Open Access

Open Journal of Internet of Things (OJIOT)
Volume 4, Issue 1, 2018

www.ronpub.com/ojiot
ISSN 2364-7108

Service-Relationship Programming Framework for the Social IoT

Ahmed E. Khaled[A], Wyatt Lindquist[B], Abdelsalam (Sumi) Helal[B]

[A] Computer and Information Science and Engineering Dept., University of Florida, Gainesville, Fl 32611 USA, aeeldin@ufl.edu
[B] School of Computing and Communication, Lancaster University, Lancaster, LA1 4WA, UK, {w.lindquist, s.helal}@lancaster.ac.uk

ABSTRACT

We argue that for a true realization of innovative programming opportunities for smart spaces, the developers should be equipped with informative tools that assist them in building domain-related applications. Such tools should utilize the services offered by the space's smart things and consider the different relationships that may tie these services opportunistically to build applications. In this paper, we utilize our Inter-thing relationships programming framework to present a distributed programming ecosystem. The framework broadens the restricted set of thing-level relationships of the evolving social IoT paradigm with a set of service-level relationships. Such relationships provide guidance into how services belonging to different things can be combined to build meaningful applications. We also present a uniform way of describing the thing services and the service-level relationships along with new capabilities for the things to dynamically generate their own services, formulate the corresponding programmable interfaces (APIs) and create an ad-hoc network of socially related smart things at runtime. We then present the semantic rules that guide the establishment of IoT applications and finally demonstrate the features of the framework through a proof-of-concept application.

KEYWORDS

Social IoT, Atlas thing architecture, inter-thing relationships, IoT programming model, service

1 INTRODUCTION

Current advancements in the Internet of Things (IoT) have evolved from the ubiquitous presence of the smart and cyber things, to the actual establishment of applications that realize the true capabilities of smart spaces [9][10]. Such inter-connected things (from the state-of-art of the current IoT infrastructures, platforms, sensing technologies and communication protocols) have triggered endless innovative scenarios that guide developers to program the surrounding smart spaces [24]. However, for a realization of the smart spaces' resources and capabilities to establish domain-related applications, the development environment should not only be based on the services offered by the things but also on the relationships that describe how such services can unite to build meaningful applications [16]. These relationships create a new paradigm named social IoT [1][11], as a social network of smart things

This paper is accepted at the *International Workshop on Very Large Internet of Things (VLIoT 2018)* in conjunction with the VLDB 2018 Conference in Rio de Janeiro, Brazil. The proceedings of VLIoT@VLDB 2018 are published in the Open Journal of Internet of Things (OJIOT) as special issue.

that inform the developer how things' services can build domain-related IoT applications.

The recently proposed ideas on social IoT [1][11] logically link the things according to their identification attributes (e.g., things from same vendor, things collocated in the same smart space). Such thing-level relationships don't reflect how the different services offered by these things are related. However, the exploitation of the service-level relationships in the context of social IoT adds an effective programming perspective to the evolving paradigm of Social IoT [16].

On the other hand, the current programming frameworks for the IoT [24][25] consider a restricted set of relationships which we think are not sufficient to build a wide range of applications. For instance, If This Then That (IFTTT) [15][26] only offers applications where one service controls the operation of another service (e.g., initiate an emergency call if smoke is detected). However, the exploitation of more service-level relationships that logically and functionally tie the different services for new engagement opportunities will highly enrich the space of innovative applications [16]. These frameworks also ignore the ad-hoc nature of the smart things and require additional effort for the manual configuration and registration of the things and services to powerful platforms (e.g., cloud). However, enabling both thing-to-thing and thing-to-cloud interactions along with the seamless integration of the things in the ecosystem empowers the properties of distributed programming environments [9][10] that reside on both the things and cloud (e.g., no single point of failure, seamless integration and management).

In [16], we presented an overview of the inter-thing relationships detailed in this paper. We demonstrated how such relationships can be utilized within a programming framework based on our Atlas thing architecture and its IoT Device Description Language (IoT-DDL) [17][18] to build a distributed programming ecosystem for the social IoT. In this paper, we extend our inter-thing relationships programming framework presented in [16] with a detailed formalization along with algorithmic implementations of each primitive or operator in the framework. Effectively, the framework broadens the social IoT thing-level relationships with a set of concrete service-level relationships that can empower developers to establish a much wider class of IoT applications.

The framework introduces service (abstraction of the function offered by a thing), relationship (abstraction of how different services are linked together) and recipe (abstraction of how different services and relationships build up an app) as three primitives. The relationships defined in the framework can be utilized by vendors of the thing, utilized by developers while building apps, and dynamically inferred from the knowledge exchanged between the things as new programming opportunities. The framework also defines Filter, Match, and Evaluate as three operators that define how the primitives are wired. The thing vendor, Atlas thing (a thing with the Atlas thing architecture) and developer are the main poles of the framework: 1) the vendor describes a thing's services and relationships with other things; 2) the thing generates services and exchanges knowledge with the other things; and 3) the developer utilizes our Atlas IDE to sense the smart space, infer new programming opportunities, and communicate back with the things for services calls.

We present the framework in detail in this paper and show how it facilitates describing an IoT application through a set of semantic rules. The semantic rules evaluate the correctness of the established application by the developer and guide the execution of the application. We also present the capability of the thing through the Atlas architecture to dynamically: 1) build run-time programmable objects for the offered services and the relationships that link them to other things; 2) generate actual services from the description provided by the vendor; and 3) formulate and generate the appropriate programming interfaces (APIs) to access the offered services by the thing.

Throughout this paper, we present a detailed proof-of-concept scenario for engaging the proposed programming framework with the Atlas thing architecture and the IoT-DDL. The presented application is a home automation scenario triggered when the smart door locker senses that no one is present at home. The scenario utilizes three things in the smart space: 1) a smart lock that locks the home door if no one is home; 2) a thermostat that adjusts room temperature; and 3) motorized window blinds that can be tilted up and down. The presented application illustrates how a service is described (as will be detailed in Section 4.1.a), how a relationship is described (as will be detailed in Section 4.1.b), how the Atlas thing dynamically generates the service (as detailed in Section 5) and how the framework primitives are wired to build such meaningful scenario (as will be detailed in Section 6).

The paper is organized as follows. Section 2 highlights related work in both social IoT and programming models for IoT. Section 3 presents an overall view on the Atlas thing architecture and the thing IoT-DDL with focus on the layers that implement the framework. Section 4 presents the details of the Inter-thing relationship programming framework and the semantic rules followed by the details of the actual generation of services at the runtime in Section 5.

Building an Atlas IoT app is described in Section 6. Finally, a discussion and future work along with the conclusion are presented in Section 7 and Section 8 respectively.

2 Related Work

Atzori et al. [1] proposed a paradigm of a social network of smart objects named Social Internet of Things (SIoT) to mimic human behavior. The authors analyzed the types of social relationships between things to be: parental (things built by the same vendor), co-location and co-work (things reside in the same place or cooperate to provide applications), and owner (things owned by the same user). The authors in [8] also presented an architecture to address network navigability along with service discovery and composition. The architecture is made up of server and objects (the physical devices) as the network elements. The server holds the relationship management module where the selection and setting of the relationships is based on human control settings along with appropriate interfaces to objects, humans and third-party services. The object side holds an abstraction layer for the device and the social management module for the communication between the device and the server.

Holmquist et al. [11] presented a context proximity procedure that creates friendship between embedded devices named Smart-Its. Smart-Its are wireless tiny devices, equipped with sensing and processing capabilities in addition to onboard accelerometers. The movement data of the device is captured and broadcast to other smart-its in range to be compared to their movement patterns. If similar patterns are detected, the former smart-it is accepted as a friend and a connection is then established with the other smart-its. The author's main concern was on the qualitative and selective connections that can be established between such smart devices.

Turcu et al. [28] considered building an RFID-based social network of cognitive robots, for human-robot and robot-robot social interactions. The authors used Twitter as a social network to build online communities, where they created Twitter accounts for each robot. A robot's behavior is determined according to the RFID-tagged entities that come across its path; the robot then exhibits a predefined behavior (e.g. love, fear, repulsiveness). Such behavior is sent as a message on Twitter and the robot then waits for a reply to decide what to do next. Kranz et al. [20] introduced further steps in integrating IoT with social networks. The authors have also chosen Twitter as an online social network, and then created accounts for cognitive plant controllers. Such controllers are equipped with a Twitter-enabled sensing system that tweets the humidity information to the plant's Twitter account.

If This Then That (IFTTT) [15][26] is a web-based service that allows users to connect various Internet-based services (e.g., Facebook) by creating rules (called recipes). IFTTT allows two services to be manually combined using simple if-then statements to accomplish a task and utilizes the APIs offered by services' vendors (e.g., Twitter) to access the client's data. As an instance, IFTTT can be used to send files uploaded into Dropbox into Evernote, automatically. As mentioned earlier, IFTTT uses recipes to describe actions, where the users of the platform can search existing preconfigured recipes. The user then needs to give permission for the services to allow IFTTT access to the personal data associated with the accounts. Recently, IFTTT has been working on integrating these services with smart products (e.g., Belkin WeMo Home automation, Philips Hue LED light bulb) through utilizing the open APIs offered by the vendors and manufacturers of these devices.

Jaeseok Yun et al. [32] demonstrated a prototype service named TTEO (Things Talk to Each Other) that enables users to program IoT through a set of if-then rules. TTEO utilizes two platforms, the connectivity platform named Mobius that resides in an IoT server and the smart service server named &Cube. The server registers and collects data from the devices, and maintains virtual representations of them. The devices can be interoperated with each other through the Mobius platform. The server allows developers to customize and configure devices connected to the Mobius and enables the developer to build new services through a predefined set of if-then rules.

Stefan Nastic et al. [25] proposed an on-cloud platform named PatRICIA for high-level IoT programming. PatRICIA is based on Service-oriented architecture (SOA) design principles. The platform holds virtualizations of the connected devices, communication protocols and connectors, and a device manager and service discovery utility. The platform also contains the application development and deployment tools as well as the programming model. The programming model defines a set of constructs and operators for the development of applications through predefined domain-specific tasks defined by domain experts. The control task represents a sequence of actuating steps to control physical devices, while the monitor task represents processing and analysis of sensory data streams for meaningful information. Each task is represented via an Intent: a data structure to describe, configure and invoke the operation of the control or monitoring task, where the execution and processing reside on the cloud.

Chao Chen et al. [5] proposed an event-driven programming model named E-SODA, as an extension of the service-oriented device architecture (SODA). SODA focuses on the services provided by system, rather than the sensor data streams. The authors developed a reference implementation of SODA, which features the Atlas sensor platform and middleware proposed in [19]. The Atlas middleware enables service discovery and composition to create an app. The Atlas sensor platform automatically represents the devices as service bundles that implement a uniform service interface and abstract the physical details. E-SODA abstracts sensor data into events while an application follows a rule-oriented processing paradigm that is composed of a list of Event-Condition-Action (ECA) rules. An ECA rule listens to the occurrence of a predefined event derived over sensor data and responds by taking the corresponding action if the condition is satisfied. An E-SODA application is a collection of interrelated services together performing the function of rule evaluation. A rule object keeps references to three services of different types that represent the event, condition, and action components of this rule.

The Web of Things (WoT) framework by the World Wide Web Consortium (W3C) [30][31] is an active research field that explores access to and handling things' digital representations through a set of web services. These services are based on event-condition-action rules that involve these virtual representations as proxies for physical entities. Such objects are modeled in terms of metadata, events, and actions, where servers then provide an interface for instantiating and registering such proxies for the things along with their descriptions. A client script interacts with these proxies exported by the server, where applications can register callbacks for events. Darko et. al. [22] utilize Thing Description (TD) to describe the different things in the WoT, in terms of thing's metadata, how to access them, different events and the corresponding actions. The TD relies on the Resource Description Framework (RDF) [13] as an underlying data model that can be extended to involve domain specific information.

The inter-thing relationship programming framework proposed in [16] broadens the social IoT thing-level relationships proposed in [1][8][11] with service-level relationships that logically and functionally show how the things' services may tie to build applications. Such service-level relationships extend the limited and restricted set of relationships presented in [5][26][32][15] with a new set of concrete of relationships that can empower developers to establish a much wider class of IoT applications. The developer utilizes the framework to describe an application (in terms of the different primitives and operators), such application is governed by a set of semantic rules that evaluate the correctness and guide the execution. On the other hand, for a true distributed programming ecosystem, the thing (through the mounted Atlas architecture on the thing and the IoT-DDL uploaded to the thing) dynamically builds run-time programmable objects for the offered services and the relationships and generates services along with the appropriate APIs to them.

The focus of this paper is on: 1) the capability of the vendor to describe services and relationships to the thing through the uploaded IoT-DDL; 2) the capability of framework to connect the different primitives and operators to build an IoT application; and 3) the capability of the thing to generate services and formulate the appropriate APIs. We also presented Atlas IDE (an application development environment that implement the proposed programming framework) that enables the developer to discover announced knowledge about TS(s) and TR(s) from the things and infers the existence of new programming opportunities from the exchanged knowledge between the things. The discovered relationships reflect how the current services can be further related to each other and enrich the programmability of the space to the developer with new service engagement opportunities. However, for space constraints and to keep the focus of the paper, the details of the implementation of the Atlas IDE is outside the scope of this paper.

3 Atlas Architecture and IoT-DDL

As mentioned earlier, our inter-thing relationship programming framework is built upon the Atlas thing architecture project and the IoT Device Description Language (IoT-DDL) specification [17][18]. The IoT-DDL is a machine- and human-readable XML-based descriptive language that describes the identity of the thing, its inner entities, resources, and offered services, and the cloud-based accessories attached to it. The architecture, utilizing the IoT-DDL specifications, allows the thing to self-discover its own capabilities and engage through different thing-to-thing and thing-to-cloud interactions with other platforms and thing mates. The thing, through the architecture, handles the dynamic formulation of services, the generation of corresponding access interfaces (APIs), and the building of run-time programmable objects for the offered services and the relationships that link them to other things. In this section, we present a brief overview of the architecture and the IoT-DDL with focus on the main aspects that empower the proposed framework.

3.1 IoT-DDL

The IoT-DDL is a schema used to describe, through a set of attributes and parameters, the thing in a smart space in terms of the set of resources, inner entities, cloud-based attachments, and interactions that engage the thing with other things and cloud platforms [17]. Resources are the components that shape the OS services (e.g., network module, memory unit). Moreover, thing entities are the physical devices, software functions, and hybrid devices that can be attached to, built into, or embedded inside the thing, where each entity provides a set of services to the smart space. A cloud-based attachment is an expansion of the thing that provides further representations (e.g., thing virtualization) and functionalities (e.g., log server, database, or dashboard) that require heavyweight resources that may not be available on more constrained things. A thing engages with others through a set of information- and action-based interactions. Information-based interactions (referred to as tweets) enable a thing to announce its identity, capabilities, and services, while the Action-based ones include management commands and lifetime updates as well as the apps that target the thing's services.

IoT-DDL is based on Atlas DDL [4], which uses an XML-based schema to describe devices to facilitate their integration in a smart space. It has been used to develop the Atlas Cloud-Edge-Beneath (Atlas-CEB) architecture [12], which uses DLL to generate Java bundles representing the devices that can be deployed on an edge and/or cloud to connect back and interact with the devices the DDL describes. DDL is used to describe a single device (sensor, actuator, or hybrid) through the device's metadata, functions, and operations. Atlas IoT-DDL extends Atlas DDL to describe the different components of the thing as outlined above. The Atlas thing section in the IoT-DDL, as illustrated in Listing 1, provides a description metadata subsection about the thing (e.g., name, vendor, operating system, overall description, Atlas thing ID, smart space ID [18]), and a resources subsection (e.g., network module, memory properties).

The Atlas entity section in the IoT-DDL, on the other hand, provides information on the attached, built-in or connected hardware and software entities of the thing. In addition to descriptive metadata information, each entity section details the set of services it can offer. Each service is characterized by its functional properties, the required inputs, and the expected outputs (in terms of the data type, units, and expected range). The properties (functional description, inputs and outputs) of these services are utilized by the Atlas thing architecture (as will be detailed in Section 5) to generate services dynamically and formulate the appropriate interfaces (APIs), allowing things in the smart space to utilize the generated services.

```
<Atlas_Thing>
    <Descriptive_Metadata>
        <Owner>Mobile Computing Lab</Owner>
        <Name>Raspberry Pi 3</Name>
        <OS>Raspbian</OS >
          <ATID>AtlasThing128</ATID>
          <SSID>SmartSpace326012</SSID>
          ...
    </Descriptive_Metadata>
          ...
    <Resources>
        <Network_Properties>
            <Module>Wifi</Module>
            <UUID>Lab Network</UUID>
            <Protocol>REST</Protocol>
            <URL>192.168.1.54</URL>
            ...
        </Network_Properties>
        <Memory_Properties>
            ...
        </MemoryProperties>
    </Resources>
</Atlas_Thing>
```

Listing 1: An IoT-DDL snippet showing the Atlas thing section

3.2 Atlas Thing Architecture

The architecture consists of a set of new operating layers that we propose to provide novel capabilities a thing requires to engage and interact with other things and platforms in the smart space. An implementation of the architecture is to be flashed into the thing using the vendor's provided IDE or OS (e.g., C/C++ for Linux-based platforms such as Raspberry Pi, Java for Android smartphones, or IDE for Arduino).

The architecture, as illustrated in Figure 1, consists of three main layers: Atlas IoT platform, host interface, and IoT OS services. IoT OS services are the basic functionalities provided by the thing's operating engine to enable the thing to be part of the ecosystem (e.g., network module, memory units, I/O ports and physical interfaces, and its process manager).

The Atlas IoT platform represents the logical layer of the architecture that provides new functionalities not currently provided by IoT OS services. Such new services revolve around the descriptive and semantic aspects of the thing as a basis for discovering and announcing presence, formulating services and access interfaces, and handling interactions. The host interface layer gives the platform the portability and interoperability needed to maximize its reliance on the IoT OS's services. The interface creates a gateway that manages the interactions between the platform and OS

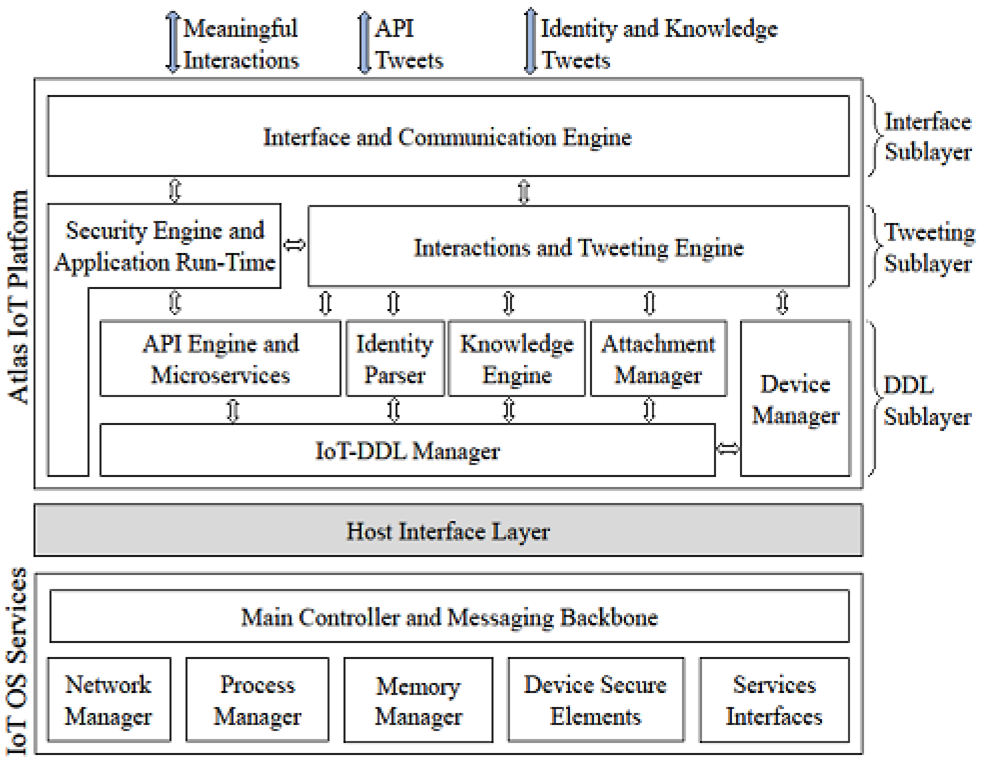

Figure 1: The Atlas thing architecture

services. On systems with little or no operating system support (e.g., Arm Mbed and Arduino), this layer also provides an implementation for the missing required functionalities.

The Atlas IoT platform is further divided into three sublayers: the DDL, Tweeting, and Interface sublayers. The DDL sublayer (the focus of this paper), through the uploaded IoT-DDL file, manages the configuration of the architecture. Such configuration enables the thing to self-discover its own properties and resources (*Identity Parser module*), enables the generation of services (*API Engine*), the creation of programmable representations of the services and their relationships with other services (*Knowledge Engine*), and thing management and lifetime configuration (*Device Manager*). From the service description detailed in Section 2, the API Engine dynamically creates the services (as will be described in Section 5), exposes the appropriate programmable access interface (API) for each, routes API calls, and checks the API inputs' types and ranges.

The DDL layer then utilizes the Tweeting and Interface layers to formulate messages and engage with the smart space, respectively. The current version of the architecture takes advantage of lightweight device management standard OMA-LwM2M [21] and communication standards CoAP [6], MQTT [23], and HTTP REST, along with the capability to interoperate between the different communication protocols through common channels [18].

4 INTER-THING RELATIONSHIP FRAMEWORK

The proposed programming framework introduces three primitives to build IoT applications: 1) Thing Service (TS) – an abstraction of the service offered by a thing to the smart space; 2) Thing Relationship (TR) – an abstraction of how the different TSs are linked together; and 3) Recipe – an abstraction of how the different TSs and TRs build up a segment of an app (an Atlas IoT app is a sequence of recipes). The framework also defines Filter, Match, and Evaluate as three operators that logically and functionally define how the primitives are wired. The thing vendor, Atlas thing (a thing that runs the Atlas thing architecture code) and developer are the main poles of the proposed framework. This section presents the framework's primitives and operators, and highlights the different relationships that can take place between the services along with the different recipe types. This section then presents the semantic rules that govern the establishment of applications as well as the roles of the main poles in establishing the IoT app at the runtime.

4.1 Primitives

The proposed framework introduces Thing Service, Thing Relationship, and Recipe as the three primitives to build an Atlas IoT application, as follows:

a. Thing Service (TS): is an abstraction of a service that an Atlas thing offers to the smart space things and developers. The vendor describes the services offered by an Atlas thing through the IoT-DDL to be uploaded to the thing (as discussed in Section 3.1). The Atlas thing—when powered up or when any change to the service description occurs—parses the IoT-DDL and creates a programmatic abstraction for each service it offers, named Thing Service (TS) in the *knowledge engine* of the DDL sublayer of the architecture. The TS represents the characteristics of the service in terms of the offered functionality, who is offering it, and how it can be accessed. The Atlas thing then advertises these TSs (one TS for each offered service) to thing mates and saves received mates' TSs. It is worth mentioning that the TS object uses the assets and ideas in service discovery protocols defined by SOA [19]. We are using the same idea in a slightly different way to enable the dynamic declaration of relationships between the different TSs in a distributed programming ecosystem. Each TS, as illustrated in Listing 2, describes the service through a set of attributes (*Attributes*) and an interface to access the offered service (*Interface*).

A TS's Attributes are the metadata that describe the characteristics of the service in key-value pairs. The attributes are sub-divided into three groups: a) Identification information for the thing with 'Space ID' [18], 'Thing ID', 'Name', 'Vendor' and 'OS' as the keys, where the values of the keys are extracted from the uploaded IoT-DDL (e.g., identity attributes for a thing that offers GPS can be {(Name, nuvi58LM), (Vendor, Garmin)}); b) Descriptive information that describes the offered functionality using a set of words declared in the IoT-DDL, with 'Keywords' as the key, (e.g., descriptive information for a navigation service could be Keywords: {Location, Map, Route}); and c) Type of the offered service, where the key 'Type' takes *condition*, *report* or *action* as value. Condition is a type of service that examines specific phenomena, returning a domain value if the condition exists and false otherwise (e.g., check if there is a parking spot, return the available spot). Report is a type of service that returns a numerical value (e.g., read a temperature sensor, return the value). Action is a type of service that performs an actuation function, returning a domain value upon a successful call and false otherwise (e.g., turn on the electric switch, return true if the function was triggered).

Structure *TS (Thing Service)*

1- **Attributes**
- *Space ID* //ID for the smart space where the thing coexist
- *Thing ID* //ID for the thing that offers the service
- *Name* //Name of the thing
- *Vendor* //Name of the thing vendor
- *OS* //The operating system the thing is running
- *Keywords* //Descriptive attributes in terms of a set of keywords that describes the offered service
- *Type* //condition, report or action

2- **Interface**
- *Name* //Name of the function
- *Inputs* //Data variables
- *Output* //Domain value if successful execution, and false otherwise

Listing 2: Structure of the thing service (TS)

```
<Entity_1>
  <Descriptive_Metadata>
    <Name>Thermostat</Name>
    <Vendor>Honeywell</Vendor>
    <ATID>AtlasThing128</ATID>
    <SSID>SmartSpace326012</SSID>
    <Description>Manage House Temperature
    </Description>
    ...
  </Descriptive_Metadata>
  <Resource_Service>
    <Service_1>
      <Name>Read Temperature</Name>
      <OutputType>Real</OutputType>
      <OutputName>Temperature Value</OutputName>
      <OutputRange>[0:100]</OutputRange>
      <OutputUnit>C</OutputUnit>
      <InputType>NULL</InputType>
      <Type>Report</Type>
      <Keywords>read, ambiance, AC</Keywords>
      ...
    </Service_1>
    <Service_2>
      <Name>Set Temperature</Name>
      <InputType>Real</InputType>
      <InputName>Temperature Value</InputName>
      <InputRange>[0:100]</InputRange>
      <InputUnit>C</InputUnit>
      <OutputType>NULL</OutputType>
      <Type>Action</Type>
      <Keywords>adjust, ambiance, AC</Keywords>
      ...
    </Service_2>
  </Resource_Service>
</Entity_1>
```

Listing 3: IoT-DDL for the thermostat hardware entity

A TS's Interface provides a direct way to trigger the offered service on the hosting thing. The interface, from the IoT-DDL, is defined in terms of the function's name, inputs, and output. Each input is a data variable that is defined by a short description, data type (e.g., integer, float point) and domain range (the acceptable input values). The output depends on the type of the offered service (condition, report or action). The API Engine of the architecture (as will be detailed in Section 5) handles the dynamic generation of the services, formulation of the appropriate programmable interfaces (APIs), routing of API calls and performing the corresponding check on the expected inputs' types and ranges.

Take a thermostat hardware entity for an Atlas thing as an example. The IoT-DDL's descriptive metadata and services sub-sections [17][18], as illustrated in Listing 3, enable the thing to create a TS that represents each of the offered services. The descriptive metadata sub-section (Line 2 to 9) represents the identification attributes for the offering thing, where ATID stands for Atlas thing ID and SSID stands for smart space ID [18]. The services subsection (Line 10 to 33) represents the parameters and attributes for the two services offered by the thermostat (reading and setting room temperature) in terms of the inputs (types, descriptions, ranges and units) and outputs. Each service is further described in terms of a set of descriptive keywords (Line 19 and Line 30) and the service type (Line 18 and Line 29).

b. Thing Relationship (TR): is an abstraction of a connection between TSs that defines how two or more services are logically and functionally tied to build meaningful applications. The relationships defined by the framework, as will be detailed in Section 4.3, can be: 1) utilized by the vendor in the thing's IoT-DDL as prior knowledge to the thing, 2) utilized by the developer while building applications, and 3) inferred as new programming opportunities by the development environment from the exchanged knowledge between the things (as will be detailed in Section 4.4). The Atlas thing—when powered up or after any change to the relationship description—parses the IoT-DDL and creates a programmable object named a Thing Relationship (TR) for each relationship established by the vendor in the IoT-DDL. The Atlas thing then advertises such TR(s) to thing mates and the Atlas IDE and saves received mates' TRs.

However, due to the lack of knowledge about all services offered by things ahead of their announcements, vendors require a way to establish a TR between an offered service (TS) and an unbounded service (UB). A UB enables the relationship establisher to describe a service, which may not yet be announced by a thing in the smart space, to be matched with one of the TSs offered by an Atlas thing later in time. During the execution of an application, the offered TSs are checked for the closest matches to the UB(s).

Structure TR (Thing Relationship)

1- Attributes
- *Name* //Name of the establisher (e.g., Samsung)
- *Type* //Control, drive, support, or extend for cooperative relations, or contest, interfere, refine, or subsume for competitive relations

2- UB(s)
- *Vendor* //The expected service vendor (e.g., Philips)
- *Type* //Condition, report or action
- *Keywords* //Set of keywords that describes the service
- *Match* //Acceptable match with TS attributes

3- Interface
- *Formula* //Input order and dependencies
- *Inputs* //TS(s), UB (s), Data variable(s)
- *Output* //Domain value if successful, and false otherwise

Listing 4: Structure of the thing relationship (TR)

When a match occurs (as will be detailed in Section 4.2), the UB is replaced with a reference (space id, thing id and TS name) to the closest matched TS (in case of a tie, the first match will be selected). The evaluation of such a TR is enabled only when there is a match for each UB defined in it. Each UB is described in terms of the expected vendor of such service, the service type (e.g., report), a set of descriptive words for the functionality, and the acceptable value of match with a TS. The acceptable value of match reflects how similar a TS should be to replace the UB. Each attribute of the UB may accept the wildcard as input (e.g., to match any TS's vendor, the UB vendor holds * as value).

- Each TR, as illustrated in Listing 4, describes the characteristics of the relationship through a set of attributes (*Attributes*), a set of unbounded services defined by the relationship vendor (*UBs*), and an interface to access the relationship (*Interface*).
- *Attributes*, metadata in key-value pairs that declare who established this relationship with 'Name' as the key and the type of the established relationship with 'Type' as the key (takes one of the following values: control, drive, support, or extend for cooperative relationships, or contest, interfere, refine, or subsume for competitive ones – will be declared in Section 4.3).
- *UB(s)*, one or more unbounded services defined in the TR, each defined with a vendor, type, keywords and match value.
- *Interface*, a direct way to execute the relationship with inputs, formula, and output. The interface input can be a TS or UB, or a data variable defined by a description, type, and domain. The formula reflects

```
1.  <Thing_Relationship>
2.    <Relation_1>
3.      <Establisher_Name>Honeywell</Establisher_Name>
4.      <Type>Contest</Type>
5.      <Unbounded_Services>
6.        <UB_1>
7.          <Vendor>Nest</Vendor>
8.          <Type>* </Type>
9.          <Keywords>ambience, AC</Keywords>
10.         <Match_Required>80</Match_Required>
11.       </UB_1>
12.     </Unbounded_Services>
13.     <Name>Adjust Roomtemperature</Name>
14.     <Inputs>Service_1, UB_1</Input>
15.     ....
16.   </Relation_1>
17. </Thing_Relationship>
```

Listing 5: Extending the IoT-DDL for the thermostat to establish a contest relationship with other thermostat services

Structure *Recipe*:

1- Attributes

- *Name* //Name of the recipe establisher
- *Type* //Simple or conditional

2- Interface

- *Formula* //Inputs order and dependencies
- *Inputs* //TS(s), TR(s)
- *Output* //Domain value if successful, and false otherwise

Listing 6: The structure of the Recipe

how the inputs are processed (will be declared in Section 4.3). The relationships are evaluated independently, even if a single TS is involved in multiple relationships (dependent relationships are outside the scope of this paper).

Consider an expanded IoT-DDL for the thermostat example with a thing relationship sub-section to describe a contest relationship with another Atlas thing, a thermostat from Nest. As illustrated in Listing 5, the other thermostat is described as a UB (Lines 7-10) to be from Nest as 'Vendor', with any type (*condition*, *action* or *report*), and ambience and AC as the 'Keywords'. The defined UB is to be compared for a match with the TSs offered by other Atlas things in the smart space. The acceptable match (to replace the UB with TS) is of value 80. The relationship is of type contest (Line 4) and is established by Honeywell (Line 3). The relationship accepts the service offered by the thing (defined in Listing 3) and the declared UB as the two inputs (Line 14).

c. Recipe: is an abstraction of a connection between different TSs and TRs to build up a segment of an application, where an IoT application is a sequence of one or more recipes. It is worth mentioning that the term Recipe was first used in IFTTT [15] to describe an application; in this paper, we use the same term to describe a sequence of TSs and TRs established by the developer and evaluated sequentially. Each Recipe, as illustrated in Listing 6, describes a segment of an application through a set of attributes (*Attributes*) and an interface through which this recipe is accessed (*Interface*).

- *Attributes*, metadata in key-value pairs that declare who established the recipe with 'Name' as the key and the type of the recipe with 'Type' as the key (either *simple* or *conditional* – will be declared in Section 4.3).
- *Interface*, a direct way to execute the recipe in terms of the inputs, formula, and output. Each input can be TR or TS. The formula reflects the sequence of how the inputs are processed and maintains the required dependencies between them (will be declared in Section 4.3).

4.2 Operators

The framework defines *Filter*, *Match*, and *Evaluate* as three operators that logically and functionally define how the primitives are wired. In this section, we will detail the operations and the attributes that configure each of the three operators.

a. Filter accepts a set of TSs and selects a subset according to preferences. A preference (declared in Equation 1) is a key-value pair that represents one of the TS attributes' keys (e.g., service type, service vendor) while the value is declared by the operator establisher (e.g. developer). The filter operator accepts (as declared in Equation 2) *n* TSs and *m* preferences then selects the subset of TSs that follows the input preferences (e.g., for TSs from 'Philips', the establisher uses (Vendor, 'Philips') as the preference – where Vendor is one of the TS's attributes while 'Philips' is the value declared by the establisher). The operator can be extended to accept a set of TRs and select a subset (e.g., get all relationships established by 'Samsung' as Name – where Name is one of the TR's attributes).

The preference, through the filter operator, can: 1) be optionally negated using the logical negation ($\neg$), thus the logical negation of the preference (Vendor, 'Philips') selects services from all vendors other than 'Philips'; and 2) be accumulated with other preferences using the logical AND ($\wedge$), OR ($\vee$), and XOR ($\oplus$) operators to select one or more TSs. Thus, as illustrated in Equation 2, the operator filters a set of *n* TSs {TS_1, TS_2, … TS_n} into either: 1) {TS_i, TS_j, … TS_k} as the subset of services that follows the logically linked and optionally negated *m* preferences (P_1, P_2, … P_m), where

i, j, and k are the indexes of the selected TSs in the original set, and $1 \le i, j, k \le$ n; or 2) an empty set $\{\emptyset\}$ when no service from the n TSs follows the preferences.

$$\begin{gathered} Preference\ p = (Key, Value), \\ where\ Key \in \{\text{Name}, \text{Vendor}, \text{OS}, \text{Type}, \text{Keywords}\} \end{gathered} \tag{1}$$

$$\begin{gathered} Filter_{(\sigma p1) \ominus (\sigma p2) \ominus \dots (\sigma pm)} \{TS_1, TS_2, \dots TS_n\} \\ where\ \sigma\ is\ an\ optional\ \neg\ and\ \ominus \in \{\wedge, \vee, \oplus\} = \\ \begin{cases} \{TS_i, TS_j, \dots TS_k\},\ TSs\ follow\ preferences\ where\ 1 \le i, j, k \le n \\ \{\emptyset\},\ otherwise \end{cases} \end{gathered} \tag{2}$$

b. Match measures the similarity between an unbounded service UB declared in a TR and a TS offered by an Atlas thing in the smart space through two methods. The first method (Equation 3) accepts a UB and a TS as input and measures the similarity between the attributes' values (vendor, type, keyword). The calculated match value (initially zero) is increased by a constant value V_1 (defined in the framework), for each match in the vendor or the type. The value is also increased by a constant value V_2 (defined in the framework) for each word in the UB's keywords that exists in the TS's keywords. The two positive constant values (V_1 and V_2) are declared and configured by the development environment that implements the proposed programming model (e.g., Atlas IDE – Section 4.5.c) to reflect the weight of each of the attributes (vendor, type, keyword) on the calculated match value.

The two positive constant values (V_1 and V_2) can be set as required by the development environment or the developer's preferences. The match value is then compared to the acceptable match value defined in the UB object. The higher the match value is, the closer the TS is to replacing the UB. In order to find a match to a UB, the development environment should apply the first method (Equation 3) on each of the available TS in the smart space. However, the second method utilizes the available relationships in the smart space for an efficient search for a match. For a smart space with n TSs and m TRs, if a UB is related to TS_x (x is the index of this TS in the n TSs where $1 \le x \le n$) through a relationship TR_i, (i is the index of this TR in the m TRs where $1 \le i \le m$).

This method (Equation 4) first checks if TS_x also exists in another relationship TR_j (j is the index of this TR in the m TRs where $1 \le j \le m$ and $i \neq j$) that is: 1) of inverse type with TR_i (control/controlled-by, support/supported-by, extend/extended-by or drive/driven-by for cooperative relationships, or refine/refined-by or subsume/subsumed-by for competitive ones –Section 4.3); or 2) of same type with TR_i (both are either contest or interfere). The method then applies the first method (Equation 3) between the UB declared in the TR_i and the other TS(s) declared TR_j for a probabilistic match.

c. Evaluate accepts either a TS or a TR (as declared in Equation 5) and triggers the *interface member* defined in the corresponding object. A TS's Interface (as illustrated in Section 4.1.a) provides a way (API call) to trigger the offered service on the hosting thing. Such API call is defined in terms of the function's name, required inputs (data variable defined by a description, type, and domain), and expected output. The TS is evaluated by an announcement to the thing that offers

$$\begin{gathered} Match\ (UB, TS) = mvalue, where\ mvalue = \\ \begin{cases} mvalue + V_1, & if\ UB\ and\ TS\ are\ of\ same\ Vendor \\ mvalue + V_1, & if\ UB\ and\ TS\ are\ of\ same\ Type \\ mvalue + V_2, & for\ each\ common\ keyword\ in\ UB\ and\ TS \end{cases} \\ where\ 0 \le V_1\ and\ 0 \le V_2 \end{gathered} \tag{3}$$

$$\begin{gathered} Match\ (TR_i, TR_j) = Match\ (UB, TS_y)\ where \\ \begin{cases} There\ is\ a\ comon\ service\ (TS_x)\ in\ both\ TR_i\ and\ TR_j \\ TR_i\ and\ TR_j\ of\ inverse\ types\ ||\ both\ are\ contest\ or\ interfer \\ UB\ is\ declared\ in\ TR_i\ \&\ TS_y \in services\ declared\ in\ TR_j \\ 1 \le i, j \le m\ \ \&\ \ i \neq j\ \ \&\ \ 1 \le x, y \le n\ \ \&\ \ x \neq y \end{cases} \end{gathered} \tag{4}$$

$$\begin{gathered} Evaluate\ (prim), where\ prim \in \{\text{TS}, \text{TR}\} = \\ \begin{cases} Initiate\ service\ API\ call\ for\ the\ offering\ thing\ , & if\ prim\ is\ TS \\ \begin{pmatrix} trigger\ a\ Match\ for\ each\ input\ UB \\ recursively\ Evaluate\ each\ input\ TS\ and\ matched\ UB \end{pmatrix}, & if prim\ is\ TR \end{cases} \end{gathered} \tag{5}$$

the service with the API call, where the API Engine of the architecture of such thing (as detailed in Section 5) routes the captured API call and perform the corresponding check on the expected inputs' types and ranges then triggers the corresponding service.

At the same time, a TR's Interface (as illustrated in Section 4.1.b) details a way to evaluate the relationship. Such interface is defined in terms of the expected inputs (TS or UB, or a data variable defined by a description, type, and domain), the formula (as detailed in Section 4.3) that reflects how the inputs are processed, and the expected output. Evaluating a TR first requires finding a match (through the match operator) for each UB defined as input to the TR. If there is a match for each declared UB, then -according to the relationship formula and type- evaluating TR requires the evaluation (Equation 5) to each TS defined in the input and each TS that replaces a declared UB (as detailed in Equation 3).

Thus, for a smart space with n services and m relationships, to evaluate TR_i (i is the index of this TR in the m TRs where $1 \le i \le m$) with p input TSs (where $1 \le p \le n$) and q input UBs (where $1 \le q$), the evaluate operator: 1) applies either the first or the second match methods to find match for each of the q UBs from the available *n* TSs; and 2) triggers a recursive evaluate call for each of the input *p* TSs and the *q* matched UBs, as detailed in the interface's formula and relationship type.

4.3 Types Relationships and Recipes

The framework so far has introduced *TS*, *TR* and *Recipe* as the primitives of an app with *Filter*, *Match* and *Evaluate* as the operators to wire these primitives. In this section, we introduce the types and formulas of the different relationships and recipes defined by the framework.

A. Types of relationships and formalizations

A relationship between two or more services is either cooperative (control/controlled-by, drive/driven-by, support/supported-by, or extend/extended-by) or competitive (contest, interfere, refine/refined-by, or subsume/subsumed-by) as follows:

- ***Control/Controlled-by*** evaluates TS_b (Equation 6) if evaluating TS_a results in logical true for condition *C*. The control condition *C*, utilized by the relationship establisher, either reflects the successful evaluation of TS_a (e.g. Pressure sensor checks the existence of someone in the room, Philips hue turns on the light when someone exists in the room) or the numerical output of evaluating TS_a is mathematically comparable to an input real or decimal value. As declared in Equation 6, TS_a is said to control TS_b (TS_b is said to be controlled-by TS_a). Control can be extended (Equation 7) to either evaluate TS_b or TS_c if evaluating TS_a results in logical true or false for condition C, respectively. The indices *a*, *b* and *c* refer to the first, second and third TS respectively in the input set of TSs to the TR's interface. Control can be extended to sequentially evaluate a set of services based on the evaluation of condition C.

$$Evaluate\ (TS_a) \xrightarrow{C} Evaluate\ (TS_b),\quad where\ C = \begin{cases} Successful\ call\ for\ TSa \\ TSa.output \circ value,\ \circ \in \{=,\neq,<,>\} \end{cases} \tag{6}$$

$$Evaluate\ (TS_a) \xrightarrow{C} Evaluate\ (TS_b); Evaluate\ (TS_c) \tag{7}$$

- ***Drive/Driven-by*** feeds the output of evaluating TS_a (Equation 8) to the input (for the TS's interface member) required for the evaluation of TS_b (e.g. Read the value of temperature sensor, and turn on AC accordingly). As declared in Equation 8, TS_a is said to drive TS_b (TS_b is said to be driven-by TS_a). The indices *a* and *b* refer to the first and second TSs respectively in the input set of TSs to the TR's interface. Drive can be extended (Equation 9) for a nested sequence of output-input feeds.

$$Evaluate(TS_b.Interface.Input(\ Evaluate\ (\ TS_a\ ))) \tag{8}$$

$$Evaluate(TS_n.Interface.Input(\dots(Evaluate(TS_a)))) \tag{9}$$

- ***Support/Supported-by*** enables the evaluation of TS_a (Equation 10) to be the pre-condition for the evaluation of TS_b (e.g. the proper display of an indoor TV requires the window blinds to close, thus the window blinds support the indoor TV). As declared in Equation 10, TS_a is said to support TS_b (TS_b is said to be supported-by TS_a). The indices *a* and *b* refer to the first and second TSs respectively in the input set of TSs to the TR's interface.

$$Evaluate(TS_b), If\ Evaluate(TS_a)\ is\ true/successful \tag{10}$$

- ***Extend/Extended-by*** concatenates the interfaces of TS_a and TS_b (Equation 11) into the interface of a newly created TS ($TS_{extended}$) (e.g. A DVR that records a TV channel, thereby enriching the functionalities of a smart TV that can display channels). As declared in Equation 11, TS_a is said to extend TS_b (TS_b is said to be extended-by TS_a). The indices a and b refer to the first and second TSs respectively in the input set of TSs to the TR's interface. The operation of Extend is algorithmically illustrated in Listing 7.

The interface of *Extend* relationship

```
• Inputs    //inputs for TSa and inputs for TSb
• Output    //Result_Set, false by default
• Formula
    $Result_a$ = Evaluate ($TS_a$)
    $Result_b$ = Evaluate ($TS_b$)
    If $Result_a$ is false OR $Result_b$ is false then  exit
    else   add both $Result_a$ and $Result_b$ to Result_Set
    end if
```

Listing 7: The *Extend* relationship

$$TS_{extended}.Interface(TS_a.Interface, TS_b.Interface) \tag{11}$$

- ***Contest, Interfere*** contest refers to two or more TSs that provide mutually exclusive solutions to the same problem (e.g. Garmin as GPS device, and a smartphone offering GPS service through Google maps), and Interfere refers to two or more TSs are considered inappropriate or insecure to coexist at same time and space (e.g. Turn off smoke detector and turn on the oven). Both types follow the same formula (Equation 12), where the relationship establisher (e.g., developer, vendor) filters a set of *n* competitive TSs through a set of *m* preferences (Equation 1), then evaluates the resulting TS (in the case that more than one TS is filtered, the formula selects the first one in the filtered set to evaluate).

$$Evaluate\ (Filter_{p\,1,\ p\,2,\dots p\,m}\ (TS_1, TS_2, \dots TS_n)) \tag{12}$$

Table 1: Atlas IoT application Semantic rules

Semantic Rule	Description
App = {Recipe}+	Atlas IoT app is a sequential set of one or more Recipes.
Resource = Relationship | Service	Resource can be either a Service or Relationship.
Recipe = {Resource}+ |C→ {Resource}+ |C→ {Resource}+; {Resource}+	Recipe is a sequential set of one or more resource. Execute set of resources for a true evaluation of condition C. Execute the first resource set if condition C is evaluated to true or the second set otherwise.
Relationship = Resource Connection Resource	Relationship is a connection between two resources that indicates how these resources are executed.

Connection =
{Control|Support|Extend| Drive}
or{Subsume|Refine|Interfere|Contest}
	The cooperative and competitive relationships.

Service = Report | Action | Condition	Returns a numerical value. Performs actuation service. Checks the occurrence of a specific event.
C = True | False | Not C | A OPR A | C OPL C | Evaluate Service Type Condition	Logical True Logical False Negation of expression Apply relational operator on arithmetic expressions Apply logical operator on logical expressions The result of evaluation a Service of Type Condition
A = n | A OPA A | Evaluate *Service Type Condition* | Evaluate *Service Type Report* | Evaluate *Service Type Action*	Holds a numerical value Apply arithmetic operator on arithmetic expressions The result of evaluation a Service of Type Condition The result of evaluation a Service of Type Report The result of evaluation a Service of Type Action
OPA = + | * | / | -	The arithmetic operations
OPR = < | > | == | !=	The relational operations
OPL = AND | OR | XOR	The logical operations

- ***Refine/Refined-by, Subsume/Subsumed-by*** refine refers to TS_a that offers more specific functionality compared to TS_b (e.g. Wifi triangulation for indoor positioning, and proximity beacons for indoor positioning), and Subsume refers to TS_a that offers functionality which is included within that offered by TS_b (e.g. Stand lamp turns on the light, and Philips hue controls the brightness). Both types follow the same formula (Equation 13) where an evaluation call for TS_a is triggered if both TS_a and TS_b are currently offered by Atlas things in the smart space. The indices a and b refer to the first and second TSs respectively in the input set of TSs to the TR's interface.

$$Evaluate\ (TS_a), If\ Both\ TS_a\ and\ TS_b\ are\ available \quad (13)$$

B. Types of recipes and formalizations

This section introduces the different types of recipes and the corresponding formulas defined in the interface member of the recipes.

- ***Linear*** evaluates one or more services and relationships sequentially (Equation 14). The indices a and n refer to the first and last primitives respectively in the input set to the recipe's interface. The linear recipe (Equation 15) can also accumulate the output results of evaluating the input primitives using the logical AND, OR and XOR operations.

$$\{Evaluate\ (prim_a), \ldots\ Evaluate(prim_n)\},\ where\ prim \in \{\mathrm{TS}, \mathrm{TR}\} \quad (14)$$

$$\{Evaluate\ (\sigma prim_a)\ \ominus\ \ldots\ Evaluate(\sigma prim_n)\},\ where\ prim \in \{\mathrm{TS}, \mathrm{TR}\}, \sigma is\ an\ optional\ \neg, \ominus \in \{\wedge, \vee, \oplus\} \quad (15)$$

- ***Conditional*** evaluates one or more application primitive (TS, TR) if the logical result of evaluating the first app primitive (TS, TR) is true/successful (Equation 16). The indices a, b and n refer to the first, second, and last primitives respectively in the input set to the recipe's interface.

$$\{Evaluate\ (prim_b), \ldots\ Evaluate(prim_n)\},\ If\ Evaluate\ (prim_a)\ is\ successful\ where\ prim \in \{\mathrm{TS}, \mathrm{TR}\} \quad (16)$$

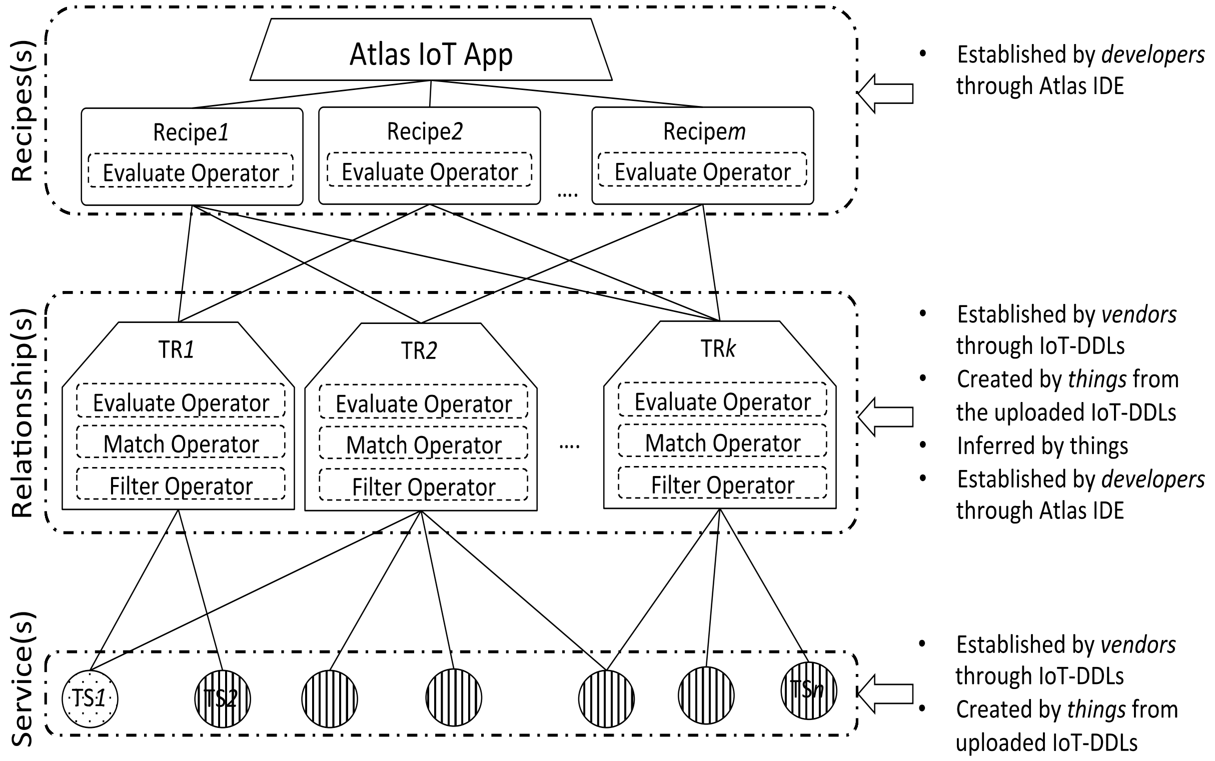
Recipes(s)
Atlas IoT App
Recipe*1*
Evaluate Operator
Recipe*2*
Evaluate Operator
Recipe*m*
Evaluate Operator
Relationship(s)
TR*1*
TR*2*
TR*k*
Evaluate Operator
Match Operator
Filter Operator
Service(s)
TS*1*
TS*2*
TS*n*
• Established by *developers* through Atlas IDE
• Established by *vendors* through IoT-DDLs
• Created by *things* from the uploaded IoT-DDLs
• Inferred by things
• Established by *developers* through Atlas IDE
• Established by *vendors* through IoT-DDLs
• Created by *things* from uploaded IoT-DDLs

Figure 2: Atlas IoT application primitives and operators tree

4.4 Primitives Interplay and Semantic Rules

For a uniform way to establish and to evaluate the validity of Atlas IoT applications, we present the semantic rules (Table 1) that describe applications in terms of the services, relationships and recipes that formulate their structures. The rules: 1) describe the application in terms of the individual services, relationships and recipes; 2) ensure the correctness and compatibility of the application; and 3) govern the execution. The application is composed of one or more sequential recipes. The recipe is a sequential set of one or more resources, where the resource can either be a relationship or a service. A relationship is either a cooperative or competitive connection between two resources (service or another relationship). These semantic rules enable the seamless composition of different types and complexities of applications ranging from a simple service to a composite relationship (relationship that imposes upon another relationship).

Figure 2 illustrates the app primitives that build up an IoT app in a top-down tree fashion and how such primitives are connected through the defined operators (filter, match and evaluate). The recipes (the high level of the tree) are established by the app developer where each recipe (linear or conditional type) is composed of one or more TRs. The TRs (the middle layer of the tree) are either established by the vendors through the IoT-DDLs, established by the developer through the IDE (will be detailed in Section 4.5), or dynamically inferred by the IDE from the exchanged knowledge between the Atlas things. Each TR (cooperative or competitive type) is composed of one or more TSs. The TSs (the low level of the tree) are established by the vendors through the things' IoT-DDLs and are created by the Atlas things from the uploaded IoT-DDLs.

4.5 Poles of the Framework

The poles of the framework to build an IoT app are the thing's vendor, the Atlas thing, and the developer. In this section, we explain in detail the role of each pole.

a. Thing vendor: utilizes the Atlas IoT-DDL web tool [14] to declare an IoT-DDL to be uploaded to the thing. Such IoT-DDL (as declared in Section 3) reflects the thing's identity, entities, services, and relationships. The vendor also utilizes the OMA-DM device management server [21] to send authorized updates during the lifetime of the Atlas thing through the device manager module of the Atlas thing architecture.

b. Atlas thing: creates, at runtime, a TS programmable object for each service it offers and a TR object for each declared relationship. Such runtime objects reside in the Knowledge Engine of the architecture. The thing creates information-based messages (known in this paper as tweets) describing the newly created TS(s) and TR(s) through the tweeting sublayer of the architecture, then utilizes the interface sublayer of the architecture to announce these tweets to other things in the smart space. At the same time, the thing creates a local graph in the knowledge engine of the architecture with TSs as vertices and TRs as edges.

This graph is updated with the received TSs and TRs from other things, and from the lifetime updates sent by the thing's vendor. The graph enables the thing to keep track of the available services and relationships in the smart space and to route the API calls to the API engine [17] of the architecture for the services hosted by the thing (as will be detailed in Section 5). The API Engine dynamically creates the services from their description in the IoT-DDL, formulates programmable interfaces for the services, captures API calls, performs checks on the expected inputs' types and ranges, and evaluates the service with respect to the inputs.

c. Developer utilizes our Atlas IDE to build IoT apps. The IDE is Java-based tool equipped with a graphical interface to sense the currently available primitives and operators, enabling the developer to establish TRs and recipes, and build up an IoT app tree as illustrated in Figure 2. The tool captures announced knowledge about TS(s) and TR(s) from the things. The IDE, from the exchanged knowledge between the things, can dynamically infer the existence of new relationships as new programming opportunities for the developer. The discovered relationships reflect how the current services can be further related to each other and enrich the programmability of the space to the developer with new service engagement opportunities.

The IDE utilizes the Transitivity, Exchange and Composition properties to discover new relationships from the previously established relationships by the developer and the received relationships from Atlas things. It is worth to mention that these properties (composition, transitivity and exchange) are logical properties that deal with the available relationships and services as black boxes to suggest the probabilistic existence of new relationships. The current version of these properties doesn't consider the linguistic meaning nor the semantical structure that define these service and relationships. To keep the focus of the paper, the implementation details of the IDE is outside the scope of the paper.

- ***Transitivity Property:*** If Service A is in a cooperative relationship (types: Control, Support, Drive or Extend) with Service B and Service B is in a cooperative relationship of the same type with Service C, then the existence of a cooperative relationship of the same type is inferred between Service A and Service C. The same property works for competitive relationships (types: Contest, Interfere, Subsume or Refine) [16]. Take the following examples to illustrate the usage of this property for both cooperative and competitive relationships, respectively: 1) If an alarm clock A controls a coffee maker B and the coffee maker B controls a toaster oven C, then the alarm clock A can control the toaster oven C: 2) If a Garmin GPS A contests a smart phone offering a GPS service B and the GPS service B contests a TomTom GPS C, then the Garmin GPS A can contest the TomTom GPS C.

- ***Exchange Property:*** If Service A is in a cooperative relationship (types: Control, Support, Drive or Extend) with Service B and Service B is in a competitive relationship (types: Contest, Interfere, Subsume or Refine) [16] with Service C, then the existence of a cooperative relationship of the same type is inferred between Service A and Service C. Take the following example to illustrate the usage of this property: If an alarm clock A controls a Bosch coffee maker B and the Bosch coffee maker B contests with a Keurig coffee maker C, then the alarm clock A can also control the Keurig coffee maker C.

- ***Composition Property:*** If Service A is in a cooperative relationship (type: Extend) with Service B and Service B is in a cooperative relationship (types: Control, Support, or Drive) with Service C, then the existence of a cooperative relationship of the same type is inferred between the Service C and the resultant of extending Service A and Service B. The same property works for competitive relationships (types: Contest, Interfere, Subsume or Refine) between Service B and Service C, where the existence of a competitive relationship of the same type is inferred between the Service C and the resultant of extending Service A and Service B. Taking the following example can illustrate the usage of this property: a DVR that records a TV channel enriches/extends the functionalities of a smart TV can be merged together as one extended service. If window blinds support the smart TV for better movie watching experience, then the window blinds also support the extended service of the DVR along with the smart TV service.

5 MICROSERVICES

To handle the runtime and just-in-time API-ing of thing services, the Atlas thing architecture utilizes the Micro-Services framework [7] in the API Engine of the DDL sublayer [17] to facilitate dynamic service generation, registration, and discovery. The Atlas thing

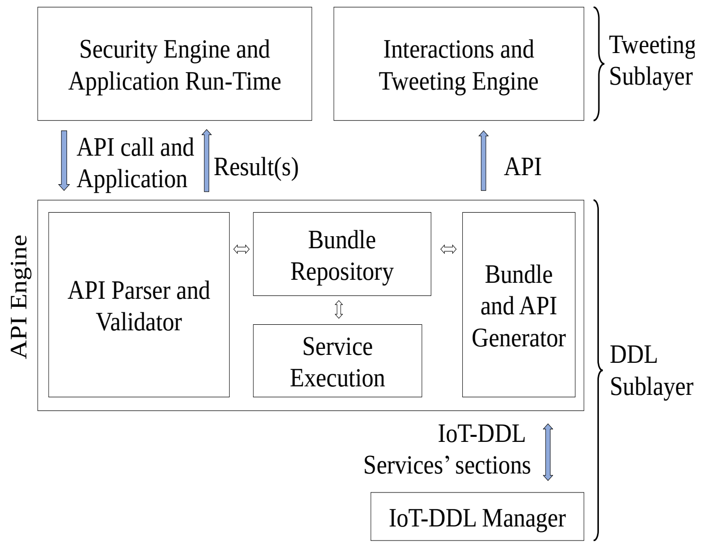

Figure 3: The structure of the API Engine

```
<Service>
    <Description>Set Temperature</Description>
    <Name>SetTemperature</Name>
    <InputType>Float</InputType>
    <InputName>TempCelsius</InputName>
    <OutputType>Integer</OutputType>
    <OutputName>Success</OutputName>
    <Formula>
      <SPIWrite channel=1>TempCelsius/100 +0.5</SPIWrite>
      <DigitalRead pin=12>Success</DigitalRead>
    </Formula>
    <Type>Action</Type>
    Keywords>Control,Temperature,Thermostat</Keywords>
</Service>
```

Listing 8: An IoT-DDL snippet representing a service offered by a thermostat thing

```
int SetTemperature(float TempCelsius) {
        int Success;
        float data = TempCelsius / 100.f + 0.5f;
        spi_write(1, (byte*)&data, sizeof(float));
        Success = digital_read(12);
        return Success;
}
```

Listing 9: The generated C code equivalent for the thermostat service

dynamically generates a software bundle for each service described by the vendor in the thing's IoT-DDL (as declared in Section 3.1). The bundle is a package containing the actual code and resources [27][29] a thing needs to provide the described functionality. The creation of the bundles, as well as installing (adding) and uninstalling (removing) them from a pool of bundles [2] is maintained by the thing through the microservices framework at the runtime.

As illustrated in Figure 3, the *Bundle and API Generator* module converts service descriptions into executable packages (bundles), interacting with the DDL manager and the compiler service provided by the IoT OS Services of the Atlas thing architecture (as detailed in Section 3.2). The *API Generator* then formulates appropriate programmable interfaces (API) for each created bundle. The API is composed of a descriptive name and the expected inputs and output (in terms of the data-type, data-range, unit, and description). The *API Engine* then exposes the created APIs to the Tweeting sublayer of the architecture for advertisement to other things in the smart space. The generated bundle is then passed to the repository, where all bundles are stored. When an Atlas thing captures an API call for one of its offered services, the call is routed to *the API Parser and Validator* module. This module checks on the validity of the input parameters to the interface in terms of the number of arguments and the expected data-type of each input. The *Service Execution module* then retrieves the relevant bundle from the repository and executes the service with respect to the inputs of the API call.

Consider a thermostat service in which the user passes the desired temperature value, as shown in Listing 8. The service takes a floating-point value, issues the command over an SPI interface, and returns a success value on another GPIO pin. From the created TS object from the IoT-DDL, the API Engine generates the bundle for this service in terms of executable code along with the appropriate resources. The bundle interface is synthesized using the given names and types, and the code by mapping the IoT-DDL tags in the formula to executable code provided by the host interface layer of the architecture. The result is a valid

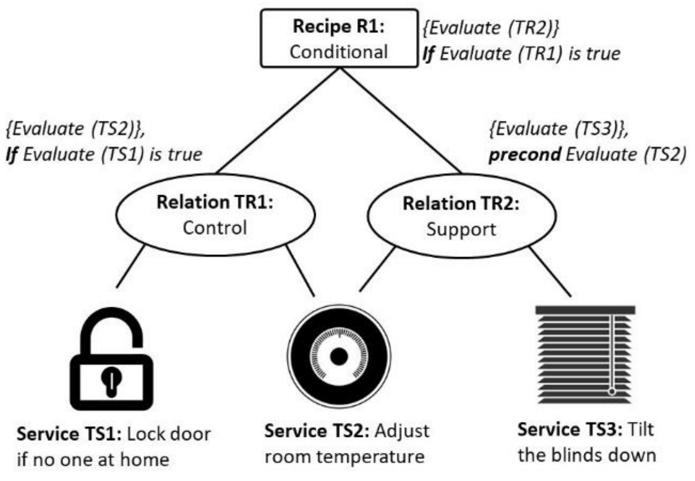
Recipe R1: Conditional
{Evaluate (TR2)}
If Evaluate (TR1) is true
{Evaluate (TS2)},
If Evaluate (TS1) is true
{Evaluate (TS3)},
precond Evaluate (TS2)
Relation TR1: Control
Relation TR2: Support
Service TS1: Lock door if no one at home
Service TS2: Adjust room temperature
Service TS3: Tilt the blinds down

Figure 4: Home automation Atlas IoT application proof-of-concept

```
<Service_1>
  <Description>Lock House</Description>
  <Name>Lock</Name>
  <InputType>Integer</InputType>
  <InputName>LockOrUnlock</InputName>
  <InputRange>[0,1]</InputRange>
  <OutputType>Integer</OutputType>
  <OutputName>Success</OutputName>
  <Formula>
    <DigitalWrite pin=3>LockOrUnlock</DigitalWrite>
    <DigitalRead pin=3>Success</DigitalRead>
  </Formula>
  <Type>Action</Type>
  <Keywords>Security,Lock,Door</Keywords>
</Service_1>
<Service_2>
  <Description>Check if House is empty</Description>
  <Name>NobodyHome</Name>
  <OutputType>Integer</OutputType>
  <OutputName>Empty</OutputName>
  <OutputRange>[0,1]</OutputRange>
  <Forumla>
    <DigitalRead pin=7>Empty</DigitalRead>
  </Formula>
  <Type>Report</Type>
  <Keywords>Energy Saver,Utility,Occupied</Keywords>
</Service_2>
```

Listing 10: An IoT-DDL snippet representing services offered by a smart lock

```
int Lock(int LockOrUnlock) {
            digital_write(3, LockOrUnlock);
            return digital_read(3);
}
int NobodyHome() {  return digital_read(7);  }
```

Listing 11: The generated C codes equivalent for the smart lock services

C function performing the equivalent operations, as seen in Listing 9.

The software bundles, once created, can be loaded (installed) and unloaded (uninstalled) according to the dynamic features of the API calls and the established applications to provide the required services to other things in the smart space. Bundles are loaded as dynamic libraries and can be utilized on any device running the Atlas thing architecture, including boards that do not provide explicit operating system support (such as Arduino and Arm Mbed). The self-contained nature of bundles allows for easy transfer of services, enabling things which cannot generate their own bundles to still obtain dynamic functionality. Once loaded, a bundle exists independently alongside previously created bundles on the thing and may be searched for and referenced from the framework by its interface. At this point, the bundle itself is transparent to the rest of the thing architecture and can be called in the same manner as any normal functionality by the framework and other things.

6 BUILDING IOT APPLICATIONS

In this section, we continue the proof-of-concept scenario started in the introduction section for engaging Atlas things in a smart space. As mentioned earlier, the application illustrates how the framework primitives are instantiated and wired to build a meaningful scenario. The presented application, as illustrated in Figure 4, is a home automation scenario when the smart door locker senses that no one is present at home.

```
Relationship TR1
1- Attributes: (Name, Vendor 1), (Type, Control)
2- UB1: (Vendor, *), (Type, action), (Keywords, ‘thermostat,
   room temperature’), (Match, 20)
3- Interface:  Formula: {Evaluate UB1}, If Evaluate TS1
               Inputs: TS: {TS1} and UB: {UB1}
               Output: true {successful execution} or false
```

Listing 12: TR1 relationship

```
Recipe R1
1- Attributes: (Name, Developer 1), (Type, Conditional)
2- Interface:  Formula: {Evaluate TR2}, If Evaluate TR1
               Inputs: TS: {TS1, TS2, TS3} and TR: {TR1, TR2}
               Output: true {successful execution} or false
```

Listing 13: R1 recipe

The scenario utilizes three Atlas things: 1) a smart lock that locks the home door if no one is home as TS_1; 2) a thermostat that adjusts room temperature as TS_2; and 3) motor-powered window blinds that tilt the blinds down as TS_3. The vendor of each thing, in the corresponding IoT-DDL, declares the services offered by the things, as illustrated in Listing 10.

The vendor of the smart lock thing, in the IoT-DDL, declares a control relationship (Relationship TR_1) with UB that adjusts the room temperature. The vendor of the thermostat, in the IoT-DDL, declares a support relationship (Relationship TR_2) with a UB that adjusts the window blinds as a post-condition for adjusting the thermostat. When the things are powered on, each thing identifies itself, discovers the services it offers, and generates and advertises its TS(s). Each thing, through the API engine and the microservices, creates a bundle with actual code for the service along with the appropriate programmable interface (API), as illustrated in Listing 11.

The smart lock thing generates a TR_1 that reflects the control relationship with a UB as given in Listing 12. The thermostat thing also generates a TR_2 that reflects the support relationship with a UB that adjusts the window blinds as a precondition for adjusting the thermostat. The smart lock thing then starts matching its UB with TSs received from other Atlas things, which is then replaced with a reference to TS_2. The thermostat thing also matches its UB with the TSs for the window blinds control functionality, which is then replaced with a reference to TS_3.

The developer, through the IDE, starts capturing announced knowledge about the TSs and TRs, and establishes a conditional recipe R_1. R_1 evaluates TR_2 if the evaluation of TR_1 is successful (no one exist in the room and the thermostat is adjusted) as given in Listing 13. The developed IoT app tree (Figure 4) is of one recipe, and the IDE parses the tree in top-down approach. R_1 requires evaluating TR_1 first, and if a successful evaluation took place the recipe then evaluates TR_2, where a relationship is evaluated through the interface’s formula with respect to the interface’ inputs and expected output. TS_2 is evaluated if evaluating TS_1 is successful for TR_1, while TS_2 should be true as a precondition to evaluate TS_3 for TR_2. Evaluating the service takes place by sending a request to the offering thing; that thing then utilizes the API engine of the architecture (as detailed in Section 5) to evaluate the API call and return the result back to the IDE.

7 Discussion and Future Work

The current implementation of the Atlas IDE (the development environment that implements the presented programming model - outside the scope of this paper) utilizes the inference properties (composition, transitivity and exchange) to: 1) infer logical relationship possibilities; 2) suggest new recipes with respect to previously established applications and the developers preferences to certain application categories and functionalities; and 3) to present these possibilities in 1) and 2) only as suggestions to the developer.

The presented programming framework as well as the IDE can be extended to extend logical relationship inference into a semantically sound inference operations. Such extension, through the appropriate ontologies and the linguistic analysis, should be considered on the level of: 1) defining a service or relationship through the IoT-DDL by the vendor and establishing relationships through the Atlas IDE by the developer; and 2) by performing semantical and logical validation of the newly inferred relationships (e.g., the operation and environment constraints that guide the relationship between these services).

Through such new extension of the current IDE, we should be able to answer the following important questions: What is the level of expressiveness such tool make available for the developer to describe IoT applications?, How can the IDE validate and verify both the established and the newly suggested applications (e.g., how secure is the application, is there a cycle of dependencies)?, and What is the level of usability of our approach in terms of the capability of the IDE to automatically convert applications’ description into modular structures that improve the execution/validation performance and enable the reuse of the different parts in other applications?

8 Conclusions

We presented the details of the Inter-thing relationships framework for a distributed programming ecosystem for the social IoT. The framework utilizes our Atlas thing architecture and thing IoT-DDL to create a uniform way of describing thing services and service-level relationships, along with new capabilities for the things to dynamically generate their own services, formulate the corresponding programmable interfaces (APIs), and create an ad-hoc network of socially related smart things at runtime. The framework proposed a set of powerful relationships over thing services that can be exploited by developers to build meaningful IoT apps. We presented these relationships in terms of a formal system of primitives and their associated operations, along with the semantic rules to guide developers to build applications. We also discussed the prerequisite roles of the thing, the vendor, and the developer as the three main actors in the framework. Finally, we demonstrated how the framework can be used through a proof-of-concept IoT application.

REFERENCES

[1] L. Atrozi, A. Lera, G. Morabito, and M. Nitti, "The Social Internet of Things (SIoT) – When social networks meet the internet of things: Concept, architecture and network characterization," Computer networks, pp.3594-3608, vol. 56, no. 16, 2012.

[2] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things," In IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1-6, Sep. 2016.

[3] C. Chen, Y. Xu, K. Li, and S. Helal, "Reactive programming optimizations in pervasive computing," In 10th IEEE/IPSJ International Symposium on Applications and the Internet (SAINT), pp. 96-104, 2010.

[4] C. Chen and A. Helal. "Device integration in SODA using the Device Description Language," The IEEE Ninth Annual International Symposium on Applications and the Internet, pp. 100-106, 2009.

[5] C. Chen and S. Helal, "Sifting through the jungle of sensor standards," IEEE Pervasive Computing, vol. 7, no. 4, Dec. 2008.

[6] CoAP, "CoAP RFC 7252 Constrained Application Protocol," http://coap.technology/, 2014.

[7] CppMicroservices, "C++ Micro Services," http://cppmicroservices.org/, 2016.

[8] R. Girau, M. Nitti and L. Atzori, "Implementation of an experimental platform for the social internet of things," In the seventh international conference on Innovative Mobile and Internet Services in Ubiquitous Computing, pp. 500-505, 2013.

[9] S. Helal, "Programming pervasive spaces," IEEE Pervasive Computing, pp. 84-87, vol. 4, no. 1, 2005.

[10] A. Helal and S. Tarkoma, "Smart paces [Guest editors' introduction]," IEEE Pervasive Computing, pp. 22-23, vol. 14, no. 2, 2015.

[11] L.E. Holmquist, F. Mattern, B. Schiele, P. Alahuhta, M. Beigl, and H.W. Gellersen, "Smart - its friends: A technique for users to easily establish connections between smart artefacts," In the proceedings of the International ACM conference on Ubiquitous Computing, Berlin, Heidelberg, pp.116-122, 2001.

[12] A. Helal and Y. Xu, "Scalable and energy-efficient cloud-sensor architecture for cyber physical systems," NSF Workshop on Big Data Analytics in CPS: Enabling the Move from IoT to Real-Time Control, Seattle, April 2015.

[13] H. Hasemann, A. Kröller, and M. Pagel, "RDF provisioning for the Internet of Things," The 3rd IEEE International Conference on the Internet of Things, Wuxi, pp. 143-150, 2012.

[14] IoT-DDL Builder, "Atlas Thing IoT-DDL configuration service," https://cise.ufl.edu/~aekhaled/AtlasIoTDDL_Builder.html, 2016.

[15] IFTTT, "If this then that," https://ifttt.com/, accessed 2017.

[16] A. Khaled and S. Helal, "A framework for inter-thing relationships for programming the social IoT," IEEE 4th World Forum on Internet of Things, pp. 670-675, Singapore, Feb. 2018.

[17] A. Khaled, A. Helal, W. Lindquist, and C. Lee, "IoT-DDL–device description language for the “T” in IoT, " IEEE Access, pp. 24048-24063, vol. 6, 2018.

[18] A. Khaled and S. Helal, "Interoperable communication framework for bridging RESTful and topic-based communication in IoT," *Future Generation Computer Systems*, (in press), 2018.

[19] J. King, R. Bose, H. Yang, S. Pickles, and A. Helal, "Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces," In proceedings of the 31st IEEE conference on local computer networks, pp. 630-638, 2006.

[20] M. Kranz, L. Roalter, and F. Michahelles, "Things that Twitter: Social networks and the

Internet of Things," In What Can the Internet of Things Do for the Citizen (CIoT) Workshop at the 8th international conference on pervasive computing, pp. 1-10, May 2010.

[21] G. Klas, F. Rodermund, Z. Shelby, S. Akhouri, and J. Höller, "Lightweight M2M: Enabling device management and applications for the Internet of things, " White paper from Vodafone, Ericsson and ARM 26, Feb. 2014.

[22] S. Käbisch and D. Anicic, "Thing description as enabler of semantic interoperability on the Web of Things," The IoT Semantic Interoperability Workshop, 2016.

[23] MQTT, "MQTT is a machine-to-machine (M2M)/ "Internet of Things" connectivity protocol," http://Mqtt.org, 2014.

[24] D. Namiot and M. Sneps-Sneppe, "On Internet of Things programming models," In International Conference on Distributed Computer and Communication Networks, pp. 13-24, Springer, Cham, Nov. 2016.

[25] S. Nastic, S. Sehic, M. Vogler, H.L. Truong, and S. Dustdar, "PatRICIA – A novel programming model for iot applications on Cloud Platforms," In the 6th IEEE International Conference on Service-Oriented Computing and Applications (SOCA), pp. 53-60, 2013.

[26] S. Ovadia, "Automate the internet with “if this then that” (IFTTT)," Behavioral & Social Sciences Librarian, pp. 208-211, vol. 33, no. 4, 2014.

[27] D. Shadija, M. Rezai, and R. Hill, "Towards an understanding of microservices," The 23rd IEEE International Conference in Automation and Computing (ICAC), pp. 1-6, Sep. 2017.

[28] C. Turcu and C. Turcu, "The social internet of things and the RFID-based robots," In the 4th international IEEE congress on the Ultra-Modern Telecommunications and Control Systems and Workshops (ICUMT), pp. 77-83, 2012.

[29] T. Vresk, and I. Čavrak, "Architecture of an interoperable IoT platform based on microservices," In the IEEE 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 1196-1201, May 2016.

[30] W3C, "Web of Things at W3C," https://www.w3.org/WoT/, 2017.

[31] Web of Things Framework, "Experimental implementation of the web of things framework 2016," https://github.com/w3c/web-of-things-framework, 2016.

[32] J. Yun, I.Y. Ahn, S.C. Choi, and J. Kim, "TTEO (Things Talk to Each Other): Programming smart spaces based on IoT systems," Sensors, p. 467, vol. 16, no. 4, 2016.

AUTHOR BIOGRAPHIES

Ahmed E. Khaled is currently pursuing the Ph.D. degree in computer engineering, at the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA. He received the B.Sc. and M.Sc. degrees in computer engineering from Cairo University, Egypt in 2011 and 2013, respectively. His current research interests include Internet of Things, smart spaces, and ubiquitous computing.

Wyatt Lindquist received the B.Sc. degree in computer engineering from the University of Florida, Gainesville, FL, USA, in 2017. He is currently pursuing the Ph.D. degree in computer science at the School of Computing and Communications, University of Lancaster, UK. His current research interests include Internet of Things, operating systems, and embedded systems, with applications to digital health.

Abdelsalam (Sumi) Helal (F’15) received the Ph.D. degree in computer sciences from Purdue University, West Lafayette, IN, USA. He is professor and the Chair in Digital Health, School of Computing and Communications, and the Division of Health Research, Lancaster University, UK. Before joining Lancaster University, he was professor in the department of Computer and Information Science and Engineering, University of Florida, USA, where he directed the Mobile and Pervasive Computing Laboratory and the Gator Tech Smart House. His research interests span pervasive systems, the Internet of Things, smart spaces, with applications to digital health and assistive technologies for successful aging and independence.