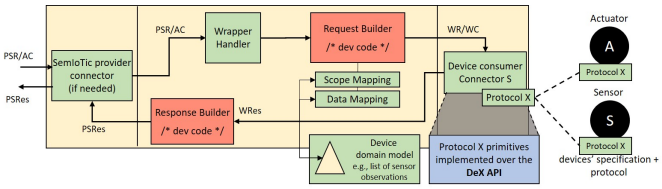

Figure 4: Device wrapper design.

The design of SemIoTic wrappers (see Fig. 4) handles such heterogeneity by providing a device-agnostic implementation for enabling cross-layer interoperability between SemIoTic and IoT devices. Our wrapper design consists of two main parts: *connector* and *mapping*. Inspired by the service-oriented architecture (SOA), we define two types of connectors: (i) *provider* – connects SemIoTic with the wrapper providing the requested data; (ii) *consumer* – connects the wrapper with the IoT device for consuming its data. Then, the mapping part bridges the provider and consumer connectors by performing: (i) *data and scope mapping* and *protocol mediation* between SemIoTic and the IoT devices.

As shown in Fig. 4, a PSR_i or AC_i can be received by the SemIoTic provider *connector* in JSON format and then forwarded to the *Wrapper Handler*. Note that the provider connector is only required if the wrapper is deployed in another machine other than SemIoTic's machine. Otherwise, the Wrapper Handler receives the PSR_i/AC_i directly from SemIoTic. The *Wrapper Handler* component provides callbacks for handling commands, single-response requests, streaming-response requests, and requests for terminating a stream. In particular, Listing 1 shows the implementation of the single-response request (lines 5-10) which is generic enough to support any request to any sensor. This incoming PSR_i is then given as input to the *Request Builder* component (line 6) that performs data and scope mapping for generating the wrapper request (WR_i) which is compatible with the corresponding IoT sensor.

To enable the developer request data from any IoT device employing any IoT protocol, she has to only specify the device's protocol name when instantiating the *Wrapper Handler* (line 3). Then, we leverage the *Data eXchange* (DeX) API [6], which implements `post` and `get` DeX primitives for sending/receiving messages using existing IoT protocols such as CoAP, MQTT, XMPP, etc. The primitives

of this API require as input the parameters $< \pi, \psi, m^{post} >$, where π is the destination of the physical device, ψ is the scope parameter and m^{post} is the request data or the command to be sent to the device. This information has to be provided by the developer in the *Request Builder* module as part of the WR_i, which we will explain in the following. Then, we instantiate the *device consumer connector* (line 7) that implements single-response requests using the DeX primitive `postExGet` with $< \pi, \psi, m^{post} >$ as input parameters. More details regarding the definition of the DeX primitives and the interaction types they support can be found in [6]. We introduce two additional parameters to the DeX API, $< \lambda, \delta >$, to manage streaming-requests. Let λ be the frequency that response items (through multiple m^{get} parameters) must be received and δ be the duration the request is active. If a response is expected, this is received through the m^{get} parameter. The mediation of the device response ($WRes_i$) to the format required by SemIoTic ($PSRes_i$) is encoded by the developer in the *Response Builder* module (line 9).

```
1  class WHandler extends Handler {
2    public WHandler(String prot) {
3      this.protocol = prot; /* device protocol name - e.g., CoAP */}
4    @Override /* code for handling requests */
5    public void handleRequest(PSRequest psr) {
6      RequestBuilder rb = new RequestBuilder(psr);
7      DevConsConnector dc = new DevConsConnector(this.getProtocol());
8      String getMsg =
           dc.getDexPrim().postExGet(rb.getDest(),rb.getScope(),rb.getPMsg());
9      ResponseBuilder resb = new ResponseBuilder(psr,getMsg);
10     psr.respond(resb.getWRes()); } }
```

Listing 1: The SemIoTic Wrapper Handler.

The *Request Builder* module has to be implemented by the developer to define the mapping of the device action parameters to the expected parameters of the DeX API (i.e., $< \pi, \psi, m^{post} >$). These can be defined by considering the SemIoTic device domain model and the technical specification of the physical device: π (destination) corresponds to the URI of the real sensor (defined when describing the sensor in the model); ψ (scope) corresponds to the operation, resource, topic or stream identifier that the data can be received from (can be identified from the list of observations in the sensor domain model and the specification of the device); m^{post} (post message) which is constructed based on the labels associated with the parameters defined in the user request and the parameters that the sensor requires. In Section 7, we will show an example of a request builder specified for a real sensor. Finally, The *Response Builder* module has to be also implemented by the developer of the wrapper to map the data returned by the sensor ($WRes_i$) to JSON format ($PSRes_i$) for the observation type in the domain model.

To summarize, if a sensor employs the CoAP protocol to receive wrapper requests (WR_i), the developer specifies the protocol name in the wrapper handler and refines the request/response builder modules. Suppose that the same sensor uses MQTT, then the developer has to only modify the protocol name and refine the request/response builders. We implement the remaining handlers of Listing 1 to enable wrapper developers supporting any possible interaction type found in the IoT. Note that the streaming-request is implemented by taking into account the frequency and duration parameters – i.e., $< \lambda, \delta >$. Hence, the *device consumer connector* must request data with a specific frequency and for a specific duration given by the application. In case an IoT protocol does not support

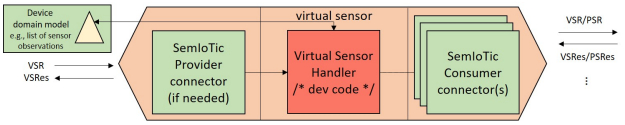

Figure 5: Virtual sensor design.

streaming interactions (e.g., HTTP), we implement these over the DeX API. In particular, we repeat a single-response request with the given frequency for the given duration. Any such streaming request can be terminated by the application using `stoppsr`.

Virtual Sensor Design. As explained in Section 4, we use the concept of virtual sensors to encapsulate the enrichment of physical phenomena captured by sensors into semantically meaningful information (e.g., extracting who is in a room based on images captured by video cameras). As in the case of device wrappers, there are multiple ways of performing such enrichment, even for the same task, using different types of input information (e.g., different algorithms exist for face recognition using different features). SemIoTic has to be agnostic to specific virtual sensors and able to interact with any of them. We provide a specification for the development of virtual sensors, similar to the one described above for device wrappers, to deal with such heterogeneity.

As depicted in Fig. 5, our design consists of three main components: the provider connector for providing requested data, a set of consumer connectors to consume data from one or more IoT devices, and the code that processes the incoming data to provide the main response. When a virtual sensor artifact is deployed, the *Virtual Sensor Handler* component awaits for incoming interactions, either virtual sensor requests (VSR) or configuration parameters (i.e., notifications for setting up data connector consumers). The purpose of the latter (see `setConsumer` method –line 2– in Listing 2) is to configure what the specific inputs (to which we refer as consumers) of the virtual sensor would be according to the selected plan for a UA. For instance, for the virtual sensor that detects people in images, the configuration notification could ask it to use as input images coming from the video camera in room 111.

When the virtual sensor receives a VSR (see `handleRequest` callback –line 5– in Listing 2) it needs to collect data from the configured consumers, perform the virtual sensing task on the collected data, and return back as response the requested (single or a stream) observation. First, the callback instantiates a list of responses that will be received later from the configured input sensors (line 6). Then, it interacts with each input/consumer to retrieve their observations. To this end, the developer of the virtual sensor can select between three request types: (i) *synchronous request* (lines 8-11): the consumer requests data and is blocked until the response is given to be stored in the overall responses list; (ii) *asynchronous request* (lines 12-16): the consumer requests data and the response is given at some point later to be stored in the overall responses list; and (iii) *streaming request* (lines 17-21): the consumer requests data with a specific frequency and for a duration of time i.e., $< \lambda, \delta >$. Multiple responses are given at arbitrary points of time but within the requested duration to be stored in the overall responses list. Finally, the developer has to implement the code snippet that performs the actual virtual sensing (i.e., processing the incoming observations and generating the higher-level information) and then provide back the response (lines 24-25).

```
class VSHandler extends Handler {
  public void setConsumer(SemConsConnector consumer, String plan) {
      consumersList.add(consumer,plan) }
  @Override /* code for handling single-response requests */
  public void handleRequest(VSRequest vsr) {
      ArrayList<Response> consRespList = new ArrayList<Response>();
      for (SemConsConnector consumer: ConsumersList) {
          /* if sync request to consumer selected */
          Response consResp = consumer.syncRequest();
          /* consumer response list to be processed in line 22 */
          consRespList.add(consResp);
          /* if async request to consumer selected */
          consumer.asyncRequest(new AsyncRequestCallback (){
          @Override
          public void onMessage (Response consResp) {
            consRespList.add(consResp); }}) ;
          /* if streaming request to consumer selected */
          consumer.streamRequest(new StreamRequestCallback (freq,dur){
          @Override
          public void onMessage (Response consResp) {
            consRespList.add(consResp); }}) ; }
      /* code to process the incoming responses in consRespList */
      /* .... */
      /* provides the final response */
      vsr.respond(new VSResponse); }
  /* .... */
```

Listing 2: The SemIoTic Virtual Sensor Handler.

It is worth noting that the developer does not have to specify any device destination IP or parameters of observations to be requested – these are already provided by SemIoTic (using our domain model) during the plan execution phase and the configuration of the consumers. Additionally, the developer does not need to deal with raw data coming directly from sensors as wrappers take care of mapping such data into the one specified in SemIoTic model. Finally, in contrast with wrappers, it is not necessary to perform data and scope mapping as well as protocol mediation – virtual sensors exchange data with other virtual sensors and SemIoTic using the same data semantics, IoT protocol and data format defined in our domain models.

7 EXPERIMENTS

We implemented a prototype of SemIoTic [3] and an application based on the motivating use case scenarios in Section 3. The prototype, domain models, wrappers, virtual sensors, and application developed for the experiments are available at [1].

Developing an application with SemIoTic. The application developed showcases an exploratory discovery of the space by enabling users to define requests to explore the spaces defined in the model and their properties. Then, the application guides the user (through a GUI) to define the SAL parameters of the user action to actuate the property for controlling the temperature of a selected space when its occupancy reaches a percentage of its capacity. It also generates another SAL request to retrieve a stream of occupancy and temperature data for the room. After posing the user actions to SemIoTic, the application generates graphs to display the obtained data (see Fig. 7).

We developed domain models, wrappers, and virtual sensors for two different scenarios: a smart office building and a smart home. For the domain models, we imported the SEMIC metaontology in the Protégé tool and extended it, with the appropriate classes and instances (e.g., rooms, occupancy, temperature, people, presence, etc.), to describe our smart Donald Bren Hall building at University of California, Irvine and a smart home (see Fig. 6(a)). We developed sensor wrappers to interact with real sensors in both scenarios including, among others, WiFi Access Points, bluetooth beacons, cameras attached to Raspberry Pis, HVAC sensors connected to the SkySpark framework (see Fig. 6(b)). Finally, we developed virtual sensors to generate occupancy, by counting the number of people in a space, and to determine where individuals are, using approaches such as exploiting connectivity data (e.g., from WiFi APs or beacons) or detecting faces on images (using OpenCV).

Handling runtime user actions with SemIoTic. We deployed two instances of SemIoTic (i.e., SemIoTic-Home and SemIoTic-Building) with the previous content. We run the application, which shows a list of spaces extracted from the domain model, and select "Room 111" and "Living Room" in the SemIoTic-Building and SemIoTic-Home instances, respectively. The application generates the following user actions that are posed into SemIoTic-Building to control the temperature of the room and to retrieve a stream of occupancy and temperature readings of the room, respectively (similar requests are generated for SemIoTic-Home):

⟨`Room111, ControlTemperature, OccupancyProp>0.5xCapacityProp`⟩
⟨`Room111, OccupancyProp/TemperatureProp, ∅`⟩

Every SemIoTic instance receives and handles the request as explained in Section 5. In both scenarios, for the action to control the temperature, SemIoTic detects the need to retrieve the room's occupancy and capacity properties first and then actuate the property to control the temperature if the condition is met. As both domain models define a virtual sensor that can retrieve occupancy observations (which is the value that the occupancy property requires) this virtual sensor is included in the tree. Next, SemIoTic discovers three virtual sensors in the model (using WiFi observations, bluetooth observations, and images, respectively) that can generate the presence data required as input to the occupancy counter sensor. These are also included in the tree as possible plans. Then, each virtual sensor appends a request to physical sensors PSR_i for consuming their output data (WiFi APs, bluetooth beacons, and cameras).

When checking realizability and feasibility of the plan, SemIoTic-Building detects that there are no cameras covering room 111 and that the virtual sensor based on beacon observations provides a more accurate answer than the one using WiFi data. Thus, the final plan selected (see Fig. 6(c)) involves a request to the virtual sensor that generates occupancy data from presence data (VSR_1), which is the same in the plan for the home, followed by a request to the virtual sensor that generates presence data using bluetooth beacon observations (VSR_2) followed by requests to the three beacons covering the room (PSR_{1-3}). SemIoTic-Home follows a similar process by detecting a camera with the living room in its view frustum (there are no WiFi APs or beacons deployed). This way, it generates a plan (see Fig. 6(c)) that involves a request to the virtual sensor producing presence based on counting faces (VSR_3), followed by a request to the virtual sensor that extracts faces from images (VSR_4), followed by a request to the camera to capture images. At execution time, each instance calls the appropriate virtual sensors and sensor wrappers according to the selected plan.

The application populates the temperature and occupancy graph (see Fig. 7) by using their definitions in the domain model. Underneath SemIoTic-Building retrieves data from HVAC thermometers and beacons whereas SemIoTic-Home uses a Raspberry Pi with a

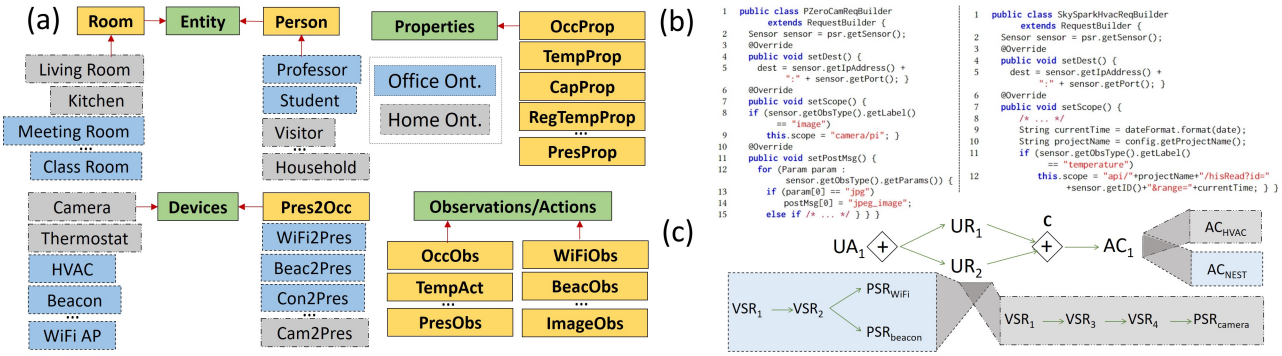

Figure 6: Snippets of (a) Two domain models based on SEMIC, (b) Code of two wrappers, and (c) Tree generated to handle a sample user action.

thermometer and a videocamera. Notice in Fig. 7 how, in the case of the building, the room starts getting full towards 9am (the starting of the meeting) and there is an increase on the temperature. When the occupancy crosses the boundary defined (in this case 75% of the capacity) the parallel user action retrieves these data and turns on the AC. Then, after some delay, the temperature starts lowering down. In the case of the smart home the situation is similar even when the underlying sensors are completely different.

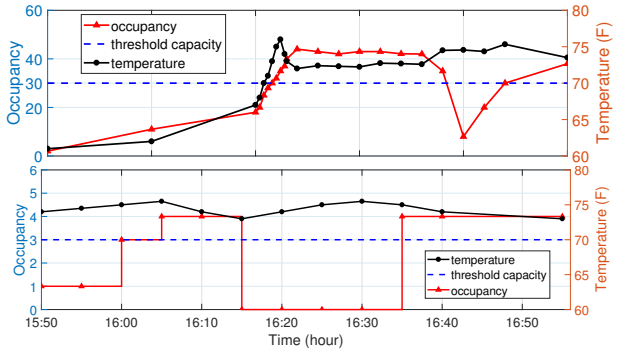

Figure 7: Graphs displayed by the application using SEMIOTIC.

In the case of the smart home, we decided to include a third user action: a policy that restricts the sharing of data. The policy (⟨`Mary, LocationProp, LocationProp=PrivateSpace, capture, deny`⟩) is processed in parallel with the other two actions and, when translated, prevents access to video camera data as it can be used to derive the location of Mary. Notice that the occupancy curve of the room drops to zero at 16:15. This is the moment when we simulated Mary arriving in the living room. At that moment, video camera data cannot be captured and this prevents the virtual sensor to obtain occupancy as the plan becomes unrealizable. Notice also, that another consequence is that the temperature starts increasing slightly as the action that controls the temperature cannot be processed due to the lack of occupancy data. At 16:35 Mary leaves the leaving room and the processing of both actions gets resumed.

As we have seen, the developer of the application did not have to deal with/understand device related information. Through SEMIOTIC, the developer relied on the information described in the domain model about entities and their properties and specified SAL user actions. Also, the same application run across different spaces without code modifications. This is possible thanks to the space-agnostic and ontology driven architecture of SEMIOTIC and the definition of both the domain models and wrappers/virtual sensors. Indeed, in this case the development effort of the application developer has been reduced due to the administrator having defined the domain models and due to the wrapper/virtual sensor developer.

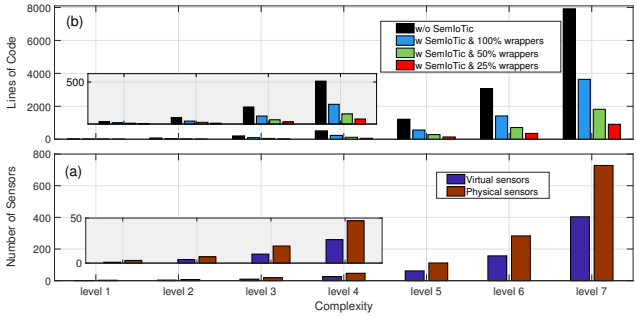

Figure 8: Development effort with (w) and without (w/o) SEMIOTIC.

Scaling up the use case complexity. For the previous experiment, we required to create 5 wrappers and 4 virtual sensors to facilitate the development of the app, which is significantly less than the effort it would require to create the same application without SEMIOTIC. In that case, there is a need to implement the logic of the application in order to support complex actions, define the metadata of spaces/devices, implement the logic for the processing of multiple inputs (data sources), implement code for accessing heterogeneous devices, configure destination URIs for handling every device action, etc. In addition, this process must be repeated to deploy the application in different spaces (due to the differences of the existing devices) and to handle different user actions. Also, this development effort will increase significantly with the increase on the level of the complexity of user actions (e.g., if a complex execution plan is required involving a large chain of virtual sensors). In the following experiment we compare the required effort for developing an application with and without using SEMIOTIC.

We developed an algorithm that generates scenarios with different levels of complexity. A scenario includes a set of sensors and virtual sensors –with varying number of inputs– which results in multiple execution plans. The algorithm starts by generating a virtual sensor and then a random number of inputs (virtual or physical sensors) by considering a maximum number of inputs $n_{in}(vs_j)$ defined as a parameter. Then, this process is performed iteratively for each new virtual sensor until the execution plan reaches a given level of complexity n_{cl}. The output is an execution plan involving $|PS|$ physical sensors and $|VS|$ virtual sensors. Using this algorithm

we generated scenarios with increasing n_{cl} (from 1 to 7) and with $n_{in}(vs_j) = 4$. For each level the algorithm created 500 different scenarios and then computed the average $|PS|$ and $|VS|$.

The algorithm also estimates the development effort to implement these plans. For that, it takes into account the cost in terms of *lines of code* (LoC) to be developed (without considering common tasks that have to be developed with and without SemIoTic such as the definition of the logic/GUI of the app, logic of the virtual sensing task, definition of metadata of the space and devices). Let LoC^{with} be the number of LoC required to develop with SemIoTic as $LoC^{with} = n_{loc}^{wrap} \times |PS|$. Where n_{loc}^{wrap} is the average LoC required to develop the data and scope mapping of a wrapper. The metric does not include virtual sensor development as we provide developers with the appropriate generic artifact so that they just need to implement the logic of the virtual sensing task. We measured the average n_{loc}^{wrap} to be 5 LoC in the simple data type wrappers generated for the previous experiments (e.g., Fig. 6(b)). Then, let $LoC^{w/o}$ be the number of LoC to develop without SemIoTic as $LoC^{w/o} = \sum_{j=1}^{|VS|} n_{loc}^{in} \times n_{in}(vs_j)$. Where n_{loc}^{in} is the average LoC required to setup an input data source (setup a consumer, configure its URI, etc). Defining n_{loc}^{in} is challenging as this may differ depending on the protocol and specific device (e.g., in [6] the authors setup an MQTT subscriber by using 8 LoC without considering the data mapping task). We assume the best case scenario when developing plans without SemIoTic by considering 5 LoC for setting up a consumer and 2 additional LoC to perform data mapping for a simple message type. Thus, we consider $n_{loc}^{in} = 7$.

Fig. 8(a) shows a plot of the average $|PS|$ and $|VS|$ with increasing level of complexity. For instance, a plan with complexity 5 would require interacting with 62 virtual sensors and 112 physical sensors. Fig 8(b) shows the number of LoC required to develop an application, with and without SemIoTic, vs. the complexity of the scenario. In the case of development with SemIoTic we consider situations with different number of wrappers required as a percentage of the number of physical sensors (as some sensors could be of the same type/brand and handled by the same wrapper). For instance, in our previous experiment for the smart building the ratio was less than 1% as we developed 4 wrappers in total for cameras (around 40), HVAC sensors (around 7K), WiFi APs (around 60), and bluetooth beacons (around 200). Fig 8(b) shows, for instance, that developing the complexity 5 plan requires 1.2K LoC without SemIoTic compared to 500, 300, 100, and 30 LoC using SemIoTic (having to develop wrappers for 100%, 50%, 25%, and 5% of the total physical sensors). Based on the results in Fig. 8(b), developing an application using SemIoTic reduces the effort (in terms of LoC) by 97% to 55%. Notice that this experiment measures development effort just in terms of LoC and thus it does not consider other efforts which SemIoTic alleviates. For instance, the effort required to find-/understand/utilize libraries to handle interactions with different protocols, to develop the logic to handle such complex plans, to handle user needs in a more semantically meaningful way, etc.

8 CONCLUSIONS

We have presented SemIoTic, an approach to deal with interoperability issues in smart spaces. At the core, SemIoTic is based on a metamodel to describe spaces in terms of the entities of interest and the underlying IoT devices. We provide a language based on the metamodel to enable application developers to express user requirements focusing on a semantic domain-relevant view of the space. A set of ontology-driven algorithms automatically translate user requirements into possible plans of actions on devices which adapt to the current deployment of devices in the space. Finally, to provide an end-to-end approach we present a proposal for the design of software artifacts that encapsulate the interaction with devices in a protocol-agnostic way. The feasibility of SemIoTic has been shown through a prototype and an application developed to handle a common use case in smart buildings.

ACKNOWLEDGMENTS

This material is based on research sponsored by DARPA under agreement number FA8750-16-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. This work is partially supported by NSF grants 1527536 and 1545071. Also, we would like to thank Rayan Al Atab, Myounghun Han, Dongcheol Jwa, Hyun Ji Lee, and Hye Eun Song for their help in the implementation.

REFERENCES

[1] 2019. SemIoTic. http://tippersweb.ics.uci.edu/semiotic
[2] Mussab Alaa et al. 2017. A review of smart home applications based on Internet of Things. *Journal of Network and Computer Applications* 97 (2017), 48 – 65.
[3] S. Almanee et al. 2019. SemIoTic: Bridging the Semantic Gap in IoT Spaces. In *BuildSys*.
[4] L. Atzori et al. 2010. The Internet of Things: A survey. *Computer Networks* 54, 15 (2010), 2787 – 2805.
[5] D. Bandyopadhyay and J. Sen. 2011. Internet of Things: Applications and Challenges in Technology and Standardization. *Wireless Personal Com.* (2011), 49–69.
[6] G. Bouloukakis et al. 2019. Automated Synthesis of Mediators for Middleware-layer Protocol Interoperability in the IoT. *FGCS* (2019).
[7] A. Castellani et al. 2012. HTTP-CoAP cross protocol proxy: an implementation viewpoint. In *IEEE MASS*.
[8] M. Collina et al. 2012. Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST. In *PIMRC*.
[9] M. Compton et al. 2012. The SSN ontology of the W3C semantic sensor network incubator group. *The Journal of Web Semantics* 17 (2012), 25–32.
[10] B. Costa et al. 2016. Modeling IoT Applications with SysML4IoT. In *SEAA*.
[11] P. Desai et al. 2015. Semantic gateway as a service architecture for IoT interoperability. In *Int. Conf. on Mobile Services*.
[12] I. V. Papaioannou et al. 2006. A QoS ontology language for Web-services. In *AINA*.
[13] N. Georgantas et al. 2013. Service-oriented Distributed Applications in the Future Internet: The Case for Interaction Paradigm Interoperability. In *ESOCC*.
[14] A. Haller et al. 2018. The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation. *Semantic Web* 10 (2018), 9–32.
[15] J. Janak and H. Schulzrinne. 2016. Framework for rapid prototyping of distributed IoT applications powered by WebRTC. In *IPTComm*.
[16] K. Kotis and A. Katasonov. 2013. Semantic interoperability on the internet of things: The semantic smart gateway framework. *IJDST* (2013), 47–69.
[17] M. Langheinrich. 2001. Privacy by Design - Principles of Privacy-Aware Ubiquitous Systems. In *Ubicomp*.
[18] F. Pramudianto et al. 2014. IoT Link: An Internet of Things Prototyping Toolkit. In *UIC*.
[19] Felix Maximilian Roth et al. 2018. XWARE-A customizable interoperability framework for pervasive computing systems. *PMC* (2018), 13–30.
[20] S. Sicari et al. 2015. Security, privacy and trust in Internet of Things: The road ahead. *Computer Networks* 76 (2015), 146 – 164.
[21] E. Sirin et al. 2007. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics* 5 (2007), 51–53. Issue 2.
[22] I. S. Udoh and G. Kotonya. 2018. Developing IoT applications: challenges and frameworks. *IET Cyber-Physical Systems: Theory Applications* 3, 2 (2018), 65–72.