The SemIoTic Ecosystem: A Semantic Bridge between IoT Devices and Smart Spaces

ROBERTO YUS, University of Maryland, Baltimore County, USA
GEORGIOS BOULOUKAKIS, Télécom SudParis, Institut Polytechnique de Paris, France
SHARAD MEHROTRA and NALINI VENKATASUBRAMANIAN, University of California, Irvine, USA

Smart space administration and application development is challenging in part due to the semantic gap that exists between the high-level requirements of users and the low-level capabilities of IoT devices. The stakeholders in a smart space are required to deal with communicating with specific IoT devices, capturing data, processing it, and abstracting it out to generate useful inferences. Additionally, this makes reusability of smart space applications difficult, since they are developed for specific sensor deployments. In this article, we present a holistic approach to IoT smart spaces, the SemIoTic ecosystem, to facilitate application development, space management, and service provision to its inhabitants. The ecosystem is based on a centralized repository, where developers can advertise their space-agnostic applications, and a SemIoTic system deployed in each smart space that interacts with those applications to provide them with the required information. SemIoTic applications are developed using a metamodel that defines high-level concepts abstracted from the smart space about the space itself and the people within it. Application requirements can be expressed then in terms of user-friendly high-level concepts, which are automatically translated by SemIoTic into sensor/actuator commands adapted to the underlying device deployment in each space. We present a reference implementation of the ecosystem that has been deployed at the University of California, Irvine and is abstracting data from hundreds of sensors in the space and providing applications to campus members.

CCS Concepts: • **Software and its engineering → Interoperability**; • **Hardware → Sensor applications and deployments;**

Additional Key Words and Phrases: Internet-of-things, semantic interoperability, open APIs, privacy

ACM Reference format:
Roberto Yus, Georgios Bouloukakis, Sharad Mehrotra, and Nalini Venkatasubramanian. 2022. The SemIoTic Ecosystem: A Semantic Bridge between IoT Devices and Smart Spaces. *ACM Trans. Internet Technol.* 22, 3, Article 76 (July 2022), 33 pages.
https://doi.org/10.1145/3527241

This material is based on research sponsored by DARPA under Agreement No. 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 also supported by NSF Grants No. 1527536, No. 1545071, No. 1952247, No. 2008993, No. 2032525, and No. 2133391.
Authors' addresses: R. Yus, University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250, USA; email: ryus@umbc.edu; G. Bouloukakis, Télécom SudParis, IP Paris Evry & Palaiseau, Greater Paris Metropolitan Region 91000, France; email: georgios.bouloukakis@telecom-sudparis.eu; S. Mehrotra and N. Venkatasubramanian, University of California at Irvine, Irvine, CA 92697-3425, USA; emails: {sharad, nalini}@ics.uci.edu.

1533-5399/2022/07-ART76 $15.00
https://doi.org/10.1145/3527241

1 INTRODUCTION

The **Internet of Things (IoT)** aims to enable the interconnection and data exchange between deployed IoT devices that are steadily growing to billions worldwide [23]. The potential benefits of our spaces (e.g., homes, offices, cities, and communities at large) becoming *smarter* thanks to such IoT devices range from efficiency to comfort including, among others: building security [4], zero-waste sustainable buildings [10], continuous health and wellness monitoring [3], personalized thermal comfort in buildings [43], and intelligent transport planning in cities [34]. However, this comes with potential security and privacy risks [5] as well as a variety of challenges [7] that make achieving such benefits complicated.

Off-the-shelf IoT devices are becoming available in the consumer market (smart lights, smart TV's, thermostats, smart TVs, assistants, etc.) that can potentially be combined to achieve application goals. However, very often, devices from a manufacturer implement proprietary interfaces and protocols—leading to vendor lock-in and hindering seamless integration with devices from other manufacturers. This limitation is due to different companies implementing different underlying interaction protocols (from **Application Protocol Interfaces (APIs)**, socket communication, to messaging protocols (MQTT, CoAP)) and data formats (from completely unstructured strings to more structured XML or JSON). This poses an interoperability challenge due to the heterogeneity of the devices and lack of a single standard interaction protocol. It also pushes application developers to encode communication with each specific device in their applications. As a consequence, reusability of applications is in general not possible. For instance, applications developed for smart homes and smart home devices cannot be used in smart office spaces with smart building devices, even if the purpose of the application is the same—e.g., providing thermal comfort. This also presents a challenge to administrators of smart spaces who need to find applications that are compatible with the deployed IoT devices in their space. Issues of reusability and interoperability have been explored in the past, for example regarding annotation of IoT devices [16, 20, 31] or dealing with the heterogeneity of exchange protocol semantics [9, 24, 42]. Additionally, the design, development, and adoption of standards helps on dealing with those issues.

An additional challenge in the IoT is that of the gap due to the low-level nature of device interaction and raw data and the more high level information needs of users. In general, the raw data captured by sensors requires further processing before it can provide users with relevant insights. For instance, a video camera image, audio captured from a microphone, or even connectivity data captured by beacons are not as useful as identifying who was in the specific image, what was the activity detected by the microphone, or how many people were in the area covered by the beacon. This highlights the "semantic gap" existing between the world of devices and the higher-level concepts that users are interested in. For example, users might want to know the occupancy of a room regardless of whether this data was obtained after analyzing images, audio, or connectivity data (or even the combination of the three). Several ontologies have been proposed to model IoT devices (SSN/SOSA [15, 29], SAREF [18]), as well as specific sensors and systems in buildings, including the **Heating Ventilation Air Conditioning (HVAC)** and lighting systems among others (Haystack [2], Brick [6]).

Finally, the recent legislative support for user privacy (e.g., the **European General Data Protection Regulation (GDPR)** or the **California Consumer Privacy Act (CCPA)**) hints toward the need for more privacy-aware IoT smart spaces. As such, people have to understand what data is being collected/inferred about them in a smart space and potentially express their preferences about such capture of their data [38]. The semantic gap in smart spaces makes this task particularly challenging. Similarly to the example above, users might want to prevent the space from locating them regardless of which specific device is used for that purpose.

The challenges listed above affect all the stakeholders in a smart space: application developers, space administrators, and inhabitants of the space. In today's IoT, those stakeholders bear the onus of bridging the semantic gap while addressing tradeoffs between computation/communication cost, interaction with devices, and even concomitant privacy implications [44]. Even though different approaches have been presented in the literature to deal with some of those issues independently, there is still the open challenge of providing a holistic end-to-end vision of smart spaces from users and applications to devices.

In this article, we extend the middleware solution presented in Reference [48] from an IoT smart space management tool to an ecosystem to facilitate the development, provisioning, and reusability of IoT content across different smart spaces from diverse domains. SemIoTic facilitates the development and provisioning of IoT smart space applications. The system deals with issues of interoperability at the semantic-layer using an extensible and general metamodel, based on the popular SOSA/SSN ontology [26]. This metamodel is used to define static and dynamic aspects of a smart space including the domain (spatial aspects and users), in situ 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, SemIoTic supports *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 an application-oriented access to the smart space with clear definitions of input and output datatypes. The main contributions of SemIoTic 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 SemIoTic ecosystem is based on two main components: (1) a centralized repository of applications developed in a space-agnostic way and (2) SemIoTic-enabled smart spaces that bridge the gap between those applications and the underlying device infrastructure. We detail the different components of the ecosystem and show the reference implementation developed (with respect to its functionality and different technologies used for its development). The reference implementation has been deployed at the University of California, Irvine campus where the system abstracts the view of the underlying network of WiFi Access Points to offer occupancy-based applications.

The rest of the article is organized as follows. We review the state of the art in terms of IoT frameworks, semantic and device interoperability in IoT environments in Section 2. We describe the ecosystem's architecture and use cases based on different stakeholders in Section 3. We detail the meta-ontology that guides the processing of the SemIoTic system in Section 4. We also provide examples of the definition of domain models (e.g., the concepts related to a smart home and smart building) based on the metamodel that can be reused in multiple SemIoTic-enabled spaces. We explain the interfacing capabilities of SemIoTic in Section 5. We introduce a language to define user requests for information, commands, and policies as well as RESTful and SPARQL endpoints

to access information within a smart space. We explain the translation process of user defined high-level actions into IoT device actions in Section 6. We detail the execution of a device action using sensor wrappers and virtual sensors in Section 7. We present experiments (Section 8) to evaluate the benefits of the ecosystem from the perspective of the different stakeholders in a smart space. Finally, Section 9 concludes the article.

2 RELATED WORK

We analyze the state of the art in IoT frameworks and semantic/device interoperability in the IoT.

2.1 IoT Frameworks

Multiple IoT frameworks to facilitate management of devices and development of applications have been presented by both industry (e.g., EvryThng, Node-Red, Google Cloud IoT) and academia [47]. For instance, the IDeA framework [17], 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 [40] 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 Reference [27] 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. Such underlying communication details are abstracted from the Omni [30] middleware so that applications can seamlessly connect to IoT devices. Mortar [21] presents an approach to enable the development of analytics based on sensor data captured in smart buildings. The focus is on enabling the development of sensor applications in categories such as measurement and fault detection and diagnosis. In contrast to our approach, applications need to encode the logic required to consume sensor data and produce the desired inferences. LinkLab [22] is a scalable IoT testbed for heterogeneous devices that supports running experiments and remote development via a web-based IDE. DDFlow [36] provides a macroprogramming abstraction and accompanying runtime that provides an efficient means to program high-quality distributed applications and improve end-to-end latency. Applications are specified through a declarative user interface (extension of the Node-RED IoT system) over a diverse and dynamic IoT network. DynaSense [32] offers a unified approach to applications for accessing data from various data sources, which can be sensors or compositions of other data sources. This is implemented on Android as a middleware system between data sources and user applications. SOUL [28] introduces an *aggregate* abstraction that enables sensors, actuators, and software services to be accessed from mobile applications. This is realized by leveraging an edge-cloud infrastructure to collect, process and disseminate data to mobile applications. Similar to our approach, DynaSense and SOUL allow applications to use high level data without requiring modifications to their code.

The main difference between the above approaches and SemIoTic is that, in general, their focus is on facilitating the understanding of what the underlying device infrastructure is and easing the process to develop applications for it. While they simplify the development of smart space applications by abstracting out communication details, the developer still bears the onus of understanding what different IoT devices do. Also, since applications are still built based on specific IoT devices, they would need to undergo code changes if the underlying device infrastructure changes. In contrast, our approach aims at enabling the creation of applications based on high-level smart space entities by completely abstracting out the existence of IoT devices for application developers. For

instance, a developer using our approach who requires occupancy data of a specific smart space just requests and obtains such data without having to interact with specific IoT devices.

2.2 IoT Interoperability

Ontologies [25] provide a common model for annotating content and thus help systems interoperate. The main standard/well-known ontologies for IoT systems can be divided into two categories based on their focus: general sensor modeling (SSN/SOSA [15, 29], SAREF [18]) and modeling of specific sensors and systems in buildings, including HVAC and lighting systems among others (Haystack [2], Brick [6]). Our approach is based on a general ontology that models the IoT smart space. Following the Semantic Web good practice of reusability, this ontology is an extension to SSN/SOSA and SAREF models, which have become de facto standard to deal with interoperability issues in IoT environments. The extension in our ontology is designed to enable the translation between smart space semantic level data (about properties of interest of different entities) and the smart space device level data (about sensors/actuators and their observations/actuations). In particular, we extend the above models to represent: (1) Virtual sensors, which generate semantic data of a specific type given as input data generated by sensors, and (2) The connection of semantic data to properties of interest.

Several approaches enable protocol-to-protocol bridging, and hence device interoperability, such as the QEST [14] broker for CoAP and REST protocols, HTTP-CoAP proxy [11], 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 or the ***Enterprise Service Bus*** **(ESB)** paradigm [12]. In Reference [35], the authors introduce the ***Lightweight Internet of Things Service Bus*** **(LISA)** for tackling IoT heterogeneity. LISA provides an API for resource-constrained devices that supports access, discovery, registration, and authentication. Devices deployed based on different standards interact via a common communication protocol. An ESB is also used in Reference [13] as the core infrastructure for an event-driven IoT service coordination platform. It enables interconnecting heterogeneous components such as devices acting as event publishers and/or subscribers and users issuing HTTP requests. The authors in Reference [19] introduce a protocol translator that utilizes an intermediate format to capture all protocol specific information. Translators can be placed in local clouds and be used in a transparent and on-demand way. While the authors claim that many different protocols can be mapped with their protocol translator, this work lacks sufficiently general abstractions for enabling its wide application. XWARE [42] implements mediators to translate messages of IoT protocols using an intermediate format. This is designed based on common interaction paradigms described in Reference [24] for SOA. Then, authors of Reference [24] extended their work in Reference [9] to deal with IoT heterogeneity using software abstractions and code generation.

While the above approaches reduce the development effort considerably, they do not take into account semantic layer incompatibilities that are a very common issues in the IoT. The main contribution of our approach compared to 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 SEMIOTIC **ECOSYSTEM**

One of the main ideas behind SemIoTic is that of collaboration and reusability as a mechanism to address heterogeneity issues and facilitate the management of smart spaces and the development/provisioning of smart applications. Hence, we propose an ecosystem around a centralized

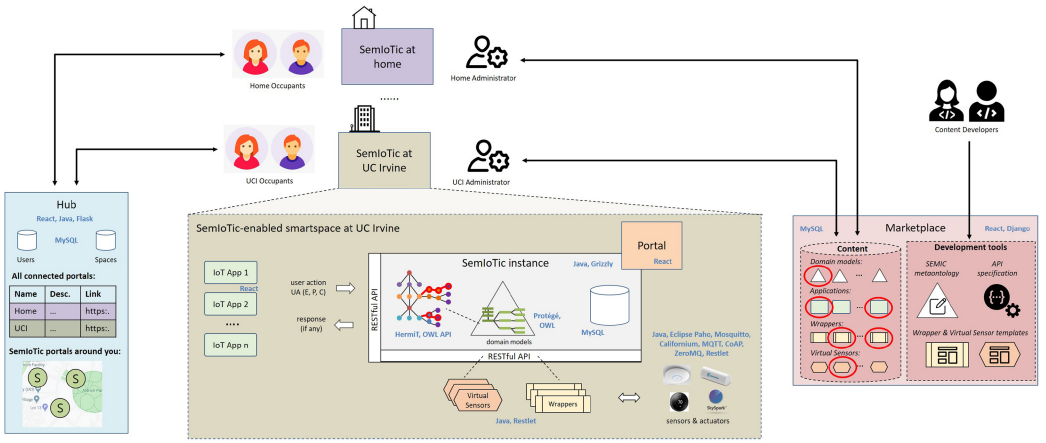


Fig. 1. The SemIoTic ecosystem and technologies used to develop its reference implementation.

repository of resources for smart spaces and a distributed network of SemIoTic-enabled smart spaces. This is similar to the current ecosystem of mobile applications where users select and install apps on their smartphones/tablets from a centralized repository. The ecosystem is based on three main components (see Figure 1): a *marketplace*, SemIoTic-*enabled smart spaces*, and *hub*.

The *marketplace* (see Section 3.1) acts as a repository where people can download SemIoTic as well as content for their smart spaces (e.g., applications and connectors to specific devices). It also hosts a set of tools to facilitate the development of such content. A SemIoTic-*enabled smart space* (see Section 3.2) is a smart space in which an instance of SemIoTic has been deployed and content has been installed. It offers installed applications to users and management tools to the smart space administrator. Finally, the *hub* (see Section 3.3) serves as a catalog of SemIoTic-enabled smart spaces to enable users to discover SemIoTic installations around them.

We envision these three components residing on different servers. Each smart space hosts their own SemIoTic instance that handles its specific IoT devices (this could be hosted on the premise or the cloud). For example, a SemIoTic instance might be installed in a particular smart home or a smart office building. Then, at least one marketplace hosts applications developed for SemIoTic instances. However, in certain situations more than one marketplace might exist. For example, if an organization wants to deploy SemIoTic instances in their buildings and wants to offer smart applications that should only be accessible to their SemIoTic instances. Finally, the hub is a single centralized entity that maintains unique identifiers for users across different SemIoTic instances and information about existing SemIoTic instances.

The main stakeholders and their interactions with the SemIoTic ecosystem are as follows:

- *Content developers*, who develop *domain models* (i.e., extensions of the SEMIC ontology to define spaces such as homes, buildings, malls, etc.), *device wrappers* (i.e., connectors to specific devices), *virtual sensors* (i.e., software to process raw data into semantically enriched information), and *applications* for the SemIoTic ecosystem. They use the content development tools provided in the marketplace for this task. Once the content is developed, they upload it to the marketplace.
- *Administrators of a smart space*, who download content from the marketplace for their SemIoTic-enabled space and install it to configure the system (e.g., to define what types of sensors are deployed, what information do they collect, etc.) and to offer applications to their inhabitants. They manage the smart space through SemIoTic expressing their needs

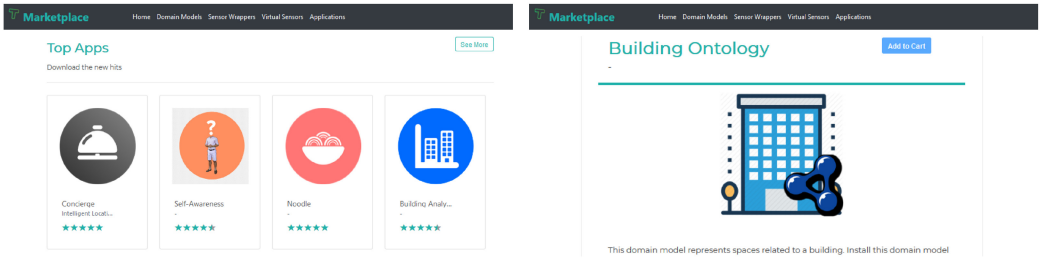


Fig. 2. The SemIoTic Marketplace.

such as controlling the temperature of a portion of the space depending on its occupancy, capturing information related to the location of visitors in an office space, and so on.

- *Application users*, who are the inhabitants of a smart space and obtain information/services about it through the applications installed on the space. Additionally, as SemIoTic is built following a privacy-by-design approach, users/inhabitants of the space are expected to express privacy preferences/policies about which data can be captured about them. Following the abstraction model of SemIoTic, such policies are also expressed using higher-level semantically meaningful concepts that abstract the underlying device infrastructure.

These stakeholders are similar to those in the context of mobile apps. The main difference is that in the mobile world, the administrator of a device (e.g., a smartphone/tablet) is also, in general, the user of the device. In our setting, even when this might happen in some situations (e.g., if SemIoTic is installed in a smart home), this is not always the case (e.g., in most of the buildings there is a figure of the building administrator who could be the administrator of the smart space). Another difference is that the owner of the mobile device is also its user and privacy preferences on the mobile device are only defined by them. In our context, there can be multiple users in a smart space whose data might be collected to process actions posed by others. Hence, SemIoTic supports the definition of privacy preferences with respect to the handling of their data when needed by others.

The reference implementation of the SemIoTic ecosystem leverages state-of-the-art technologies to implement the different components (see Figure 1 where each technology used is included). The complete system and sample content (domain models, wrappers, virtual sensors, and applications) developed are available at Reference [1]. In the following, we describe each of the components in details.

3.1 SemIoTic Marketplace: Discovering Content

The marketplace serves as the repository from which smart space administrators can retrieve content for their spaces. The marketplace is a web application (developed using the React[1] JavaScript framework for its frontend and the Django[2] web framework along with MySQL for its backend) that shows the content along with ratings, comments, screenshots, and other information to help administrators to browse and select appropriate content. Figure 2(a) shows a screenshot of the main page of the marketplace with the most downloaded content by category. Figure 2(b) shows the information displayed about a specific content piece. We consider SemIoTic content to be the following:

- *Domain model*, which is an OWL ontology that extends the SemIoTic meta-ontology (semic) and defines different concepts related to a specific domain. A domain (e.g., smart home,

[1] https://reactjs.org.
[2] https://www.djangoproject.com.

office, mall, airport, city, etc.) might include specifications about space-related concepts such as the types of spaces within the domain (e.g., for a smart home the concepts of home, apartment, house, floor, bedroom, living room, etc.) and interesting properties related to them (e.g., temperature of a room, occupancy of a building, etc.), people-related concepts (e.g., types of people in a university domain are professors, staff members, students, etc.) and their relevant properties (e.g., heart rate and location of a person), and finally device-related concepts (e.g., typical sensors found in a smart building such as HVAC, WiFi APs, thermostats, etc.).

- *Sensor wrappers*, which are software components (encoded in any programming language of choice) that encapsulate the code required to enable SemIoTic installed in a particular space to interact with a specific devices (e.g., Amazon Alexa, Google Nest, etc.). Every wrapper offers a set of APIs that SemIoTic can call to obtain information (in the case of sensors) or perform actuations (in the case of actuators).
- *Virtual sensors*, which are software components, similar to sensors wrappers, that can process data and output higher-level information (e.g., process images to infer the number of people in a room). We refer to them as virtual sensors as, like physical sensors, they output observations.
- *IoT Applications*, which offer functionalities related to the smart space to its inhabitants.

We envision developers to focus on specific content pieces (e.g., applications) instead of handling the development of the complete flow (i.e., from software to manage devices to applications). This way we compartmentalize the development task. For instance, the manufacturer of a specific sensor can develop a wrapper to interact with it, an ontology engineer can develop domain models for specific smart space types, and app developers can develop, or enhance their apps, to communicate with the SemIoTic-enabled smart space. The latter can develop space-agnostic applications by posing high-level information requests to each specific SemIoTic smart space and letting the system process it taking into account the underlying device infrastructure available.

To facilitate the development of each content piece, the marketplace offers a development toolkit. This contains the OWL file and specification of the semic meta-ontology that can be imported in an ontology editing tool (e.g., *Protégé*[3] [46]) to be extended to create specific domain models. We detail the meta-ontology in Section 4. The toolkit also contains the complete specification of the interfaces of SemIoTic (we detailed this in Section 5). Application developers can use such specification to construct the calls to the underlying SemIoTic where the application is deployed to retrieve information about the space. To create space-and-device-agnostic applications, the calls are made using the SemIoTic language including high-level concepts. Finally, the toolkit contains wrapper and virtual sensor templates that include code required to: (1) create and advertise endpoints so that SemIoTic can communicate with them, and (2) encapsulate the communication with popular IoT interaction protocols (e.g., MQTT, CoAP, web socket, etc.). Hence, the developer of a wrapper/virtual sensor just have to download the template and fill in the code required to perform their specific processing (as we will describe in Section 7).

To facilitate the installation of content on any SemIoTic-enabled space, we require it to be uploaded using *container* technology. We use Docker[4] to automatically build images by reading the instructions from a `Dockerfile`. This contains all the commands that a content developer could call on the command line to assemble an image. For example, if an application requires a backend with a MySQL database and an HTTP Apache server, then the specific commands to install and execute those systems in the host SemIoTic instantiation will be specified in the Dockerfile.

[3] https://protege.stanford.edu.
[4] https://www.docker.com.

From the point of view of smart space administrators, the marketplace enables them to download SemIoTic along with the appropriate content. When choosing content, the marketplace indicates dependencies among different pieces of content and automatically adds them to the order. For instance, if an application has been build considering specific concepts of a smart home (e.g., the concept of different rooms such as living room, bedroom, etc.) then the specific domain model for the smart home that the application requires will be associated to it. Also, if the application requires some high-level data such as occupancy of the rooms, it will be associated to virtual sensors that can produce that information. Similarly, a wrapper to connect to a specific device (e.g., Nest thermostat) will depend on the concepts associated with the device and the observations/actuations it can produce (e.g., thermostat, temperature observation, control of temperature actuation, etc.) and thus it will be associated to a specific domain model for those devices (e.g., the domain model that defines smart home devices in this example). The administrator has to also be aware of the devices deployed in their smart space to select the wrappers needed to connect with them.

To facilitate the deployment of SemIoTic and the selected content in smart space, the marketplace generates a specific container for that administrator that they can easily execute in the server they selects to host their SemIoTic instance. This packaging is done based on Kubernetes[5] (an open-source system for automating deployment, scaling, and management of containerized applications, wrappers, etc.). Each of the individual containers (e.g., SemIoTic runs in a container, each selected application/wrapper/virtual sensor in a different container each) are specified in a YAML file that Kubernetes running on the host server can interpret to orchestrate the installation of required software, configuration, and execution automatically.

3.2 SemIoTic-Enabled Smart Space: Managing IoT Devices

Once SemIoTic is installed in a smart space, it becomes a SemIoTic-enabled smart space in the ecosystem. A deployment of SemIoTic in a specific space handles its underlying IoT devices and offers smart applications to its inhabitants. The deployment handles *user-defined actions*, posed by applications, space administrators, and/or users, of three types: (1) *requests* for dynamic or static information about the space (e.g., 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 the location of a person). The architecture of a SemIoTic-enabled smart space is based on three main modules (see Figure 3):

- *Model Handling*, which enables administrators to describe the smart space in terms of types of *spaces*, *users*, and *devices*, as well as specific instances of those types.
- *User Action Handling*, 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 smart space. Since a smart space may include multiple IoT devices to execute an action, this component generates possible execution plans by taking into account the description of devices in the model and the ones that are currently deployed. Then, it leverages information related to the devices (such as their cost and QoS) to select an optimal plan.
- *Device Action Handling*, which accesses 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.

[5] https://kubernetes.io.

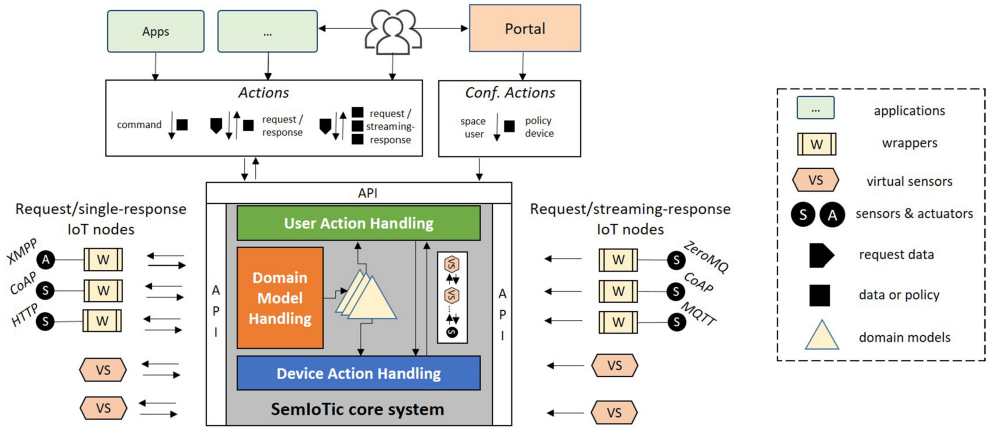
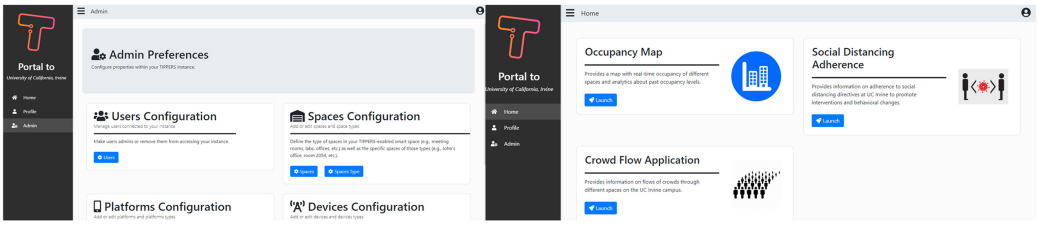


Fig. 3. High-level architecture of a SemIoTic-enabled smart space.



(a) Configuration of SemIoTic instance (b) IoT apps of SemIoTic instance

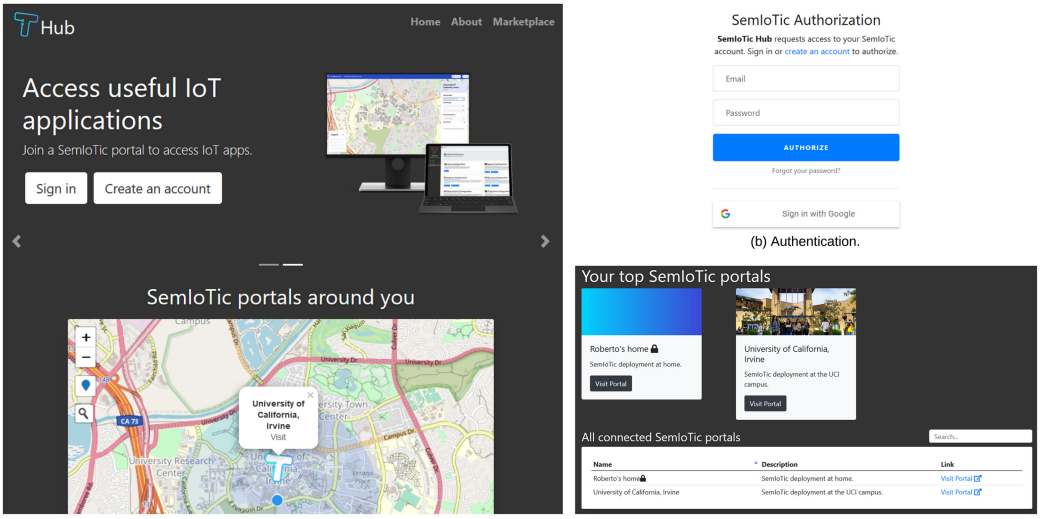
Fig. 4. The SemIoTic Portal Web Interface.

Additional components of SemIoTic are (1) a set of APIs that enable applications pose their actions (as explained before) and (2) a **Graphical User Interface (GUI)** to administer the space and access its applications. The latter, to which we refer as the *Portal* to the SemIoTic-enabled space, is a web application (developed using the React framework). It offers functionalities to enable the smart space administrator instantiate the domain model(s) selected for the space (as shown in Figure 4(a)). For instance, the administrator can use the Portal to fill in information about the name of rooms in the space, their type, and their static properties (e.g., their capacity, extent, etc.), the specific parameters associated with devices deployed (such as their IPs, names, descriptions, location within the space, etc.). Inhabitants of the space can use the Portal to register their own information (e.g., their offices, profiles, sensors they own, etc.), define their privacy policies (e.g., do not share my location with advertisement applications), and discover applications deployed in the space (see Figure 4(b)).

3.3 SemIoTic Hub: Discovering Smart Spaces

The final component of the SemIoTic ecosystem is the *Hub* (see Figure 5(a)). This web application's purpose is twofold by enabling users (application developers, space administrators, space inhabitants) to (1) register into the ecosystem and obtain an identifier that works across SemIoTic deployments and (2) discover SemIoTic deployments around them. For the former, the Hub implements an authorization framework based on OAuth 2.0,[6] the industry-standard protocol for authorization, to manage both authorization of users and of applications (see Figure 5(b)). The purpose

[6] https://oauth.net/2.



(a) Hub main page.

(b) Authentication.

(c) Smart space discovery.

Fig. 5. The SemIoTic Hub.

is to establish secure communications where applications deployed in a specific space performing a user action on behalf of a specific user have to be authorized to do so. In addition, usage of global identifiers for users make it possible for them to maintain the same profile across spaces.

The main purpose of the Hub is to serve as a platform for people to discover SemIoTic-enabled smart spaces and access the ones they inhabit. When a space administrator configures SemIoTic deployed in their space, they can define its visibility to be either public (which means that people using the Hub can discover and join it) or private (which means that people can only access it through an invitation). We expect deployments in public spaces such as a mall, an airport, a city, a university campus to be defined as public and hence discoverable, while other spaces such as homes or offices to be private. Once a user joins a specific deployment, either by discovering it through the map interface in Figure 5(a) in the case of public deployments or through an invitation, this association is retained in the Hub to enable them to manage their presence in the different deployments from a centralized point. As shown in Figure 5(c), a SemIoTic user using the Hub can view their top visited smart spaces—i.e., the ones they visits frequently such as their office, home, and so on—as well as all the smart spaces that they have been part of—i.e., smart spaces visited less frequently such as a shopping mall, the city hall, and so on. Accessing any of those SemIoTic-enabled smart spaces from the catalog offered by the Hub redirects the user to that specific deployment's Portal from which the user can access the smart applications offered by the space.

4 SEMIC META-ONTOLOGY

SemIoTic bases its processing on a meta-ontology (semic) that describes the main elements in any smart space including entities (such as people and spaces), devices, observations and actuations. semic is an extension on the SSN/SOSA ontology [26] to support the automatic translation of user actions defined around higher level concepts into device actions at a lower level. In the following, we introduce the main components of the semic ontology (see Figure 6, which shows its main classes and properties and their alignment to existing ontologies). Additionally, in Section 4.1, we show how the semic ontology can be used to define a domain model (i.e., a smart campus) and a specific instantiation for a smart space (i.e., one of the buildings in the campus).

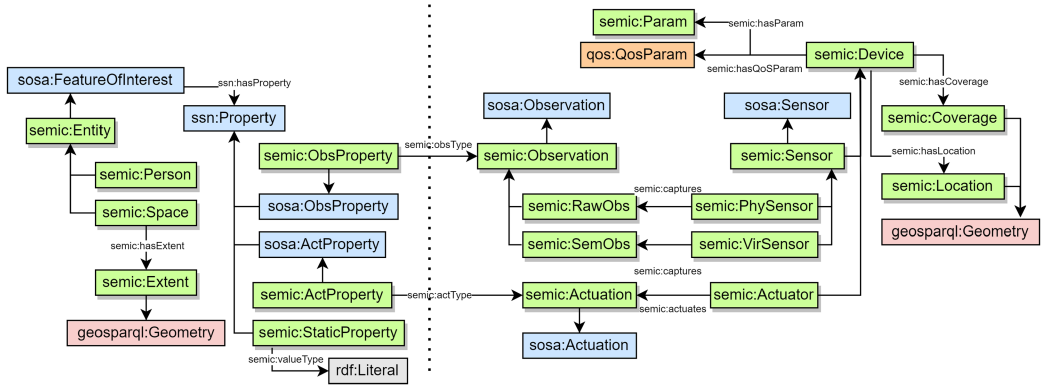


Fig. 6. Overview of the SEMIC meta-ontology to support the description of a smart space.

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 that is used to describe their geographic extent in an X-Y-Z coordinate system. SEMIC:`Extent` is defined as a subclass of the spatial object class from the GeoSPARQL ontology [8]. Hence, the associated geometry can be defined using GeoSPARQL's geometry class. SEMIOTIC uses GeoSPARQL built-in functions (e.g., `funcs:sfIntersects`, `funcs:sfWithin`) related to geometries to perform spatial reasoning (e.g., to check whether the coverage of a sensor intersects with the extent of a space).

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 physical sensor "Thermometer," which observes the subclass of raw observation "Temperature." Similarly, one could define the subclass of virtual sensor "Image to Occupancy counter," which observes the semantic observation "Occupancy" after consuming "Image" observations.

We use the concept of observation, connected to sensors, along with the aforementioned concept of observation property, connected to entities, to bridge the gap between high-level and low-level concepts (done through the semic:obsType property that connects both concepts). Similarly, the semic:actType property connects an actuatable property of an entity to the actuations performed by actuators. This way, SemIoTic can infer that a thermometer observes temperature data, which is required by the temperature property of a room. To further enable SemIoTic to infer which specific devices in a SemIoTic-enabled smart space can be used to obtain data, semic introduces two concepts: coverage and location. These represent the coverage area of a device (e.g., the view frustum of a camera) and its location in the smart space. Both concepts are subclasses of geosparql:Geometry as in the case of the extent of space.

Finally, semic supports the definition of *Quality-of-Service* features for IoT devices. We integrated semic with the QoS ontology presented in Reference [37] to represent metrics related to devices (both physical and virtual). QoS includes vocabulary to describe information such as the value, unit, and type of a metric. Also, it contains classes for common QoS parameters such as performance, throughput, cost, reliability, accessibility, accuracy, and so on. The QoS parameters associated with a device will be leveraged by SemIoTic when choosing a plan that satisfies the requirements of the user.

4.1 Defining Domain Models based on semic

The semic meta-ontology covers the basic concepts and properties relevant to any smart space. It is designed to be extensible and enable others to define space-specific domain models on it. This task includes two main steps: (1) creation of domain models that extend semic to specify domains such as an office building, a university campus, a smart home, a smart city; (2) instantiation of the selected domain model(s) for a specific SemIoTic-enabled smart space (e.g., by defining the specific information of the buildings on the UCI campus and the sensors deployed on them).

The definition of a domain model is expected to be done by domain experts who would share such extensions of semic in the marketplace. For instance, we loaded the semic OWL file on *Protégé* and defined two domain models: a university campus and a smart home. As depicted in Figure 7, the smart campus has spaces such as *meeting rooms*, *classrooms*, and so on, and user profiles such as *professor* and *student*; whereas the smart home has spaces such as *living room* and *kitchen* and user profiles such as *household members* and *visitors*. People and spaces are defined as subclasses of the semic:Entity class, which specializes the sosa:FeatureOfInterest class. Along with the definition of those entities, we defined also common static and dynamic properties for spaces such as *occupancy*, *temperature*, *capacity*, *regulation of temperature* and for people such as *location* and *heart rate*. Each property is a subclass of the semic:Property class and is linked to the entity through the corresponding ssn:hasproperty element. Also, for observable properties, we defined the types of observation they take as values (e.g., occupancy observations for the occupancy property).

We also defined types of physical IoT devices that are generally used in each domain. For instance, we defined *cameras* and *thermostats* in the smart home and *HVAC*, *Bluetooth beacons*, and *WiFi APs* in the smart campus. For each device, we defined the observation types it can capture (in the case of sensors) or actuation it can perform (in the case of actuators). For instance, we defined that a *WiFi AP* captures observations of the type *Connectivity* using the semic:captures property.

The instantation of a domain model is expected to be performed by the administrator of the smart space where SemIoTic is deployed after installing the appropriate domain model. This is done through the Portal frontend using a GUI interface that shows forms that the administrator can fill in (as explained Figure 4). For instance, Figure 8 shows part of the instantiation of the University Campus domain model to define a specific space in our campus. The figure shows the definition of a room (Room111) along with its extent (which is defined as polygon based on the Well-known text

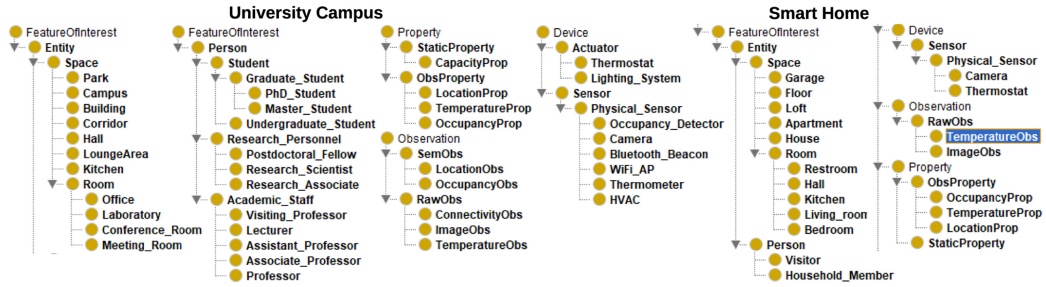


Fig. 7. Snippets of two domain models based on SEMIC.

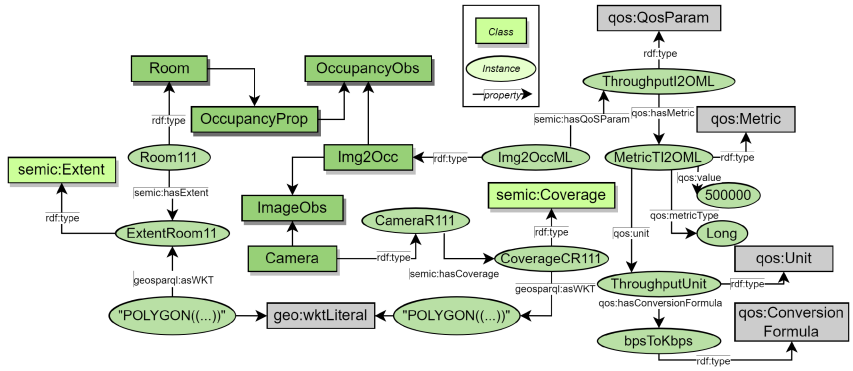


Fig. 8. Snippet of the instantiation of a domain model for a specific smart building.

–WKT– standard in GeoSPARQL). Then, it shows the definition of a specific camera (CameraR111) along with its coverage (again defined using WKT). Finally, it shows the definition of a specific virtual sensor installed in that SEMIOTIC deployment, which uses an ML algorithm to translate images into occupancy. Note that the figure shows, as an example, the definition of one of the QoS parameters associated with the virtual sensor. In this case, it shows the definition of the throughput of the virtual sensor based on the QoS ontology.

5 INTERACTING WITH SEMIOTIC

We explain the SEMIOTIC **Action Language (SAL)** used to express user actions and the different mechanisms to express such actions. Also, we show an example application developed using SAL.

5.1 Action Language

SEMIOTIC internally uses the SAL that enables the specification of user actions (denoted in the rest of the article as *UA*) to express either a request for data (*UR*), a command (*UC*), or a policy (*UP*). The general format of such actions includes the following definitions:

- *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 "Public spaces" or a specific instance such as "Room 111."
- *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, or that has to be protected (i.e., $\langle \rho_i$, `rdf:type`,

semic:Property⟩). For example, "location," "heart rate" in the case of a person, or "occupancy," "capacity," and "control temperature" in the case of a room.

- *Conditions*, C, which is an expression that has to be satisfied as a condition to perform the user action. We assume that the condition expression contains a Boolean expression including one or more properties (e.g., "when I am in a private space," "when the occupancy of the room is greater than the capacity").
- *Parameters*, PR, which is a set of parameters (involving both QoS and/or parameters related to the observation/action to obtain/perform). For example, the definition of the measure unit for the temperature values to obtain (e.g., Fahrenheit or Celsius). The definition on restrictions on the QoS parameters the user/application is willing to tolerate when processing a user request/command are based on the QoS module of semic. In particular, the SemIoTic instance in a smart space checks all the QoS definitions in its instantiated ontology (e.g., the ones in the example in Figure 8) and offers those as options when creating a user request.

User policies (based on the model in Reference [39]) contain two additional elements: (1) *Interaction to control*, *ic*, which refers to the process (i.e., capture, retention/storage, usage/processing, and sharing) that can/cannot be applied on the property of interest. (2) *Preferred action*, *pa*, which is the action that has to be applied to the previous interaction (i.e., accept or deny).

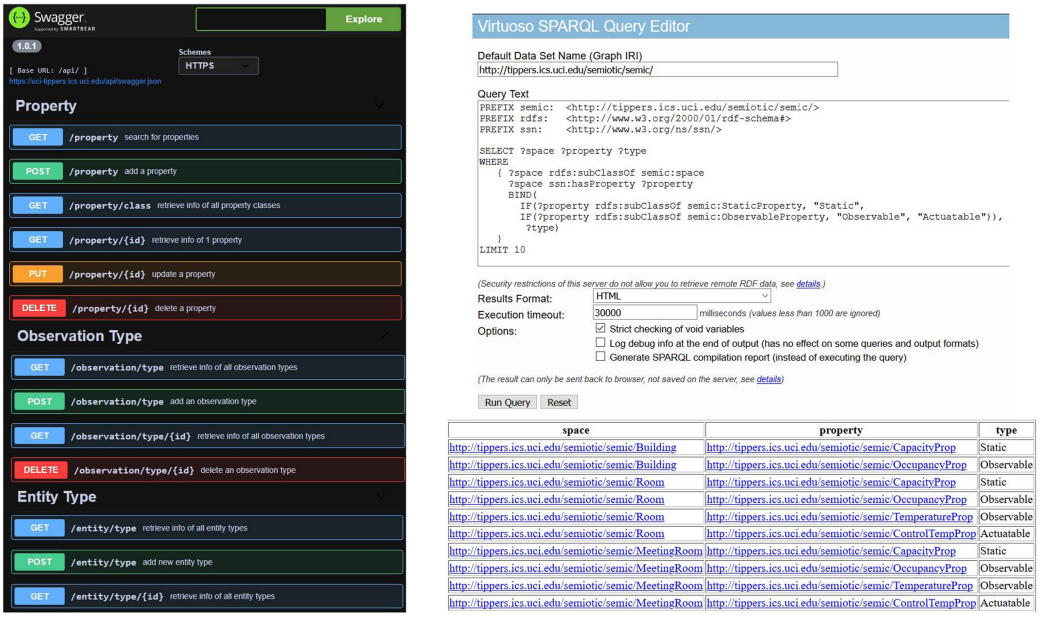
Hence, formally a UR or UC can be defined as the tuple $\langle E, P, C, PR\rangle$ while a UP is the tuple $\langle E, P, C,$ ic, pa$\rangle$. Given this sal definition, we can express actions such as: "retrieve the current location of John and Mary" (⟨John, Mary, LocationProp, ∅, ∅⟩), "retrieve the current occupancy of room 111 in less than 3 s and without conflicting with any user policy" (⟨Room111, Occupancy, ∅, responseTime<3 and policyConflict=0⟩), "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 that 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, is the tuple $\langle D, \varepsilon, a, PR\rangle$ where the different elements represent: (1) *Device*, D, to perform the action (this can be a class of devices or a specific instance). (2) *Entity of interest*, ε, that the device should observe/actuate upon (this has to be an instance of an entity). This parameter is optional and used if the device coverage involves more than one entity and the action has to be performed in only one of them. (3) *Type of observation/action*, a, to request/command the device. (4) *Parameters*, PR, as in the UA. For example, the following SR captures temperature data from a thermometer in Celsius ⟨thermometer111, room111, TemperatureObs, unit=Celsius⟩.

5.2 SemIoTic API

SemIoTic provides application developers with a set of APIs. The APIs include, in addition to the one that enables users to pose actions defined using sal, other useful interfaces to retrieve information from the instantiated semic in the specific space where the application is being executed. The APIs are defined using the REST architectural style and the parameters, as well as the results, are expressed in JSON format. This way, applications can be developed using any client-based Web (e.g., Javascript, HTML/CSS) or mobile technology (e.g., Android, iOS) that supports RESTful invocations. Internally, we encode the APIs using the popular Grizzly framework.[7] We use the open-source framework Swagger[8] to help content developers design, build, and consume these RESTful

[7] https://javaee.github.io/grizzly.
[8] https://swagger.io.



(a) (b)

Fig. 9. (a) Snippets of SemIoTic API specification and (b) SPARQL endpoint and results for a given query.

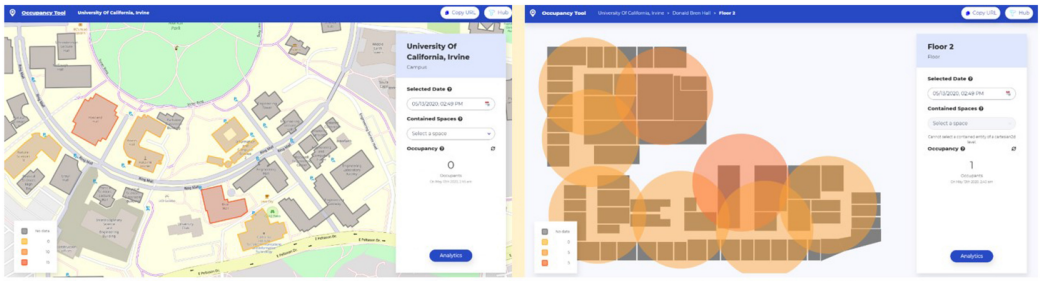
endpoints. Figure 9(a) shows a snippet of the OpenAPI specification[9] of the APIs.[10] For instance, Figure 9(a) shows the APIs an application developer can use to retrieve the types of entities defined in a SemIoTic-enabled space. This is done through the RESTful API `url:port/entity/type`, where the `url:port` part defines the specific SemIoTic-enabled space and that returns all the types of spaces and people (i.e., subclasses of the semic:`Entity` class) in JSON format.

Additionally, SemIoTic offers a SPARQL query endpoint implemented using Virtuoso.[11] The SPARQL query language [41] is an RDF declarative query language (similar to SQL) on RDF triples. We offer this additional endpoint for more advance application developers that want to perform queries on the underlying domain model. For instance, Figure 9(b) shows the SPARQL endpoint of SemIoTic when running a SPARQL query to retrieve subclasses of semic:Space class along with their properties and their type (on the top) and the results obtained (on the bottom).

5.3 Developing Applications Interfacing with SemIoTic

One of the main goals of SemIoTic is to facilitate the development of applications for smart spaces. This is achieved by abstracting out the IoT device infrastructure (through the usage of the SAL language) to enable space-agnostic applications to be used in different smart spaces. Developers can use any client-based Web or mobile technology to build applications and express their actions via RESTful APIs or SPARQL queries. We developed several sample applications based on this model. The main of those is an occupancy tool, developed using React, that showcases an exploratory discovery of occupancy levels at any space. This application is currently in use in our main deployment at UCI (Figure 10 shows screenshots of the application running at UCI).

[9] https://swagger.io/specification/.
[10] The complete specification is available at Reference [1].
[11] https://virtuoso.openlinksw.com.



(a) IoT spaces (b) Floor level IoT devices

Fig. 10. Example SemIoTic IoT application developed using SAL.

The application first retrieves the instances of spaces defined in the SemIoTic-enabled smart space through the `/space` GET REST API. In our deployment at UCI, this returns each of the buildings on campus, their floors, and rooms. Then, following the geographical model in SEMIC, it renders those spaces on a map (see Figure 10(a)). Along with that information, the application requests SemIoTic for the occupancy levels of each of the buildings on campus and color codes it on the map. To do this in a general way, the application obtains the hierarchy of spaces using the `/space/type` GET REST endpoint. In this case, the first node is "Campus" class, followed by a "Building" class underneath. If this was running, for instance, in a smart home, then the first node would be "Home" and it could be followed by "Room" and hence, the application would use "Room" instead of "Building." Then, it poses a SAL request ⟨`Building, OccupancyProp,` ∅, ∅⟩ to retrieve the occupancy of all buildings on campus. When a user clicks on one of the buildings, the application repeats the same process for the spaces defined within (e.g., floors). This process is performed recursively for each space. For instance, Figure 10(b) shows the occupancy levels at the 2nd floor of the Donald Bren Hall building at UCI. Note that in the figure the application is rendering the spaces within the floor, which are in this case rooms as well as the regions of coverage of WiFi APs deployed there (which the application retrieves as simply "Regions"). Thus, the application renders the occupancy values of rooms (gray as the information cannot be computed in that case) and of the regions (colorful as this information can be estimated using a virtual sensor that leverages connectivity events from WiFi APs [33]).

Hence, the highest complexity involved in the development of the application is the development of the graphical user interface and navigational logic. Retrieving data (e.g., space hierarchy, space instance, occupancy data) is simply done through a few lines of codes required to make an API call. Additionally, the same application, without any code modification, can run in other SemIoTic-enabled smart spaces with different domain models and different underlying sensor infrastructures as we describe in Section 8.

6 TRANSLATING SAL ACTIONS

We explain how SemIoTic translates a SAL user action defined at a higher level into the set of actions required from the underlying IoT devices. Then, from the set of possible plans involving different devices, SemIoTic chooses an optimal one and executes it (as it will be explained in Section 7). The general translation process, illustrated using the BPMN inspired data structure in Figure 11, involves four steps. We will use the user command ⟨`Room, ControlTempProp, OccupancyProp>0.5xCapacityProp`, ∅⟩ in Figure 12 as a running example to explain the translation process.

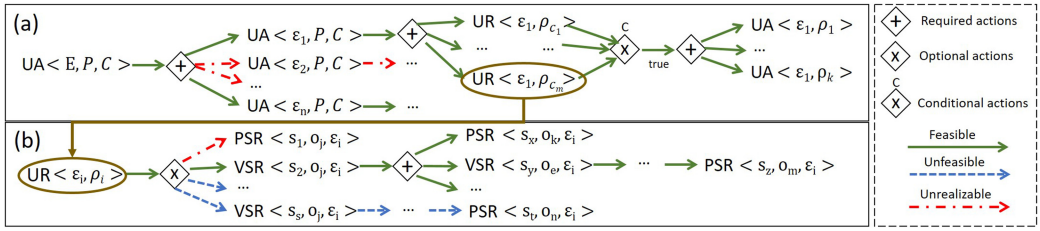


Fig. 11. Structures generated to handle a User Action: (a) flattened tree for UA and (b) execution plans generated for a UR.

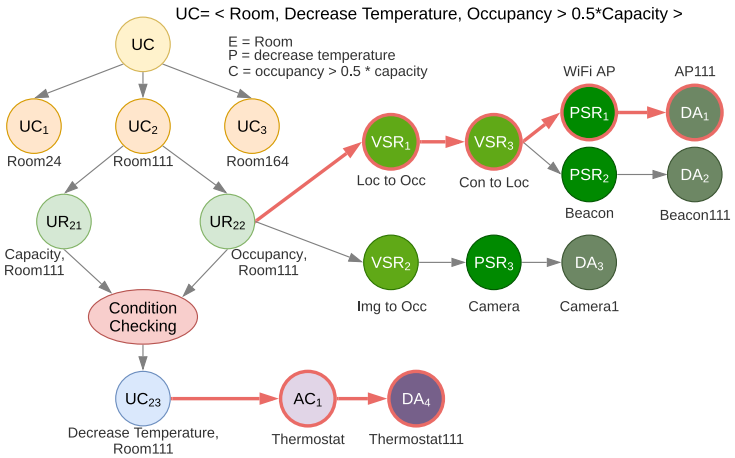


Fig. 12. Example of the translation of a user command.

6.1 Flattening

Complex user actions, UAs (e.g., containing conditions), require the processing of other internal actions to resolve them. For instance, in the command of our running example (see Figure 12), we need to execute requests to obtain the occupancy of specific rooms to determine whether the temperature has to be decreased. 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, $UA =< E, P, C >$,[12] and generates a tree structure, $\mathcal{T}_{UA}$, that contains the high-level plan required to process it in terms of other UAs. The $\mathcal{T}_{UA}$ generated in this step (see Figure 11(a)) 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 "Room" in our running example—Room24, Room111, and Room164). 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 capacity of the room). This is a set of URs such that $UR_{ij} =< \varepsilon_i, \rho_j, \varnothing >$, where ρ_j refers to the jth property needed to compute a condition $c \in C$. For instance, Figure 12 shows the two requests UR_{21} and UR_{22} to obtain the capacity and occupancy of Room111. Note that conditions require data to evaluate them and such data is obtained through requests and not commands.

[12]Note that in the following, we will omit parameters associated with actions (which are passed down from the user action to the leaves of the plan tree) to improve readability.

(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 of spaces to decrease their temperature). Thus, it contains the UAs leaf nodes such that $UA_j =< \varepsilon_i, \rho_j, \varnothing >$ 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. Extracting properties and conditions is straightforward but extracting entities in E requires an additional step, since the UA might defined them semantically (e.g., "Room"). Internally, the method $ExtractEntities(UA, \mathsf{M})$ returns the same set E if it contains a set of entities such that $\langle \varepsilon_i$, `rdf:type`, `SEMIC:Entity` $\forall \varepsilon_i \in E\rangle$ (e.g., if $E =< Room1, Room2 >$). Otherwise, if $\langle \varepsilon_i$, `rdfs:subClassOf`, `SEMIC:Entity`$\rangle$ $\forall \varepsilon_i \in E$ (e.g., if $E =< Room >$), then the method uses a *Description Logic reasoner* [45] to obtain any instance $m_j \in \mathsf{M}$ such that $\langle m_j$, `rdf:type`, $\varepsilon_i\rangle$. Thus, by taking the hierarchical nature of the representation into account, 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=⟨`Room 111`, `rdf:type`, `Meeting Room`⟩, then m_j will be returned for a UA where ε_i ="Room"$\in E$.

6.2 Execution Plans Generation

After flattening, $\mathcal{T}_{UA}$ contains the set of internal UAs to be processed to handle the user action, i.e., the different UA_i in each level of the flattened tree of Figure 11(a). Each $UA_i \in \mathcal{T}_{UA}$ will require a set of device actions, DA, to be executed. Note 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, Figure 12 shows that to retrieve the occupancy of "Room111," the previous step generated UR_{22}. The goal of the execution plans generation step, in this example, will be to determine that such occupancy can be computed using two different virtual sensors (VSR_1 and VSR_2) utilizing different types of underlying sensors deployed in the space.

The execution plans generation step expands $\mathcal{T}_{UA}$ by extending each UA_i with a set of execution plans as $\mathcal{T}_{UA_i}$. Figure 11(b) shows the structure of the execution plans for a particular $UR =< \varepsilon_i, \rho_j, \varnothing >$ that is used to expand the highlighted node in Figure 11(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) sensors that can obtain the input required by s_k. (3) The leaf nodes of the tree contain only PSR. In our running example of Figure 12, we can see that the constraints are met. First, and focusing on UR_{22}, we see that each level of its subtree contains either virtual sensor requests (i.e., VSR_1, VSR_2, and VSR_3) or physical sensor requests (i.e., PSR_1, PSR_2, and PSR_3). Also, each virtual sensor request is always followed by one or more virtual/physical sensor requests and the leaf nodes are always physical sensor requests. For instance, VSR_2, which transforms image data into occupancy, is followed by PSR_3, which obtains images from cameras, and this is the leaf node at this step.[13]

Thus, this step iterates for each $UA_i =< \varepsilon_i, \rho_j, \varnothing >$ node in $\mathcal{T}_{UA}$ and extracts 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). Note that more than one device d can be obtained for the same property. For instance, two different virtual sensors could retrieve the occupancy of a room using different inputs (e.g., location data or video camera feeds). For each d retrieved, the algorithm creates a device action, DA_d where

[13]Note that the figure shows another node after it, DA_3, but this node is added in the next step.

$DA_d = SR_d$ 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 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*.

6.3 Plan Realizability Checking

A plan could be *unrealizable* given the deployment of devices in the scenario. For example, in Figure 12 consider the plan involving the request to the virtual sensor that translates images into occupancy, VSR_2, which takes input from a request to a camera sensor, PSR_2. That plan will be unrealizable for a specific room that is not in the view frustum of any video camera.

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 general translation tree of Figure 11, we have marked some of the plans according to their realizability as an output of this step. Note 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 algorithm 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(ε_i). The checkCoverage function returns all devices 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) and can cover the entity ε_i by using the semic:Extent associated with ε_i and the semic:coverage property associated with d. In particular, the function runs a GeoSPARQL query that retrieves all devices whose coverage extent intersects with the extent of the instance. Since the definition of extents is based on the GeoSPARQL ontology (see Section 4), the query uses the built-in geof:sfIntersects method that computes such intersections. For example, in Figure 12 PSR_3 requires images from cameras that can cover "Room111." The result of executing checkCoverage("Room111") returns "Camera1," since its coverage/view frustum has been defined as a triangle that intersects the extent of the room. Also, if the entity is an entity whose location/extent is not known at the moment (e.g., for a specific person), then the checkCoverage function will return all devices covering any portion of the smart space to maximize the chances of retrieving the required data.

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 device action $DA_k =< d_k, \varepsilon_i, o_j >$, which gets added as child to $\mathcal{N}_{DA}$.

6.4 Feasibility checking

Some of the execution plans generated can be realizable but *unfeasible*. This can be determined regarding different criteria including how costly (e.g., in terms of processing time or even economically) the execution would be or whether it would conflict with existing user policies. For instance, in Figure 12 the request for occupancy of Room111 can be satisfied by capturing video data and then processing it or by capturing connectivity data. If the user defined that processing cost is an important aspect, e.g., by defining a parameter such as $timeCost < 1$ in the original request, then

processing video will be unfeasible if the QoS parameter of the virtual sensor that handles the processing is greater than 1 s.

SemIoTic computes a *feasibility score* for each device action $DA_k =< d_k, \varepsilon_i, o_j >$ in each plan and adds it to the metadata of its associated node $\mathcal{N}_{DA_k}$. This feasibility score is used to compare whether a plan is more desirable than others. 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 include processing time (as number of seconds), quality of the answer (from a scale from 0 to 1 where the lowest the highest the quality), economical cost (as amount of dollars), and conflicting policies (as the number of policies that prevent the access of such device). The previous metrics are mostly static values that are defined in the domain model per device, as showed in the example in Section 4.1, except for the conflicting policies metric. This is computed by using the `checkConflict(`d`)` method, which uses the set of all UPs defined by users of SemIoTic. The method checks whether policies restrict access to d. Given a user defined policy UP_n the same translation 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 `checkConflict()` returns true. Then, the number of policies that restrict access to each device are counted and this information is appended to the node.

The metrics associated with a specific node are aggregated 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 kth cost, c_{jk}, associated with the device d_k in DA_k. The weights are useful in situations where the administrator of a SemIoTic-enabled smart space decides to impose constraints on user requests. For instance, that preserving people privacy is a must or that processing time is the most important criteria. By default, we consider that the value for the weights are assigned uniformly. The administrator can modify the values of the weights associated to each cost (as defined in the ontology) through the Portal GUI interface used to configured their SemIoTic instance (see Figure 4).

7 ACTION EXECUTION

In this section, we present the selection and execution of a plan from the possible plans resulting from the translation of a user action. Then, we detail the execution of device actions using device wrappers and virtual sensors to access/control IoT devices.

7.1 Plan Selection and Execution

The translation of a user action UA can result in several feasible execution plans. If the goal is to maximize the chances of carrying out the action (as devices might fail or produce noisy results), then 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 a 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 $\mathcal{T}_{UA_{rk}}$. Each node $\mathcal{N}_m \in \mathcal{T}_{UA_k}$ is handled as follows depending on its type. If $m = AC$, then the appropriate wrapper is notified for actuating a device. Then, if $m = VSR$, then the engine sets up the required data inputs based on its children nodes. This is performed by creating

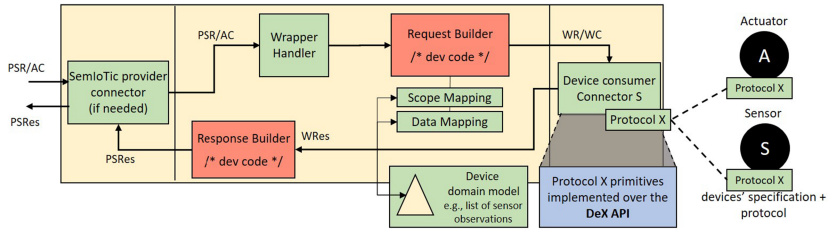


Fig. 13. Device wrapper design.

consumers that subscribe to receive data from virtual/physical sensors. 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, then 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.

7.2 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 [9, 24]. 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 (e.g., topic-based) or stream identifier. These may differ from device to device even if they observe the same type of data (e.g., temperature).

The design of SemIoTic wrappers (see Figure 13) 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/scope mapping* and (ii) *protocol mediation* between SemIoTic and the IoT devices.

A device action $DA = \langle d_i, a_j, \varepsilon_k \rangle$, a leaf node of the $\mathcal{T}_{UA}$ tree structure, is a PSR or a AC that can be handled by a *device wrapper*. As shown in Figure 13, 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 4–8), 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 5) that performs data and scope mapping for generating the wrapper request (WR_i), which is compatible with the corresponding IoT sensor.

To enable developers request data from any IoT device employing any IoT protocol, they have to only specify the device's protocol name when instantiating the *Wrapper Handler* (line 2). Then, we leverage the *Data eXchange* (DeX) API [9], which implements `post` and `get` DeX primitives for sending/receiving messages using existing IoT protocols such as CoAP, MQTT, XMPP, and so on. 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* 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 Reference [9]. 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, then 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 7).

```
1 class WHandler extends Handler {
2   public WHandler(String prot) { this.protocol = prot; /* device protocol name - e.g., CoAP */ }
3       @Override /* code for handling requests */
4       public void handleRequest(PSRequest psr) {
5               RequestBuilder rb = new RequestBuilder(psr); DevConsConnector dc = new DevConsConnector(this.getProtocol());
6               String getMsg = dc.getDexPrim().postExGet(rb.getDest(),rb.getScope(),rb.getPMsg());
7               ResponseBuilder resb = new ResponseBuilder(psr,getMsg);
8               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. 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), then 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. If an IoT protocol does not support streaming interactions (e.g., HTTP), then 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`.

7.3 Virtual Sensor Design

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). 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

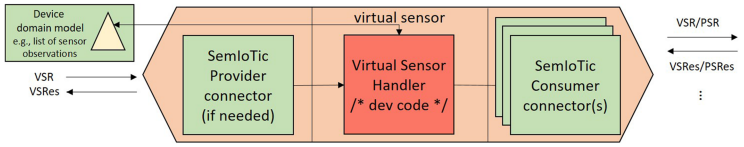


Fig. 14. Virtual sensor design.

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 Figure 14, 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 4—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 5). 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 7 and 8): the consumer requests data and is blocked until the response is given to be stored in the overall responses list; (ii) *asynchronous request* (lines 9–11): 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 12–14): 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 17 and 18).

```
1  class VSHandler extends Handler {
2    public void setConsumer(SemConsConnector consumer, String plan) { consumersList.add(consumer,plan) }
3          @Override /* code for handling single-response requests */
4          public void handleRequest(VSRequest vsr) {
5        ArrayList<Response> consRespList = new ArrayList<Response>();
6        for (SemConsConnector consumer: ConsumersList) {
7            Response consResp = consumer.syncRequest();  /* if sync request to consumer selected */
8            consRespList.add(consResp); /* consumer response list to be processed */
9            consumer.asyncRequest(new AsyncRequestCallback () {  /* if async request to consumer selected */
10           @Override
11           public void onMessage (Response consResp) { consRespList.add(consResp); }}) ;
12           consumer.streamRequest(new StreamRequestCallback (freq,dur) {  /* if streaming request to consumer selected */
13           @Override
14           public void onMessage (Response consResp) { consRespList.add(consResp); }}) ; }
15       /* code to process the incoming responses in consRespList */
16       /* .... */
17       /* provides the final response */
18       vsr.respond(new VSResponse); }
19         /* .... */
```

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.

8 EXPERIMENTS

In this section, we present the experiments performed to validate our approach. First, we describe the experimental setup. Then, we evaluate the handling of user actions, efficiency of user action execution, and scalability. Finally, we present a discussion based on the results.

8.1 Experimental Setup

The SemIoTic ecosystem have been deployed at the **University of California, Irvine (UCI)**. We acted as "space administrators" and installed and configured SemIoTic as follows. First, we selected the SemIoTic core system (see Figure 3) and an application. This application (an extension of the occupancy application shown in Figure 10) showcases an exploratory discovery of the space by enabling users to define actions to find the spaces defined in the model and their properties. In particular, it guides the user to define the SAL parameters (through HTML forms) 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 Figure 10(b)).

The app's dependencies include: (i) The "building" domain model and (ii) Software components that provide occupancy observations and control temperature. We selected the "wifi2occ" virtual sensor, which converts WiFi connectivity data to occupancy data, and a wrapper to connect to SkySpark, a software that handles the HVAC data in our buildings. Additionally, we selected a WiFi wrapper component that converts WiFi data to SemIoTic's format. In the current deployment, we have real-time access to WiFi data produced from 484 WiFi APs located in 15 buildings. We used Kubernetes to deploy the system on a server in the Donald Bren Hall building at UCI. Then, using the Portal web interface (see Figure 4), we defined the geographical space (buildings, floors, areas, and rooms) of UCI (see Figure 7), assigned the real WiFi AP to the designed areas, and added accounts for space occupants (using an invitation link for registering to SemIoTic).

We deployed another instantation of SemIoTic to simulate a smart home to show how space administrators can deploy IoT applications across different spaces. The installation process was similar to the previous. The only difference was the selection of a different set of wrappers and virtual sensors. In particular, we deployed a real Raspberry Pi with a camera sensor connected and hence downloaded a wrapper to connect to it. Then, we downloaded a virtual sensor to extract faces from pictures and another one that recognize faces and associates the person with a location. We simulate the thermostat sensor with a software component and develop a sample wrapper to communicate with it to retrieve temperature readings and control the temperature.

8.2 Handling Runtime user Actions

We evaluate the execution of the user requests posed by the application in both deployments. We take the role of two space occupants (one at UCI and one at the smart home), and use the application to control the temperature of two spaces, "Room 111" and "Living Room" in the SemIoTic-UCI

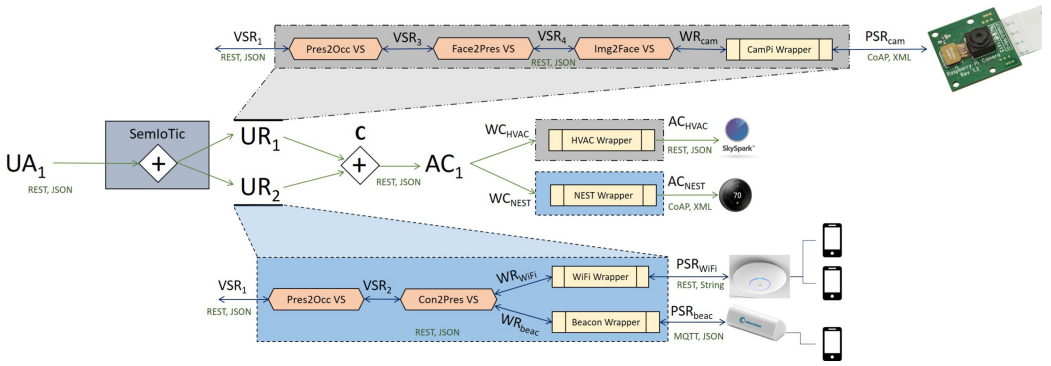


Fig. 15. Plans generated to handle a user action.

and SemIoTic-Home instances, respectively. The application poses the following user actions to SemIoTic-UCI to control the temperature of the room and to retrieve occupancy and temperature readings of the room (similar requests are generated for SemIoTic-Home):

⟨`Room111, ControlTemperature, OccupancyProp>0.5xCapacityProp`⟩

⟨`Room111, OccupancyProp/TemperatureProp,` ∅⟩

Both instances receive and handle the request as explained in Section 6. First, SemIoTic detects, for the action to control the temperature, the need to retrieve the room's occupancy and capacity properties and then actuate the property to control the temperature if the condition is met. The virtual sensor defined in both domain models to retrieve occupancy observations (which is the value that the occupancy property requires) is included in the plan. 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-UCI detects that there are no cameras or bluetooth beacons covering room 111. Thus, the final plan selected (see UR_2 in Figure 15) involves a request to the virtual sensor that generates occupancy data from presence data (Pres2Occ VS, 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 connectivity observations (Con2Pres VS, VSR_2), followed by a request to the WiFi AP covering the room. 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 UR_1 in Figure 15) that involves a request to the virtual sensor producing presence based on recognizing faces (Face2Pres VS, VSR_3), followed by a request to the virtual sensor that extracts faces from images (Img2Face VS, VSR_4), followed by a request to the camera (using a wrapper) 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 Figure 16) by using their definitions in the domain model. Underneath, SemIoTic-UCI retrieves data from the HVAC system and WiFi APs whereas SemIoTic-Home uses the simulated thermostat and the Raspberry Pi with a videocamera. Notice in Figure 16 how, in the case of the building, the room starts getting full at the start of the meeting at 9 a.m., which increases 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.

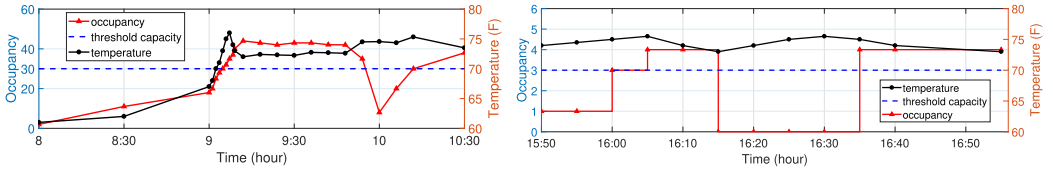


Fig. 16. Graphs displayed by the application using SemIoTic.

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.

8.3 Evaluating the Efficiency of user Action Execution

We evaluate the efficiency of user action execution involving plans with different level of complexity. Since we would like to explore the performance with increasing numbers of sensors/virtual sensors, we perform this experiment using simulated devices and virtual sensors. Virtual sensors are implemented in Java and their endpoints using the Restlet framework[14] for exchanging data in REST and JSON format. To simulate data processing in virtual sensors, their code performs an operation for a time extracted from the range 10–40 ms. Wrappers are implemented in Java using Restlet and the Eclipse Californium CoAP framework to exchange data with IoT devices in JSON and XML formats using REST or CoAP. Wrappers exchange data with SemIoTic using the common format and protocol (REST and JSON). Finally, IoT devices, which in this experiment are replaced by other software components instead of real devices, are implemented using the data exchange protocol (CoAP, REST) and data format (JSON, XML) of the corresponding connected wrapper.

To represent a realistic deployment between IoT devices and wrappers (i.e., realistic propagation and transmission delays), we use Mininet.[15] In particular, we create a virtual network in our machine and we deploy the generated application code of IoT devices and wrappers to different Mininet virtual hosts. The same machine hosts the SemIoTic instance and virtual sensors. We utilize a modest machine, an i5-8250U CPU (8 GB RAM) with an Ubuntu 16.04 OS, to test the performance of the system in deployments such as a smart home where higher-end servers might not be available.

Next, we generate plans with complexity levels 1 to 5. We equate complexity level to the number of levels in the plan to be processed. Finally, we simulate user actions that result in the above plans posed by multiple users in parallel. To this end, we use LOCUST,[16] an open source load testing tool, which allows to define multiple users, along with their behaviour defined as a Python script. Upon the arrival of responses on LOCUST, we measure the round-trip response time of each action, and then we estimate the average values for all user actions.

[14] https://restlet.talend.com.
[15] http://mininet.org.
[16] https://locust.io.

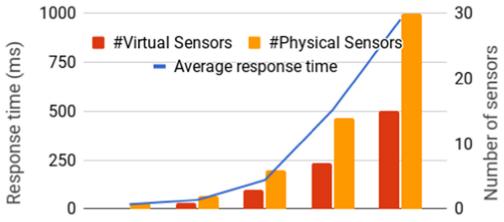


Fig. 17. Response time values when handling 10 requests from different execution plans.

Table 1. Response Time Values when Handling Requests from 100 Simultaneous Users

Number of users	Requests/s	Response time (ms)	
		Level-2	Level-4
10	1	49	552
25	3	55	782
50	9	64	1,024
75	12	73	—
100	20	92	—

We define through LOCUST 10 users each sending one user action in parallel and measure the average response time across all actions (Figure 17 presents the measured round-trip response times). For a *level-3* complexity plan, which involves three virtual sensors and six physical sensors, the resulting average response time for executing 10 requests is 148.7 ms. When we increase the complexity of the plan to *level-5*, which involves 30 physical sensors and 15 virtual sensors, the maximum average response time is 969.9 ms. Note that the execution of the plans is synchronous, meaning that every component (virtual sensor, wrapper) forwards the request to the next one and then it blocks its processing until it receives the response. Thus, the overhead introduced by our approach to generate and execute the plan by requesting data from IoT devices through wrappers, and transferring that information to the virtual sensors is small. Our approach handles efficiently the execution of plans in reception of data from 30 IoT devices (physical sensors) that are processing in 15 virtual sensors.

Next, we evaluate the performance with increasing number of simultaneous user actions. Our SEMIC metaontology is defined to generate plans of two and four levels of complexities for this experiment. Then, we defined through LOCUST 100 simultaneous users, each sending 1–20 requests per second according to a Poisson process. We defined the hatch rate of users, which is one in this scenario, as the rate per second in which users are spawned until they reach 100.

Table 1 presents the average response time when handling increasing number of simultaneous user requests. Based on Table 1, our approach, with the experimental setup explained before, can handle efficiently 2,000 user requests per second within 92 ms when executing *level-2* complexity plans (involving one virtual sensor and 2 IoT devices). When executing *level-4* complexity plans (involving seven virtual sensors and 14 IoT devices), our approach can handle up to 450 requests per second with an average response time of 1,024 ms. This is because the buffers of virtual sensors and wrappers fill up quicker with requests due to the synchronous nature of interactions. Hence, our machine runs out of memory while the CPU utilization reaches 100%.

8.4 Measuring Development Effort

In the following experiment, we take the role of content developers, and 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

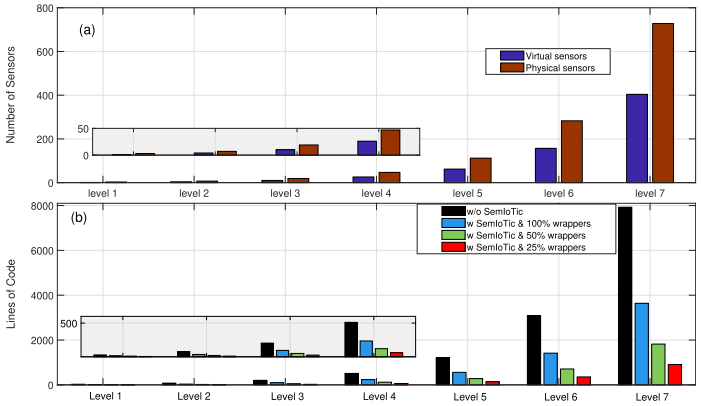


Fig. 18. Development effort with (w) and without (w/o) SemIoTic.

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. 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 Reference [9] the authors setup an MQTT subscriber by using eight LoC without considering the data mapping task). We assume the best case scenario when developing plans without SemIoTic by considering five LoC for setting up a consumer and two additional LoC to perform data mapping for a simple message type. Thus, we consider $n_{loc}^{in} = 7$.

Figure 18(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. Figure 18(b) shows the number of LoC required to develop an application, with and without SemIoTic, vs. the complexity of the scenario. With SemIoTic, we vary the 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 four wrappers in total for cameras (around 40), HVAC sensors (around 7K), WiFi APs (around 60), and bluetooth beacons (around 200). Figure 18(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 Figure 18(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 that 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, and so on.

8.5 Discussion

SemIoTic aims to facilitate the development of IoT applications in different smart spaces. While its translation of low-level to high-level information adds latency, this has to be performed at some point, since IoT data is too low-level for users to consume. Today, this is usually done at the application level (which is sometimes pushed to the cloud). This means that applications have to receive low-level data, which can incur in additional network latency costs, and then translate it. In SemIoTic, this is done within the system to avoid that additional network transmission costs; reduce the complexity and interactions between IoT applications and heterogeneous devices; and facilitate IoT developers that might not be familiar with low-level IoT data.

The experiments have shown the feasibility of our approach in small and medium scale IoT deployments. Scalability depends on several factors such as: (i) the number of data requests; (ii) the complexity-level of the generated plans—i.e., the number of VSs involved in each plan; (iii) the time VSs take to process raw data; and (iv) the type of each user action—i.e., synchronous, asynchronous, and streaming. Based on the experiment in Section 8.3, SemIoTic handles up to 450 requests per second from 50 simultaneous space occupants with an average response time of 1,024 ms. Every request is served using a plan that involves on average 7 virtual sensors and 14 IoT devices and every virtual sensor processes data for a time extracted from a range of 10–40 ms.

Scenarios that require a much larger scale (e.g., a SemIoTic managing the sensors in a smart city) would require extensions to support real-time operations. In particular, translating each user action in real-time might not be feasible in such scenarios but it also might not be required. Strategies that cache both previous translation plans, as well as previous results obtained when processing different user actions, can help in those situations. However, this presents additional challenges such as maintaining the cache, updating it when the underlying sensor infrastructure changes, and so on.

Additionally, we have shown that SemIoTic decreases application development effort measured in terms of LoC written. It would be interesting to understand and measure also how the reduction in the LoC required to develop a smart application translates into decreasing the overall application development time. However, designing and performing such a study in a meaningful manner is not trivial and presents several challenges that made it infeasible for this article. This include, among others, gathering a statistically relevant group of diverse engineers with knowledge/experience on the development of IoT applications and IoT communication/data protocols.

9 CONCLUSIONS

We have presented the SemIoTic ecosystem, a holistic approach to facilitate application development, space management, service provision, and reusability of IoT content across different smart spaces from diverse domains. The ecosystem includes three main components: (1) a Marketplace that serves as a repository of content (i.e., applications, domain models, wrappers, and virtual sensors) for a smart space; (2) a Hub that enables users to discover smart spaces around them; and (3) the network of SemIoTic-enabled smart spaces that offer applications to users. SemIoTic facilitates the development of space and sensor agnostic applications by offering interfaces to express requirements (i.e., requests for data, commands, and policies) based on high-level semantically meaningful concepts. Then, SemIoTic translates those into actions on the underlying IoT device infrastructure deployed in each smart space. Finally, it communicates with the devices to retrieve data or actuate them. We have shown the feasibility of our approach through a reference

implementation and the deployment at the University of California Irvine campus. In the future, we plan to continue deploying the system in other smart spaces and domains, such as nursing homes, and promoting the development of content for it. We also plan to deal with other challenges related to the definition and management of consistent user-defined policies.

ACKNOWLEDGMENTS

We thank Andrei Homentcovschi, DongCheol Jwa, HyeEun Song, Hyunji Lee, Jorge Carlos, Lasse Nordahl, Leo Peng, MyoungHoon Han, Nada Lahjouji, Nathan Ma, Rayan Al Atab, Tristan Jogminas, Stuart McClintock, Vikram Miryala, William Sun, and Yiming Wang for their help with the implementation.

REFERENCES

[1] SemIoTic. [2021]. Retrieved from https://tippers.ics.uci.edu/semiotic.
[2] Project Haystack. [2021]. Retrieved from https://project-haystack.org.
[3] Mussab Alaa, A. A. Zaidan, B. B. Zaidan, Mohammed Talal, and Miss Laiha Mat Kiah. 2017. A review of smart home applications based on internet of things. *J. Netw. Comput. Appl.* 97 (2017).
[4] Pandarasamy Arjunan, Nipun Batra, Haksoo Choi, Amarjeet Singh, Pushpendra Singh, and Mani B. Srivastava. 2012. Sensoract: A privacy and security aware federated middleware for building management. In *Proceedings of the 4th ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*.
[5] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The internet of things: A survey. *Comput. Netw.* 54, 15 (2010).
[6] Bharathan Balaji, Arka Aloke Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, Mario Berges, David E. Culler, Rajesh E. Gupta, Mikkel Baun Kjærgaard, Mani B. Srivastava, and Kamin Whitehouse. 2016. Brick: Towards a unified metadata schema for buildings. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys'16)*.
[7] Debasis Bandyopadhyay and Jaydip Sen. 2011. Internet of things: Applications and challenges in technology and standardization. *Wireless Personal Commun.* 58, 1 (2011).
[8] Robert Battle and Dave Kolas. 2011. Geosparql: Enabling a geospatial semantic web. *Semantic Web J.* 3, 4 (2011), 355–370.
[9] Georgios Bouloukakis, Nikolaos Georgantas, Patient Ntumba, and Valérie Issarny. 2019. Automated synthesis of mediators for middleware-layer protocol interoperability in the IoT. *Future Gener. Comput. Syst.* 101 (2019).
[10] Joshua Cao, Jesse Chong, Marissa Lafreniere, Owen Yang, Primal Pappachan, Sharad Mehrotra, and Nalini Venkatasubramanian. 2020. The ZotBins solution to waste management using Internet of Things: Poster abstract. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems (SenSys'20)*.
[11] Angelo Paolo Castellani, Thomas Fossati, and Salvatore Loreto. 2012. HTTP-CoAP cross protocol proxy: An implementation viewpoint. In *Proceedings of the 9th IEEE International Conference on Mobile Ad-Hoc and Sensor Systems (MASS'12)*.
[12] David A. Chappell. 2004. *Enterprise Service Bus*. O'Reilly Media.
[13] Bo Cheng, Da Zhu, Shuai Zhao, and Junliang Chen. 2016. Situation-aware IoT service coordination using the event-driven SOA paradigm. *IEEE Trans. Netw. Service Manage.* 13, 2 (2016).
[14] Matteo Collina, Giovanni Emanuele Corazza, and Alessandro Vanelli-Coralli. 2012. Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST. In *Proceedings of the 23rd IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC'12)*.
[15] Michael Compton, Payam Barnaghi, Luis Bermudez, Raúl García-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, Vincent Huang, Krzysztof Janowicz, W. David Kelsey, Danh Le Phuoc, Laurent Lefort, Myriam Leggieri, Holger Neuhaus, Andriy Nikolov, Kevin Page, Alexandre Passant, Amit Sheth, and Kerry Taylor. 2012. The SSN ontology of the W3C semantic sensor network incubator group. *J. Web Semant.* 17 (2012).
[16] Michael Compton, Payam M. Barnaghi, Luis Bermudez, Raul Garcia-Castro, Óscar Corcho, Simon J. D. Cox, John Graybeal, Manfred Hauswirth, Cory A. Henson, Arthur Herzog, Vincent A. Huang, Krzysztof Janowicz, W. David Kelsey, Danh Le Phuoc, Laurent Lefort, Myriam Leggieri, Holger Neuhaus, Andriy Nikolov, Kevin R. Page, Alexandre Passant, Amit P. Sheth, and Kerry Taylor. 2012. The SSN ontology of the W3C semantic sensor network incubator group. *J. Web Semant.* 17 (2012).
[17] Bruno Costa, Paulo F. Pires, and Flávia Coimbra Delicato. 2016. Modeling IoT applications with SysML4IoT. In *Proceedings of the 42nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA'16)*.
[18] Laura Daniele, Frank den Hartog, and Jasper Roes. 2015. Created in close interaction with the industry: The Smart Appliances REFerence (SAREF) ontology. In *Proceedings of the International Workshop Formal Ontologies Meet Industries*.

[19] Hasan Derhamy, Jens Eliasson, and Jerker Delsing. 2017. IoT interoperability—On-demand and low-latency transparent multiprotocol translator. *IEEE Internet Things J.* 4, 5 (2017).

[20] Pratikkumar Desai, Amit P. Sheth, and Pramod Anantharam. 2015. Semantic gateway as a service architecture for IoT interoperability. In *Proceedings of the IEEE International Conference on Mobile Services (MS'15)*, Onur Altintas and Jia Zhang (Eds.).

[21] Gabe Fierro, Marco Pritoni, Moustafa AbdelBaky, Paul Raftery, Therese Peffer, Greg Thomson, and David E. Culler. 2018. Mortar: An open testbed for portable building analytics. In *Proceedings of the 5th Conference on Systems for Built Environments (BuildSys'18)*, Rajesh Gupta, Polly Huang, and Marta Gonzalez (Eds.).

[22] Yi Gao, Jiadong Zhang, Gaoyang Guan, and Wei Dong. 2020. LinkLab: A scalable and heterogeneous testbed for remotely developing and experimenting IoT applications. In *Proceedings of the IEEE/ACM 5th International Conference on Internet-of-Things Design and Implementation (IoTDI'20)*.

[23] Gartner. 2019. 5.8 Billion Enterprise and Automotive IoT Endpoints Will Be in Use in 2020. Retrieved from https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io.

[24] Nikolaos Georgantas, Georgios Bouloukakis, Sandrine Beauche, and Valérie Issarny. 2013. Service-oriented distributed applications in the future internet: The case for interaction paradigm interoperability. In *Proceedings of the 2nd European Conference on Service-Oriented and Cloud Computing (ESOCC'13)*, Kung-Kiu Lau, Winfried Lamersdorf, and Ernesto Pimentel (Eds.), Vol. 8135.

[25] Thomas R. Gruber. 1993. A translation approach to portable ontology specifications. *Knowl. Acquisit.* 5, 2 (1993).

[26] Armin Haller, Krzysztof Janowicz, Simon J. D. Cox, Maxime Lefrançois, Kerry Taylor, Danh Le Phuoc, Joshua Lieberman, Raúl García-Castro, Rob Atkinson, and Claus Stadler. 2019. The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation. *Semant. Web* 10, 1 (2019).

[27] Jan Janak and Henning Schulzrinne. 2016. Framework for rapid prototyping of distributed IoT applications powered by WebRTC. In *Proceedings of the Principles, Systems and Applications of IP Telecommunications (IPTComm'16)*.

[28] Minsung Jang, Hyunjong Lee, Karsten Schwan, and Ketan Bhardwaj. 2016. SOUL: An edge-cloud system for mobile applications in a sensor-rich world. In *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC'16)*.

[29] Krzysztof Janowicz, Armin Haller, Simon J. D. Cox, Danh Le Phuoc, and Maxime Lefrançois. 2019. SOSA: A lightweight ontology for sensors, observations, samples, and actuators. *J. Web Semant.* 56 (2019).

[30] Tomasz Kalbarczyk and Christine Julien. 2018. Omni: An application framework for seamless device-to-device interaction in the wild. In *Proceedings of the 19th International Middleware Conference*.

[31] Konstantinos Kotis and Artem Katasonov. 2013. Semantic interoperability on the internet of things: The semantic smart gateway framework. *Int. J. Distrib. Syst. Technol.* 4, 3 (2013).

[32] Pratik Lade, Yash Upadhyay, Karthik Dantu, and Steven Y. Ko. 2016. Developing adaptive quantified-self applications using dynasense. In *Proceedings of the IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI'16)*.

[33] Yiming Lin, Daokun Jiang, Roberto Yus, Georgios Bouloukakis, Andrew Chio, Sharad Mehrotra, and Nalini Venkatasubramanian. 2020. LOCATER: Cleaning WiFi connectivity datasets for semantic localization. *Proc. VLDB Endow.* 14, 3 (2020).

[34] Marcin Luckner, Maciej Grzenda, Robert Kunicki, and Jaroslaw Legierski. 2020. IoT architecture for urban data-centric services and applications. *ACM Trans. Internet Techn.* 20, 3 (2020).

[35] Behailu Negash, Amir M. Rahmani, Tomi Westerlund, Pasi Liljeberg, and Hannu Tenhunen. 2016. LISA 2.0: Lightweight internet of things service bus architecture using node centric networking. *J. Ambient Intell. Human. Comput.* 7, 3 (2016).

[36] Joseph Noor, Hsiao-Yun Tseng, Luis Garcia, and Mani Srivastava. 2019. Ddflow: Visualized declarative programming for heterogeneous iot networks. In *Proceedings of the International Conference on Internet of Things Design and Implementation*.

[37] Ioannis V. Papaioannou, Dimitrios T. Tsesmetzis, Ioanna Roussaki, and Miltiades E. Anagnostou. 2006. A QoS ontology language for web-services. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA'06)*.

[38] Primal Pappachan, Martin Degeling, Roberto Yus, Anupam Das, Sruti Bhagavatula, William Melicher, Pardis Emami Naeini, Shikun Zhang, Lujo Bauer, Alfred Kobsa, Sharad Mehrotra, Norman M. Sadeh, and Nalini Venkatasubramanian. 2017. Towards privacy-aware smart buildings: Capturing, communicating, and enforcing privacy policies and preferences. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS'17)*.

[39] Primal Pappachan, Roberto Yus, Sharad Mehrotra, and Johann-Christoph Freytag. 2020. Sieve: A middleware approach to scalable access control for database management systems. *Proc. VLDB Endow.* 13, 11 (2020).

[40] Ferry Pramudianto, Carlos Alberto Kamienski, Eduardo Souto, Fabrizio F. Borelli, Lucas L. Gomes, Djamel Sadok, and Matthias Jarke. 2014. IoT Link: An internet of things prototyping toolkit. In *Proceedings of the IEEE 11th International Conference on Ubiquitous Intelligence and Computing*.

[41] Eric Prud'hommeaux and Andy Seaborne. 2008. SPARQL Query Language for RDF. W3C Recommendation. Retrieved from http://www.w3.org/TR/rdf-sparql-query/.
[42] Felix Maximilian Roth, Christian Becker, Germán Vega, and Philippe Lalanda. 2018. XWARE—A customizable interoperability framework for pervasive computing systems. *Pervas. Mob. Comput.* 47 (2018).
[43] Eun-Jeong Shin, Roberto Yus, Sharad Mehrotra, and Nalini Venkatasubramanian. 2017. Exploring fairness in participatory thermal comfort control in smart buildings. In *Proceedings of the 4th ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys'17)*.
[44] Sabrina Sicari, Alessandra Rizzardi, Luigi Alfredo Grieco, and Alberto Coen-Porisini. 2015. Security, privacy and trust in internet of things: The road ahead. *Comput. Netw.* 76 (2015).
[45] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. 2007. Pellet: A practical OWL-DL reasoner. *J. Web Semant.* 5, 2 (2007).
[46] Tania Tudorache, Csongor Nyulas, Natalya Fridman Noy, and Mark A. Musen. 2013. WebProtégé: A collaborative ontology editor and knowledge acquisition tool for the Web. *Semant. Web* 4, 1 (2013).
[47] Itorobong S. Udoh and Gerald Kotonya. 2018. Developing IoT applications: Challenges and frameworks. *IET Cyper-Phys. Syst.: Theory Appl.* 3, 2 (2018).
[48] Roberto Yus, Georgios Bouloukakis, Sharad Mehrotra, and Nalini Venkatasubramanian. 2019. Abstracting interactions with IoT devices towards a semantic vision of smart spaces. In *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (BuildSys'19)*.

Received September 2021; revised January 2022; accepted March 2022