Re-Engineering IoT Systems through ACOSO-Meth: the IETF CoRE based agent framework case study

Claudio Savaglio*, Teemu Leppänen‡, Wilma Russo*, Jukka Riekki‡, Giancarlo Fortino*
* Dept. of Informatics, Modelling, Electronics and Systems (DIMES), University of Calabria, 87036 Rende (CS), Italy
csavaglio@dimes.unical.it, w.russo@unical.it, g.fortino@unical.it,
‡ Center for Ubiquitous Computing, University of Oulu, FI-90014, Finland
teemu.leppanen@oulu.fi, jukka.riekki@oulu.fi

***Abstract*—The Agent-based Cooperating Smart Objects methodology (ACOSO-Meth) fully supports the systematic development of Internet of Things (IoT) systems from analysis to implementation by tackling their manifold requirements (e.g., self-management, distributed smartness, interoperability). At the same time, ACOSO-Meth allows the re-engineering of existing IoT systems, thus enhancing their maintainability, reusability and extensibility. In such direction, this paper (i) first presents the integration of the resource-oriented agent framework complying with the IETF Constrained RESTful Environment (CoRE) framework into ACOSO-Meth; then (ii) reports a case study to exemplify the re-engineering of a resource-constrained agent application through the ACOSO-Meth metamodel-driven approach.**

Keywords—Internet of Things; Smart Objects; Methodology; Software Agents; Resource-oriented Architecture; IETF CoRE.

I. INTRODUCTION

The Internet of Things (IoT) relies on the seamless interaction among humans, conventional computers and Smart Objects (SOs), namely everyday devices augmented with sensing, actuation, processing, and communication capabilities [18]. SOs, despite their constrained hardware/software resources (i.e., limited memory and processing units, small batteries, lightweight operating systems and communication protocols), are widely acknowledged as fundamental building blocks for realizing ubiquitous IoT systems and disruptive services on top of them, in every application scenario. In particular, SOs are expected to (i) autonomously operate on their local environment, observing and collecting data and executing their tasks; (ii) collaborate to provide cyberphysical services outside the capabilities of a single SO, sharing information and utilizing other SOs' resources; (iii) distribute their smartness and functionality, in a bottom-up fashion, across the whole IoT system. Such advanced behaviors exactly match the distinctive software agents' prerogatives, this is the reason for the wide adoption of the agent-based computing paradigm for developing IoT systems in terms of multi-agent systems (MASs) [17]. Indeed, as opposed to cloud-centric and client/server architectures (which have led to application-specific, unscalable and isolated IoT silos), MASs are a natural selection to realize decentralized architectures based on autonomous, interoperable, cognitive entities, just like SOs [18]. However, beside the promises of an open ecosystem providing innovative cyberphysical services, ***IoT systems and SOs pose challenging development issues like scalability*** (IoT systems have to cope and perform under the massive SO diffusion that is expected within the next few years), ***interoperability*** (IoT systems of different domains have to synergistically interact despite the heterogeneity of SOs' communication protocols, data standards, enabling technologies, etc.), ***efficiency*** (SOs are typically resource-constrained and battery-powered, hence requiring ad-hoc working modalities to perform at their best but saving energy), ***and management*** (massive scaled IoT systems cannot constantly rely on human administrators, thus claiming for automatic government mechanisms and self-steering SOs). Therefore, tackling such complex, heterogeneous cyberphysical IoT systems without systematic approaches and proper development frameworks is definitively ineffective, error-prone and time-consuming.

To address the aforementioned issues and support the IoT systems development, different methodologies have been proposed over the years. Some of them provide domain-specific and technology-dependent approaches. For example, [19] and [20] deal with interoperability solutions for industrial contexts, while [25] focus on the optimal deployment of IoT applications in the Fog. Other methodologies, instead, are general purpose but do not cover the entire IoT systems development process. For example, [21-23] fully support IoT system analysis and design through systematic collections of guidelines, software engineering abstractions and reference models, but providing implementation frameworks and tools is out of the scope. To bridge the gap, ACOSO-Meth [1] has been proposed as a novel full-fledged, application domain-neutral, agent-oriented, and metamodel-driven methodology. It supports the development of SO-based IoT systems (from the preliminary analysis phase to the implementation) as well as re-engineering legacy IoT frameworks to enhance their maintainability, reusability and extensibility (features which cannot be underestimated in the constantly evolving IoT scenario, where novel devices and services steadily show up). ***In this paper, the ACOSO-Meth approach is exploited for re-engineering a resource-oriented software agent framework that is based on the IETF Constrained RESTful Environments (CoRE)*** [2,4,7]. The agent framework (ROAgent) enables heterogeneous, resource-constrained agent-based SOs interoperating in IoT systems in a standardized way. The integration of ROAgent framework into ACOSO-Meth is straightforward, effective and sound, since they share the SO-based vision, IoT system requirements (interoperability,

dynamicity, scalability, etc.) and the enabling paradigms (agent-based and service-oriented computing).

The rest of the paper is organized as follows. Sections II and III introduce ACOSO-Meth and ROAgent framework, respectively. Section IV describes the re-engineering of the framework according to ACOSO-Meth, whose analysis, design and implementation metamodels are instantiated on a ROAgent based SO application in Section V. Finally, conclusions are drawn and future work briefly delineated.

II. ACOSO-METH

Raised from a thorough state-of-the-art analysis aimed at the identification of the fundamental development requirements, ACOSO-Meth [1] supports the engineering of IoT systems through a metamodel-driven approach and the joint exploitation of well-known computing paradigms and programming frameworks (respectively, agent-based computing and ACOSO middleware). Indeed, a set of technology-agnostic metamodels, placed at different abstraction levels but strongly interrelated and completely decoupled from any specific application context, provide generality to ACOSO-Meth approach and drive IoT developers towards the development phases of analysis, design and implementation.

Analysis phase aims at providing an inclusive, but not too complex, high-level SO representation that is compliant with emerging IoT architectural standards such as AIOTI and IoT-A domain models [21, 22]. Indeed, the metamodel of the Analysis phase disregards all behavioral elements but highlights those basic features and static/dynamic information (e.g., physical properties, location, hardware/software characteristics) which are suitable to describe any SO at a first glance and in every domain. Therefore, the High-Level SO Metamodel of this phase represents the basement for the further design and implementation processes (see Figure 3), which conversely provide technology-dependent solutions for both SO-SO and SO-IoT System interactions.

Design phase focuses, more than on SO data, on SO functionalities, i.e., communication, augmentation, service provision and information management. Indeed, metamodel of the Design Phase highlights the functional components of the SO and their interactions, eliciting adopted computing paradigms and enabling mechanisms, independently from specification or low level details. In particular, ACOSO-Meth designs an SO as a software agent according to the ACOSO middleware and its agent model (namely, ACOSO-based SO Metamodel). SO communication is driven by Events handled by a CommunicationManagementSubsystem, which provides a common interface enabling communication toward the SO itself or toward external entities. Similarly, interactions between the SO and its augmentation devices, regardless of their specific technology or protocols, are handled by a DeviceManagementSubsystem. By means of Tasks, the SO exploits these subsystems and reacts to external stimulus, fulfills specific goals and exploits inference rules on local/remote knowledge bases. Both Events and Tasks are specialized, respectively, with regard to their sources (e.g., InternalEvent if the event source is an SO internal component, DeviceEvent if the event source is an SO device) and purposes (i.e., SystemTasks refer to basic SO lifecycle operations while UserDefinedTasks define specific application-oriented SO operations).

Implementation phase finalizes the outcomes of the previous stages and its metamodel (JACOSO Metamodel) elicits the programming paradigms and technology chosen to realize SO functionalities of communication, augmentation, service provision and information management. In particular, for the SO implementation ACOSO-Meth relies on JADE, a FIPA-compliant, open-source and Java-based agent platform whose extensions JADEX and JADE-LEAP also run on constrained devices (e.g., Java Micro Edition-enabled and Android-supported devices, motes of wireless sensors networks). JADE provides an effective agent-oriented management/communication infrastructure, that comprises an Agent Management System (AMS), an ACL-based Message Transport System (MTS) and a Directory Facilitator (DF, which has been purposely extended for realizing a dedicated and dynamic SO discovery service according to the High-Level SO Metamodel data). Aiming at interoperability, two sets of software adapters allow the management of different communication paradigms (direct message passing, asynchronous one-to-may communication, etc.) and augmentation devices (sensors and actuators supporting typical IoT standards such as ZigBee, Bluetooth, etc.).

According to well-known and important good practices of software engineering (more than ever crucial for the IoT context), the ACOSO-Meth approach guarantees:

(i) a suitable and customizable ***degree of abstraction***, essential to address the difficulty of complex IoT systems' modeling, through a high-level analysis metamodel that is incrementally refined and detailed;

(ii) ***modularity and maintainability***, by delegating SO functionalities to loosely-coupled dedicated subsystems;

(iii) ***reusability and extensibility***, with a hierarchical classification of Events/Tasks/Adapters, where most of the code can either be directly reused (frozen-spots) or at least customized (hot-spots, programmed by extension) according to the particular application requirements;

(iv) ***interoperability and evolvability***, with design/implementation choices (e.g., device and communication adapters, uniform interfaces) purposefully aimed at accommodating IoT system heterogeneity and its dynamic evolution.

These properties together make ACOSO-Meth general and flexible enough to facilitate and speed up the development of novel SOs/IoT systems as well as the advantageous re-engineering of existing ones. Indeed, according to both identified SO functionality groups (communication, augmentation, service provision and information management) and development principles (modularity, reusability, etc.), customized design and implementation metamodels of an already existing IoT system can be defined starting from the High-Level SO metamodel. As shown in Figure 1, the ELDA framework has been partially re-engineered [24] (ELDA-based SO aims at being simulated but no executed), while the

ROAgent framework has been fully integrated in ACOSO-Meth, as reported in Section IV. More details on ACOSO-Meth, in particular about the evolution of SO-related concepts from high-level abstractions to implementable software components can be found at [1] (e.g., the SO Service first abstractly presented in terms of Operations and QoS indicators at the analysis phase and then refined at both design and implementation phases as application-level UserDefinedTasks with related ServiceEvents).

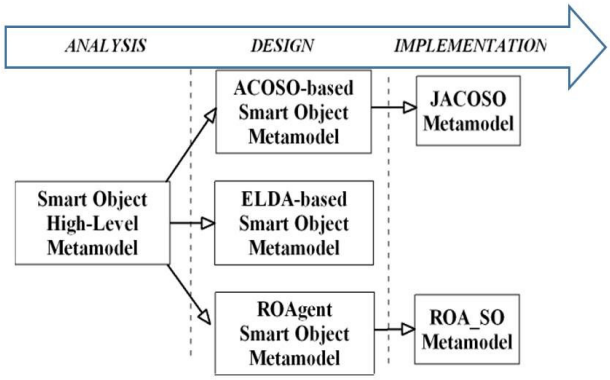

Figure 1 Integration of ELDA and ROAgent frameworks in ACOSO-Meth

III. RESOURCE-ORIENTED AGENT FRAMEWORK FOR IOT

The resource-oriented agent framework (ROAgent) [2,7] is conceptually based on resource-oriented architecture (ROA) [9] and its realization is based on two existing standards. First, the standardized IETF CoRE framework [10] specifications are followed to integrate resource-constrained embedded IoT devices into the Internet through optimized embedded Web protocols. The use of embedded Web services is justified from the interoperability point of view, as Web technologies are currently the only viable solution for Internet-scale interoperability. Within the embedded Web, resources and services of IoT devices become browsable and searchable across organizational boundaries for both human-machine interactions and machine-based automatic IoT application development. Second, the framework provides a subset of FIPA functionality on the IoT devices which operate as agent platforms. The FIPA specifications are followed with regard to agent framework and platforms. However, the ROAgent framework does not fully realize FIPA ACL but, instead, embedded Web protocols are used for agent interactions, as detailed in [7]. This ROAgent framework is currently the only fully REST-compliant agent framework and the only one enabling autonomous agents to interact using embedded Web protocols. Previous work in the context of IoT and software agents has utilized Web services and protocols for interactions [11,13,16], Web documents for hosting agents representations [14,15] and wrappers for translating between ACL and REST interactions [12].

Conceptually, the main abstraction in ROA is a resource, as prescribed by the well-known Representational State Transfer (REST) architectural principles [9]. Generally, everything that has value in the system is abstracted as a resource and identified and addressed through a unique URI. This includes data, IoT devices, their physical components such as sensors, other system components and services in the system, regardless of how they are provided. In addition, REST gives the constraints to define a uniform interface that enables software components, e.g. IoT devices and services, to automatically built distributed applications [9]. The interface is typically realized with well-known Web protocols, such as HTTP or CoAP for the embedded Web, because their request-reply semantics (e.g., GET and POST with resource URIs) are universally known and proven operationally simple. This allows realizing a uniform interface, as a RESTful API, for the IoT system components, including resource-constrained devices [10].

To comply with ROA, the agent framework integrates software agents and their properties into the system as resources [7]. The uniform interface is realized for the framework through two Web protocols, i.e., HTTP and CoAP. This way, same methods are provided for all system components for accessing agent services through the resource abstraction, but also the agents can interact with other system services using the same interface. Figure 2 illustrates the ROAgent framework architecture and infrastructure components. The FIPA DF (including FIPA Agent and Service Directories) functionality is provided with Distributed Resource Directory (DRD) [3]. Following the uniform interface, each system component registers its resources (their URIs) into the DRD. System components can then perform runtime lookups to locate needed resources in the system. The Proxy component is a protocol translator between HTTP and CoAP protocols, allowing seamless bidirectional resource access through the uniform interface. In [7], two types of ROAgent platforms for resource-constrained IoT devices, i.e., Android smartphones and Arduino based embedded boards, are presented. Both platforms provide the minimum required FIPA components, i.e., AMS, MTS and DF (through DRD). In the platforms, either HTTP or CoAP serves as the combined message transport and agent interaction protocol, realizing the RESTful uniform interface. Therefore, the agents' vocabulary for interactions is limited to the protocol method semantics and expressive power, as described in detail in [7]. With regard to agent-based SOs, the framework's RESTful API and the ROAgent architecture were extended in [4] to include SO-specific capabilities and high-level abstractions.

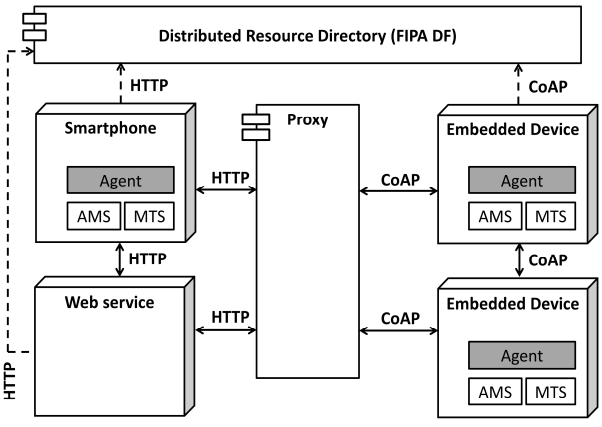

Figure 2. Resource-oriented agent framework

To fully benefit from the simplified operational semantics provided with the REST principles and the uniform interface, the agent architecture needs also to comply with the REST and ROA principles. An agent architecture [2,7] is defined where all the agent components, state and properties are abstracted as resources and thus have their own (hierarchy-based) URIs. The benefit is that the resource abstraction allows to distribute browsability, searchability, interoperability, etc. into the system components and utilize them similarly with the uniform interface, regardless of their location. In other words, the agent-based SO functionality can now be distributed in a fully transparent way [4].

Table 1 illustrates the resource-oriented agent (ROAgent) architecture [2,4]. ROAgent itself has a name, i.e., a unique identifier, that is the agent URI. In the Code segment, the agent has a set of operations that define the agent functionality and provide services. Individual tasks can be integrated into a service. These programs, in turn, utilize the other segments, e.g., knowledge base, and/or interact with local or remote resources through their corresponding URIs. The Resource segment defines the device and system resources that the agent utilizes in its operation. Lastly, the State segment contains the values of agent properties, such as service content, task and interaction results, and metadata such as physical or virtual location and agent creator. Each agent component has its own URIs that are organized in an agent-based hierarchy in the agent architecture. As an example, through the RESTful uniform interface, the agent service content can be retrieved by any system component using the request *GET /agent_name/* or specifying a service with GET */agent_name/service*. Internally, agent starts a task with *POST /task* and receives its results though request *GET /task*. Local resources are accessed by an agent task through *GET /sensor*. Similarly, the agent can access remote resources with *GET /remote_device/sensor* or request an operation for the resource with *POST /remote_device/actuator*.

As described in [2,4,7], the ROAgent framework does not have its own security mechanisms, but it relies on those provided by the IETF CoRE specifications, e.g., identification, authorization, CoAP and lower layer protocol stack security. Indeed, from ROAgent perspective, both agent platforms and DRD (which actually handles authentication and authorization) with its registered resources are considered trusted. Conversely, for agent interactions with remote IoT systems (e.g., external Web services) over Internet, resources should be considered untrusted and are therefore accessed in a secure manner, e.g., through HTTPS or a proxy to maintain security.

IV. Integration of ACOSO and ROA agent framework

In this Section we describe the application of ACOSO-Meth to the ROAgent-based SO (ROA_SO), which is compliant with the ROAgent framework and fully compatible with IETF CoRE framework.

The integration aims at highlighting ROAgent framework own features (e.g., support to interoperability, attention for resource-constrained SOs) and, at same time, providing a higher degree of modularity, maintainability and evolvability. The performed re-engineering process covers the three development phases of analysis, design and implementation, next described in detail.

TABLE I. Resource-oriented agent architecture

Name	Resource URI of the agent		
Code	Operation	Tasks (agent programs)	
		Services (agent programs)	
Resource	Local	Device resources	Data, Sensors, Actuators, Services
	Remote	System resources	Any system resource: Devices and their components, Services, ..
State	Service content		
	Knowledge base	Task results, interaction results	
	Metadata	Location, Physical Properties, Creator, ..	

A. Analysis phase

Metamodel at Analysis phase abstracts SO basic features, categorized in four main groups, as shown in Figure 3.

- *SO BasicInfo* comprises basic SO information and setting. Status contains a list of key-value pairs that capture the SO state, e.g., current SO temperature, residual energy, etc.; Location represents its geophysical position expressed through latitude and longitude coordinates and/or location tags; PhysicalProperty is a list of physical properties of the original object without any hardware augmentation and embedded smartness, e.g., SO dimensions, weight; FingerPrint comprises SO information like the SO identifier, the SO creator, the resource URI, etc.
- *SO Service* models a digital service provided by an SO. Each service is characterized by a name, description, type (e.g., sensing and actuation), input data and output data. Service is implemented by one or more Operations and by one or more QoSIndicators (e.g., latency, accuracy) whose associated values are provided. In detail, an Operation, which defines an individual operation that may be invoked on a service, has a description, a set of input data types necessary for its invocation, and an output related to its result.
- *SO User* identifies an entity using the services provided by an SO. In particular, SO Users can be humans (according to the conventional man-machine use relationship), SmartObjects (SOs can take advantage of the services exposed by other SOs), or DigitalSystems (representing a generic digital entity, like a web service, software agent, robot or a more complex system).
- *Augmentation* defines the hardware and software characteristics of a device that allows augmenting the physical object and making it smart. A device can be specialized in one of the following three categories: (i) Computer, which represents the features of a processing unit of the SO, e.g., PC, smartphone, and embedded device; (ii) Sensor, which models the characteristics of a sensor node of the SO, e.g., presence/temperature/light sensor; and (iii) Actuator, which models the

characteristics of an actuator node of the SO, e.g., led, buzzer, switch.

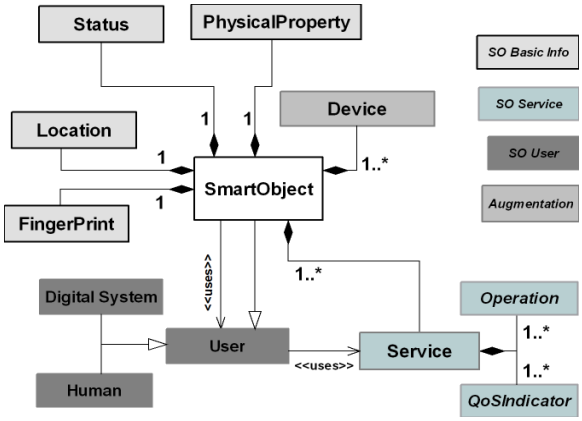

Figure 3. SO High-Level Metamodel at Analysis Phase

B. Design phase

SO High-Level Metamodel is refined at the Design phase to obtain the ROA_SO Metamodel complying with the ROAgent framework, as shown in Figure 4. The SO is modeled as an event-driven, lightweight and platform-neutral agent that exploits the resource abstraction and ROAgent architecture for its operations. The SO lifecycle is specified in terms of Behavior. Behavior consists of one or more state machine-based components named Tasks and coordinated by SO_Manager. Tasks can refer to internal system operations (SystemTask), e.g., management of the SO lifecycle, or to SO application-specific agent-based functionalities (ServiceTask). Inputs required by Tasks are acquired through the uniform interface and resource URIs, providing transparent access regardless of resource type and location. Tasks are driven by Events according to the following interaction model. Whenever an external request or a response to an internal request arrives to the SO, it is caught by the uniform interface (SO_Interface), which creates an Event wrapping the message and delivers it to the SO_Manager. The SO_Manager instructs Tasks responsible of the needed resources on how to process either the request or the response to an earlier request. Similarly, such procedure is followed if, during the request/response processing, the task need to access other resources. After the request/response has been processed, an Event is created that contains the output data, handled again by the SO_Interface and disseminated to the requestor, if needed. This way tasks cooperate internally to the SO and can answer complex or aggregated request.

As done for the Analysis phase, the ROA_SO Design metamodel's elements are categorized into four main groups:

SO BasicInfo reported at the Analysis phase is extended to contain the ROAgent state, its resource descriptions (from the DRD), e.g., tasks and results, their invoking events and the content description of its knowledge base.

SO Service provided by the SO are encapsulated in specific ServiceTasks. The ROA_SO service content is derived from the ROAgent's service content, that typically comprises its task results, and if required, from its knowledge base or through agent's interactions. In the ROAgent architecture, each service and related tasks are invoked using the uniform SO_Interface

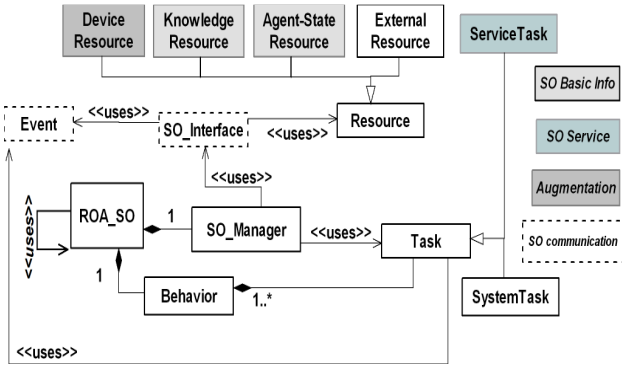

Figure 4. ROA_SO metamodel at Design Phase

by the agent itself or any an external component, e.g., SO Manager.

Augmentation is provided by the Device and External Resources, e.g., SO sensors, actuators, local and remote computing capabilities. To comply with the ROAagent framework, these components are also modeled as resources. The ROA_SO utilizes the uniform SO_Interface and resource abstractions to access any of the hardware or software components it needs, despite their actual location (internal or external to the SO).

SO Communication is performed through the uniform SO_Interface component, which is built on REST principles, i.e., well-known semantics of standardized Web protocols (HTTP or CoAP), their methods, the URIs and protocol response codes [9].

C. Implementation phase

At this phase, ROA_SO Metamodel is implemented with regard to the heterogeneous ROAgent platforms. On one hand, the Android platform is implemented with Java atop the Android OS and provides the hardware/software resources of a modern smartphone. On other hand, the Arduino platform has very limited hardware components, e.g., a 8-bit microcontroller with only a few kilobytes of RAM, and in fact it belongs to the lowest class of devices, resource-wise, defined in the CoRE standards. As shown in [2,8], minimal ROAgent can be implemented into a such platform through the C programming language. The detailed descriptions of the agent platform implementations are given in [2,7].

The Implementation phase Metamodel is the following (Figure 5):

SO BasicInfo are retrieved from the ROAgent knowledge base and task results (the results are the information in the knowledge base that can be used by others or returned to a client). Details of ROAgent task execution and interaction are given in [7]. The knowledge base is updated through the results of Task execution and ROAgent interactions. Due to limited hardware resources in the Arduino platform, typically the responses contain only a few bytes of data and rules regulating Tasks invocation are simple as well, for example sensor data filtering. The platform available memory size further limits the possibility to execute multiple Tasks and the interaction capabilities between Tasks and with resources.

SO Service content is encapsulated into the ROAgent state and typically composed of the knowledge base, task and interactions results. An SO Service is accessed through the uniform interface by using the ROAgent URI. If the ROA_SO includes several Tasks and Services, the results of each computational component can be retrieved through the URI hierarchy. This way also the distributed MAS-based SO functionality [4] can be realized.

Augmentation capabilities refer to the hardware and software components of the SO hosting devices. Android devices typically provide a number of embedded sensors (e.g., accelerometer, gyroscope), but also remote devices (e.g., Bluetooth connected wearables). These all can be modeled as resources for the ROA_SO, whenever the platform provides means to interact with them. In the Arduino platform, due to limited hardware resources, simple sensors and actuators are easy to integrate through predefined I/O interfaces as resources. For interoperability with the IETF CoRE framework, platforms are required to register their available resources into the system DRD, so that ROA_SOs can share and discover nearby resources.

SO Communication is managed by the RESTful uniform interface, implemented either as HTTP (for Android) or CoAP (Arduino) server. In both platforms, SO_Manager and uniform interface functionality are implemented as a part of the FIPA AMS implementation (AMS in Figure 5). This component is also responsible for handling the communication with the IETF CoRE framework components, e.g., to register ROA_SO resources and possibly to interact with the Proxy. IETF CoRE Proxies are expected to follow the uniform interface, which makes the interactions straightforward for ROA_SO. As discussed above, FIPA ACL is not supported in the ROAgent framework due to limited resources in the constrained ROAgent platforms, e.g., Arduino. However, to comply with FIPA specifications, an external resource, e.g., a wrapper, can be introduced that translates FIPA ACL messages into RESTful interactions for the uniform interface as in [12]. The ROAgent interactions should be then directed into this resource whenever ACL-based interactions are required.

V. CASE STUDY: SMARTMOBILITY

To demonstrate the effectiveness of ACOSO-Meth in re-engineering existing IoT systems that have been previously developed without a systematic approach, we present a ROA_SO-based IoT application SmartMobility as a case study.

The SmartMobilityDetector is an SO that provides movement traces of individual users and flocks of users within the premises of the Faculty of Information Technology and Electrical Engineering (ITEE) at the University of Oulu, Finland. Movement traces are initially based on collecting Android smartphone ID's (i.e., MAC addresses) while they are connected to the Wi-Fi access points. To provide indoor high precision location tracking and to perform the flock detection, the devices are expected to periodically perform a Wi-Fi scan, and this operation returns a data vector of detected Wi-Fi access points and their respective signal strengths (RSSI) [5]. The flocks of users are then detected based on similarity measure of the data vectors of different smartphones, as

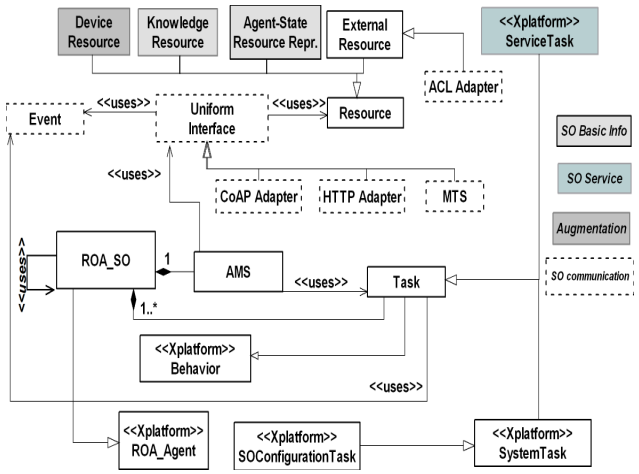

Figure 5. ROA_SO Metamodel at Implementation Phase

described in detail in [6]. In addition, a set of stationary Arduino-based nodes also execute Wi-Fi scans that provide reference information, in their physical location, of the detected access points and their RSSI's. Now, the precise location over time of the smartphones can be estimated with a similarity measure between the access points scan data and nodes scan data. In [6], we outline the following system operation. The IoT devices, i.e., smartphones and Arduino nodes, have joined the ROAgent framework and registered their resources into the DRD. The sensing operation in the devices are controlled by ROAgents, as in [5, 8], which enables the agents to optimize the device energy consumption by controlling the sensing operation in the devices, e.g., scan period and length. In addition, the SmartMobilityDetector runs in a virtual machine (VM) in an IoT edge server that performs the computationally expensive flock detection and trace calculation operations.

Next, we show how the ACOSO methodology is applied for the re-engineering the SmartMobility case study.

A. *SmartMobility Analysis Phase*

The Analysis phase metamodel of ROA_SO SmartMobilityDetector is shown in Fig. 6. As deployed into the ITEE Faculty premises, the SmartMobilityDetector exploits, in the infrastructure side, a number of Arduino nodes (identified as AN#x), which are connected to the ROAgent framework as system resources. This enables them to update their data to the edge server. Also, the Wi-Fi access points (identified as AP#x) are considered infrastructure components and exploited as sensors (Wi-Fi scan).

The deployed devices define the SmartMobilityDetector coverage area (e.g., 100x100x5m). The SmartMobilityService service comprises two Operations: IndividualMobilityDetection, focused on a single smartphone mobility trace, and FlockMobilityDetection, which first identifies a set of flocks by analyzing the sensor data from the devices and secondly provides the aggregated mobility trace. Both operations are invoked by defining the (sub)area/timeframe of interest, and are featured by a certain reliability (calculated as correlation between user data vector and historical movement traces, which provides means for anomaly detection). Client Web services can access, according

to Administrator (security and building maintenance operators) requests, the SmartMobilityDetector through the uniform interface with its resource URI that includes an ID code for the SO (e.g., SMD#1).

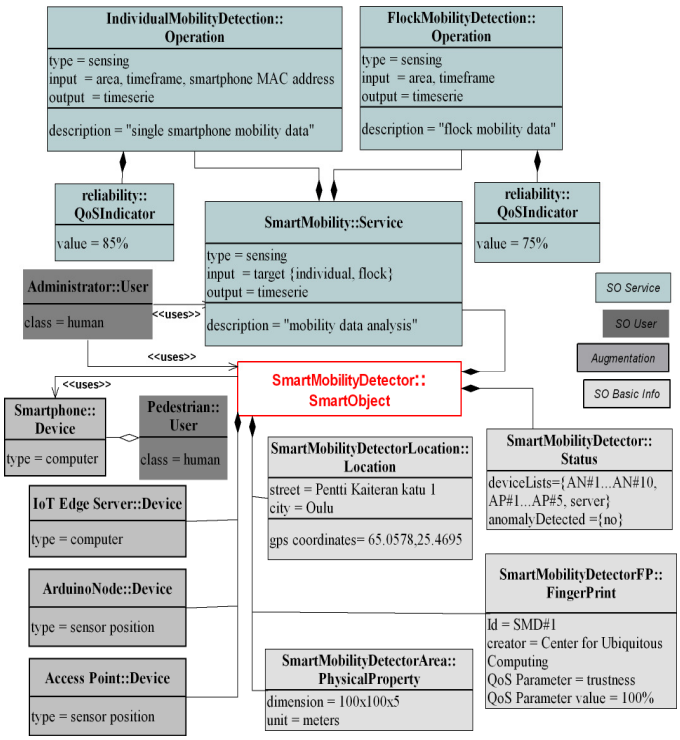

Figure 6. SmartMobilityDetector Analysis Metamodel

B. *Smart Mobility Design Phase*

The refined SmartMobility ACOSO Design metamodel is shown in Figure 7.

The SO BasicInfo consists of the ROA_SO knowledge base and lists of participating devices (from DRD based on real-time information) and infrastructure components. The ROA_SO knowledge base contains the following information: 1) collected raw sensor i.e., Wi-Fi scan data from the devices; 2) task results, i.e., individual movement traces (IndividualTrace Task), detected flocks (FlockDetection Task) and their traces (FlockTrace Task), which are shared among Tasks to refine the information. The SmartMobilityDetector relies on infrastructure components (Wi-Fi access points and Arduino nodes) as DeviceResources, the DRD and ROA_SO ROAgent knowledge base as InternalResource and the pedestrians' smartphones as ExternalResource. The ROA_SO services are implemented as Tasks of the ROAgent. Tasks are driven by Events, which are delivered from the resources through the uniform interface. In details, InternalEvents notify that smartphones join/leave the framework. DeviceEvents notify that new scan data has been uploaded from the Arduino nodes or that Wi-Fi access point information has changed. ServiceEvents (traceEvent and flockEvent) are exploited whenever a task produces results, which are exploitable by other tasks. ExternalEvents enable the interaction with the ROAgent running in the smartphones, e.g., when SmartMobility service sets the required Wi-Fi scanning parameters or a ROAgent in a smartphone informs that its battery level is insufficient to continue participation.

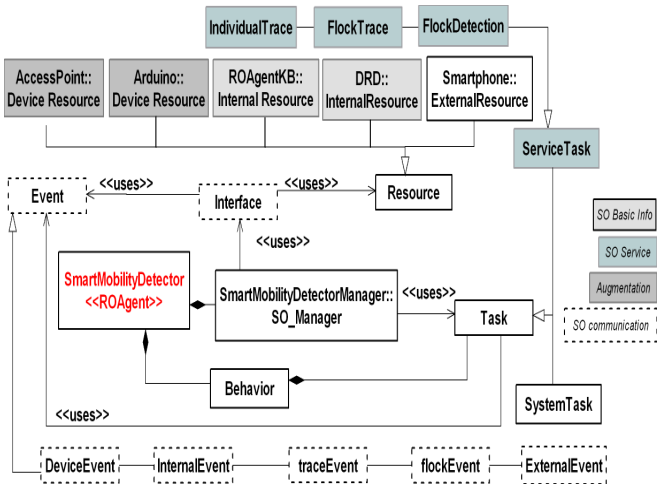

Figure 7 SmartMobilityDetector Design Metamodel

C. *SmartMobility Implementation*

It is expected that the participating IoT devices run the ROAgent platform, which enables autonomous smart operation and also integration into the ROAgent framework. The IoT edge server implements the SmartMobilityDetector as a VM, thus encapsulating SO functionality into an executable unit that can be migrated in the IoT system infrastructure side, following the common solution for edge computing applications.

The ROAgent-based SmartMobilityDetector follows the architecture shown in Table 1 and it is shown in Figure 8. SmartMobilityDetector BasicInfo includes the state of the ROAgent. The knowledge base is implemented in the Android platform as a database and in the Arduino platform as a shared memory component The SmartMobility service is implemented through Operations, i.e., agent programs for flock detection, IndividualMobilityDetection and FlockMobilityDetection. The actual SmartMobility implementation is then a composite of the Task agent programs and their results, which is responsible for event-driven execution of the Tasks. In addition, the ServiceConfigurationTask allows interacting with the ROAgents in the smartphones and Arduino boards, for example, in order to set Wi-Fi scan parameters. SOConfigurationTask, instead, allows setting the SO internal parameters, such as SO creator, ID, etc. The ROAgent platform AMS implements the uniform interface, which enables to utilize internal (ROAgent state and DRD) and external resources (participating devices) and their data in the service execution. To interact with the heterogeneous IoT devices, it is required that the AMS provides both CoAPAdapter (for Arduino boards) and HTTPAdapter (for Android smartphones). This realizes the MTS. As an example, HTTP/CoAP POST method is used to set the Wi-Fi scan operation parameters and GET method (or CoAP OBSERVE method) is used to retrieve data from the devices. The ACLAdapter resource is not required for the SmartMobilityDetector SO.

CONCLUSION

Complexity and requirements characterizing IoT systems and SOs claim for proper development methodology and

frameworks. In this paper the re-engineering of the ROAgent framework according to ACOSO-Meth methodology has been

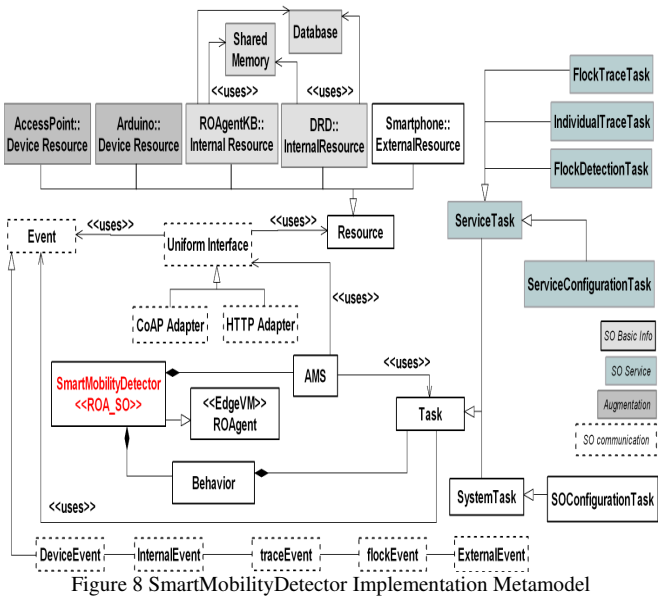

Figure 8 SmartMobilityDetector Implementation Metamodel

presented and exemplified with a ROAgent framework–based crowdsensing application of SmartMobility. In particular, we showed that ACOSO-Meth-based re-engineering highlights and enhances both functional (e.g., support to interoperability, attention for resource-constrained SOs) and non-functional (e.g., modularity, maintainability, evolvability) features of ROAgent framework, due to the full-fledged, domain-neutral and metamodel-driven approach. All the aforementioned features, both functional and non-functional, generate fundamental benefits for complex and heterogeneous IoT scenarios. The integration of ROAgent framework into ACOSO-Meth is straightforward, effective and sound since they share the SO-based vision, IoT system requirements and the enabling computing paradigms (agent-based and service-oriented). As future work, we plan to study the outlined SmartMobility application in real-world settings at the smart university and smart city, to gain deeper understanding on the challenges of resource-constrained IoT development [26, 27].

ACKNOWLEDGMENT

This work has been carried out under the framework of INTER-IoT, Research and Innovation action - Horizon 2020 European Project, Grant Agreement #687283, financed by the European Union.

REFERENCES

[1] G. Fortino, W. Russo, C. Savaglio, W. Shen, and M. Zhou, "Agent-Oriented Cooperative Smart Objects: From IoT System Design to Implementation," IEEE Transactions on Systems, Man, and Cybernetics: Systems, pp. 1–18, 2017. doi: 10.1109/TSMC.2017.2780618

[2] T. Leppänen, M. Liu, E. Harjula, A. Ramalingam, J. Ylioja, P. Närhi, J. Riekki, and T. Ojala, "Mobile Agents for Integration of Internet of Things and Wireless Sensor Networks," in IEEE Int'l Conf. on Systems, Man, and Cybernetics (SMC2013), pp. 14-21, Manchester, UK, 2013.

[3] M. Liu, T. Leppänen, E. Harjula, Z. Ou, A. Ramalingam, M. Ylianttila and T. Ojala, "Distributed resource directory architecture in Machine-to-Machine communications," in 9th IEEE Int'l Conf. on Wireless and Mobile computing, Networking and Communications (WiMob), pp. 319-324, Lyon, France, 2013.

[4] T. Leppänen, J. Riekki, M. Liu, E. Harjula and T. Ojala, "Mobile Agents-based Smart Objects for the Internet of Things," in: Fortino and Trunfio (eds.), Internet of Things based on Smart Objects: Technology, Middleware and Applications, pp. 29-48, Springer, Heidelberg, 2014.

[5] T. Leppänen, T., J. Alvarez Lacasia, Y. Tobe, K. Sezaki and J. Riekki, "Mobile Crowdsensing with Mobile Agents," in Autonomous Agents and Multi-agent Systems, vol. 31, no. 1, pp. 1-35, Springer, 2017.

[6] J. Alvarez Lacasia, T. Leppänen, M. Iwai, H. Kobayashi and K. Sezaki, "A method for grouping smartphone users based on Wi-Fi signal strength," in Forum on Information Technology, paper J-032, Information Processing Society of Japan, 2013.

[7] T. Leppänen, "Resource-oriented mobile agent and software framework for the Internet of Things". Doctoral dissertation, Acta Universitatis Ouluensis, C Technica, no. 645, University of Oulu, ISBN 978-952-62-1813-7, 2018.

[8] T. Leppänen and J. Riekki, "Energy Efficient Opportunistic Edge Computing for the Internet of Things", Web Intelligence, IOS Press, 2018 [Accepted].

[9] L. Richardson & S. Ruby, "RESTful web services", O'Reilly, 2008.

[10] Z. Shelby, "Embedded web services", IEEE Wireless Communications, vol. 17, no. 6, 2010.

[11] P. Leong and L. Lu, "Multiagent web for the internet of things," in IEEE ICISA2014, pp. 1-4, 2014.

[12] L. Braubach and A. Pokahr, "Conceptual integration of agents with wsdl and restful web services", in Int'l Conf. on Autonomous Agents and Multiagent Systems, pp. 17-34, 2012.

[13] J. Jamont, L. Medini and M. Mrissa, "A web-based agent-oriented approach to address heterogeneity in cooperative embedded systems," in Trends in Practical Applications of Heterogeneous Multi-Agent Systems. The PAAMS Collection, pp. 45-52, 2014.

[14] J. Voutilainen, A. Mattila, K. Systä and T. Mikkonen, "HTML5-based mobile agents for Web-of-Things," Informatica, vol. 40, no. 1, 2016.

[15] D. Mitrovic, et al., "The Siebog multiagent middleware", Knowledge-Based Systems, vol. 103, 56-59, 2016

[16] P. Pico, J. Holgado-Terraza, "A Framework for the Development of Smart Ubiquitous Real-Time Systems Based on the Internet of Agents and Internet of Services Approaches", 12th Int'l Conf. on Intelligent Environments, vol. 21, p. 76, 2016.

[17] C. Savaglio, G. Fortino, M. Ganzha, M. Paprzycki, C. Bădică, and M. Ivanović, "Agent-Based Computing in the Internet of Things: A Survey," in Int'l Symposium on Intelligent and Distributed Computing, 2017, pp. 307–320.

[18] C. Savaglio, G. Fortino, and M. Zhou, "Towards interoperable, cognitive and autonomic IoT systems: An agent-based approach," in Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum on, 2016, pp. 58–63.

[19] D. Slama, F. Puhlmann, J. Morrish, and R. M. Bhatnagar, "Enterprise IoT: Strategies and Best Practices for Connected Products and Services," O'Reilly Media, Inc., 2015.

[20] T. Collins, "A methodology for building the Internet of Things," 2017.

[21] A. Bassi et al., "Enabling things to talk," Springer, 2016.

[22] O. Vermesan and P. Friess, "Building the hyperconnected society: Internet of things research and innovation value chains, ecosystems and markets," vol. 43. River Publishers, 2015.

[23] F. Zambonelli, "Towards a general software engineering methodology for the Internet of Things," *arXiv preprint arXiv:1601.05569*, 2016.

[24] G. Fortino, A. Guerrieri, W. Russo, and C. Savaglio, "Towards a development methodology for smart object-oriented IoT systems: A metamodel approach," in Systems, Man, and Cybernetics (SMC), 2015 IEEE Int'l Conf. on, 2015, pp. 1297–1302.

[25] S. Venticinque and A. Amato, "A methodology for deployment of IoT application in fog," Journal of Ambient Intelligence and Humanized Computing, pp. 1–22, 2018.

[26] R. Casadei, G. Fortino, D. Pianini, W. Russo, C. Savaglio and M. Viroli, "Modelling and Simulation of Opportunistic IoT Services with Aggregate Computing," in Future Generation Computer Systems, to appear.

[27] G. Fortino, and P. Trunfio, "Internet of things based on smart objects: Technology, middleware and applications," Springer Science & Business Media, 2014.