SeCoMan: A Semantic-Aware Policy Framework for Developing Privacy-Preserving and Context-Aware Smart Applications

Alberto Huertas Celdrán, Félix J. García Clemente, Manuel Gil Pérez, and Gregorio Martínez Pérez, *Member, IEEE*

***Abstract*—This paper is intended to provide a solution for developing context-aware smart applications preserving the users' privacy in the Internet of Things (IoT). In this sense, we present a framework called Semantic Web-based Context Management (SeCoMan) aimed at offering a set of predefined queries to provide applications with information about indoor location of users and objects, as well as context-aware services. SeCoMan uses a semantic-oriented IoT vision where semantic technologies play a key role. In fact, SeCoMan uses Semantic Web for modeling description of things, reasoning over data to infer new knowledge, and defining context-aware policies. SeCoMan also defines a layered architecture, including functions related to the management of the users' privacy in a manner that accommodate IoT requirements, in addition to not affecting system performance nor introducing excessive overheads. A thorough discussion on other related works, together with some experiments to measure the throughput and scalability, confirm that SeCoMan is a solution that improves the most relevant proposals existing so far.**

***Index Terms*—Context awareness, Internet of Things (IoT), pervasive computing, privacy preserving, semantic reasoner.**

I. Introduction

THE INTERNET of Things (IoT) enables the design and creation of smart objects, exploring new ways of user interaction in smart spaces as well as the development of smart services [1]. Smart spaces are characterized by being areas for cooperation of objects and systems, and for ubiquitous interaction with people. The deployment of smart applications is a complex process due to the lack of frameworks providing support for essential tasks, such as acquiring the information generated by the IoT from various sources, performing context interpretation and inferring new knowledge based on such context, managing rules to dynamically create new knowledge, defining basic location queries that provide context-aware information, allowing users to manage how the framework should use their locations regarding their privacy needs, sharing the knowledge among heterogeneous systems, and providing specific tools to develop smart applications. Many frameworks for developing smart applications use a semantic-oriented IoT vision, where semantic technologies play a key role. In fact, there are solutions based on Semantic Web in a manner that accommodate IoT requirements, but none of them fully supports all the previous tasks.

In order to conduct such tasks, we present in this paper a solution called Semantic Web-based Context Management (*SeCoMan*). Our main contribution behind SeCoMan is to provide support for developing context-aware smart applications preserving the users' privacy in a semantic-oriented IoT vision. Smart applications will be able to gather the information generated by the IoT using a set of queries predefined in SeCoMan, which are categorized into six groups: *operational queries*, providing context-aware information; *location queries*, yielding the indoor location of the elements (objects and people); *range queries*, supplying the elements contained in a given place; *closeness queries*, supplying the elements close to the requester; *navigation queries*, giving the path to arrive to a place or element; and *authorization queries*, providing specific information about the users' permission to stay in a place. The space and context information is shaped in a structured way by using a collection of ontologies [2]. Furthermore, the use of semantic reasoners allows us to infer new knowledge that can be easily shared with other independent systems.

In an IoT context, privacy is a critical issue often overlooked by schemes proposed to date. This fact has been recently identified in [3]. Perera *et al.* argue that privacy is a significant challenging issue in the IoT, and it is largely unattended at the context-aware middleware level in the existing solutions. To address this, SeCoMan supports *semantic rules* to define policies. These policies will allow users to share their location to the right users, at the right granularity, at the right place, and at the right time. Using location policies, users will be able to manage their privacy independently of the applications:

1) *hiding* their locations to other persons;
2) *masking* their locations with fictitious positions;
3) establishing the *granularity* at which they want to be located;
4) defining the level of *closeness* accepted to be located.

Manuscript received September 10, 2013; revised December 19, 2013; accepted December 29, 2013. Date of publication January 21, 2014; date of current version August 23, 2016. This work was supported in part by the Spanish Ministry of Science and Innovation (MICINN) through Project RECLAMO: Virtual and Collaborative Honeynets based on Trust Management and Autonomous Systems applied to Intrusion Management under Grant TIN2011-28287-C02-02 and through Project Mejora de Arquitectura de Servidores, Servicios y Aplicaciones under Grant TIN2012-38341-C04-03, by the European Commission through the European Regional Development Fund, and by the Séneca Foundation through the Funding Program for Research Groups of Excellence under Grant 04552/GERM/06.

A. Huertas Celdrán, M. Gil Pérez, and G. Martínez Pérez are with the Department of Information and Communication Engineering, University of Murcia, 30100 Murcia, Spain (e-mail: alberto.huertas@um.es; mgilperez@um.es; gregorio@um.es).

F. J. García Clemente is with the Department of Computer Engineering and Technology, University of Murcia, 30100, Murcia, Spain (e-mail: fgarcia@um.es).

Digital Object Identifier 10.1109/JSYST.2013.2297707

SeCoMan also manages authorization policies to control who can access (or stay in) a given space. The IoT information is provided by certain location systems and middleware, which are independent to the framework. This independence allows SeCoMan to choose the location systems and middleware depending on the characteristics of the environment.

The remainder of this paper is organized as follows. In Section II, we discuss the related work regarding other context-aware solutions. Section III presents the SeCoMan architecture, whereas the collection of ontologies managed by SeCoMan is described in Section III-B. Taxonomies of policies and queries are presented in Sections IV and V, respectively. Section VI shows the deployment of a smart application making use of SeCoMan that offers advanced services in a supermarket scenario. Section VII reports some experimental results to illustrate the performance of the SeCoMan framework. A thorough discussion comparing our approach with other related systems is performed in Section VIII, and finally, conclusions and future works are drawn in Section IX.

II. Related Work

The large number of objects involved in the IoT makes organization, representation, storage, and sharing is a potentially challenging task. In such a context, "semantic-oriented" IoT visions are available in the literature to provide modeling solutions for things description, reasoning over data generated by the IoT, semantic execution environments, and architectures that accommodate IoT requirements [3], [4]. A common ontology is a key factor to develop context-aware systems and smart applications, as it allows knowledge sharing between independent systems and uses semantic reasoning about the context to offer advanced services to customers.

A recent publication conducted a depth survey on context-aware systems oriented to the IoT, where a large number of solutions are analyzed by considering different topics [3]. Considering the semantic-oriented IoT vision, systems like Feel@Home [5], Hydra [6], CroCo [7], SOCAM [8], and CoBrA [9] provide support to "security and privacy" features, as our solution. Feel@Home is a context-aware framework that supports communications between contexts or domains, considering intra- and interdomain interactions. Hydra is an ambient intelligence middleware system oriented to the IoT, which integrates the device, semantic, and application contexts to offer context-aware information. On the other hand, CroCo is a cross-application context management service for heterogeneous environments, whereas SOCAM uses a collection of ontologies that shapes the quality, dependence, and classification of the context information. This collection is built on a common upper ontology for all contexts, as well as for domain-specific ontologies that define concepts of each one. Another related work in this context, which is not included in the survey presented in [3], is CoCA [10]. This proposal presents a collaborative context-aware service platform, where a neighborhood-based mechanism to share resources is introduced. CoCA infers users' location by considering information about the context and the location of the elements.

Four of the five solutions described make use of semantic rules for different purposes, being Feel@Home the only one that does not. Hydra, CroCo, and SOCAM do use semantic rules to infer new information about a given context, taking into account information from others. Instead, CoCA makes use of semantic rules to manage the ontologies, e.g., a property is the inverse of another property, as well as additional information about the domain. Yet, none of these four solutions uses semantic rules to define policies oriented to protect users' privacy preferences. Users' privacy should be supported by any context-aware framework, with which the users are capable of dynamically restricting or disclosing information to others depending on their location and their preferences in terms of privacy. Consequently, the current trend in context-aware systems focuses on controlling the disclosure of users' location by using policies.

There are a number of systems based on Semantic Web that manage policies to preserve users' privacy. For example, CoBrA presents a context-aware architecture that allows distributed agents to share information with each other. CoBrA defines an ontology that shapes spaces composed of smart agents, devices, and sensors, and protects the privacy of its users by using rules that deduce whether they have the right permissions to share and/or receive information. Another example is PPCS [11], where a semantically rich policy-based framework with different levels of privacy to protect users' information in environments with mobile devices is presented. Dynamic information observed or inferred from the context, along with static information about the owner, is taken into account to make access control decisions. Location and context information of the users are shared (or not) depending on their privacy policies. Another proposal supporting privacy policies without using Semantic Web technologies is CoPS [12]. In CoPS, users can control who can access their context data, when, and at what level of granularity. It organizes policies into different hierarchical levels, defining a default policy according to an optimistic or pessimistic approach.

Despite the work and progress made by the systems discussed, a lot of work is still required to improve key aspects, such as policies and context management, users' privacy, availability and quality of services, and robustness. In Section VIII, we thoroughly discuss and compare our framework with others that also manage users' privacy through policies.

III. SeCoMan Architecture

SeCoMan is a trusted third party that manages users' privacy about their location. It supplies the context and space information provided by the IoT to smart applications that could be not reliable enough for managing this information. The SeCoMan architecture is composed of three layers to allow framework actors to manage the resources and develop applications more efficiently. Fig. 1 shows the components and actors forming the multilayered architecture of SeCoMan.

A. Actors

We defined three kinds of actors to interact with SeCoMan. First, the *Framework Administrator* manages the common

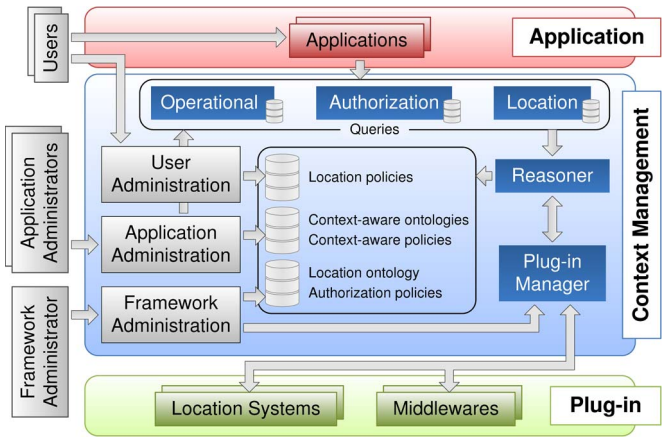

Fig. 1. Overview of the multilayered architecture of SeCoMan.

resources for all contexts, among which we emphasize the management of the *Location ontology* that models the space information and the definition of the *Authorization policies*. Furthermore, this administrator registers the smart applications that can make use of SeCoMan and indicates to the *Plug-in manager* module the location systems and middleware that has to be used to receive the space and context information.

On the other hand, each *Application Administrator* is in charge of managing the context of his/her own applications, handling the *Context-aware ontologies and Policies* as well as more sophisticated queries by combining the *Operational*, *Authorization*, and *Location* ones. Finally, the last kind of actor is *Users*. They are persons who use the smart applications to obtain information about the environment in which they are located. They define their location policies to manage their privacy directly in the framework without having to rely upon the applications. For that reason, our framework acts as a trusted third party for the users as applications might not be reliable enough with the location information that they manage.

B. Layers of the SeCoMan Architecture

The three layers composing the SeCoMan architecture shorten the complexity of the IoT infrastructures and provide the necessary resources for the previous actors; therefore, they can manage smart applications, the space and context information, and the plug-ins for the location systems and middleware. First, the *Application* layer contains smart applications that provide users with specific information about the spaces in which they are located. To that end, the *Applications* will make queries to the *Context Management* layer in order to obtain the space and context information desired by the users.

In order to manage the context, the Context Management layer uses ontologies to shape the information gathered from the *Plug-in* layer, the semantic rules to define the policies that control the system behavior, and the semantic reasoning to infer new knowledge, taking into account the previous information sources. To perform all these tasks, the *Operational*, *Authorization*, and *Location* modules provide smart applications with a number of queries predefined in the framework, which offer certain information regarding these topics. Queries are applied on the new knowledge inferred by the *Reasoner* module. This takes as input the ontological model, formed by the union of the ontologies updated according to the information collected by the *Plug-in Manager* module, and the semantic rules defined by the actors through their corresponding administration components.

Finally, the Plug-in layer obtains the space and context information about the elements that form part of the environment and their locations, as well as further information from these elements depending on the environment. This is composed of different plug-ins that interact, on one hand, with the *Middleware* (which in turn communicate with sensors or other devices to receive context information) and, on the other hand, with the *Location Systems* to obtain information about the space. This layer provides independence to SeCoMan with regard to the location system used, thus allowing Application Administrators to choose the best location system or middleware depending on the characteristics of the environment.

We describe here the main ontology managed by SeCoMan and an example about a supermarket scenario, which will be used through this paper to introduce all concepts related to our proposal. Using this scenario, we implemented a smart application offering location-based services to customers.

C. Location Ontology

SeCoMan defines a collection of ontologies to shape the space and context information. This collection is composed of an ontology called *Location* that models the indoor location, common for all contexts, and a set of ontologies for the smart applications that provide specific services in different contexts. Fig. 2(a) shows the Location ontology. This ontology models the space and provides a set of primitives with which to describe regions of the space and relationships among them.

The Location ontology is categorized into three different but related topics: element, authorization, and space. The top-level class in the element topic is *Element*, which refers to any entity that forms part of the environment (persons or objects). Elements can have several *Roles* and *Privileges* that can be used to provide personalized information. Note that Privilege is the most important class in the authorization topic. Privileges are used to allow Elements to perform certain actions, such as staying in a specific position. The Element class has two predefined subclasses, *System* and *Person*, which are defined to be disjointed. A Person defines the accuracy on the granularity and closeness at which he/she wants to release his/her location by using the *Accuracy*, *Granularity*, and *Closeness* classes. Finally, and in order to support location generalization, the Location ontology uses a hierarchical model for location. *Space* is the top-level class in this model, having five predefined subclasses, namely (from low to high accuracy): *Building*, *Floor*, *Area*, *Section*, and *Position*. Position establishes the *Geographical* or *Absolute Position* of an element, where several Positions form a Section that has two predefined subclasses, i.e., *Corridor* and *Room*.

The Location ontology entities are related each other by properties. A portion of these properties is used to establish new relationships through policies. For example, authorization policies use the *hasAuthzAccess* property to link Persons and

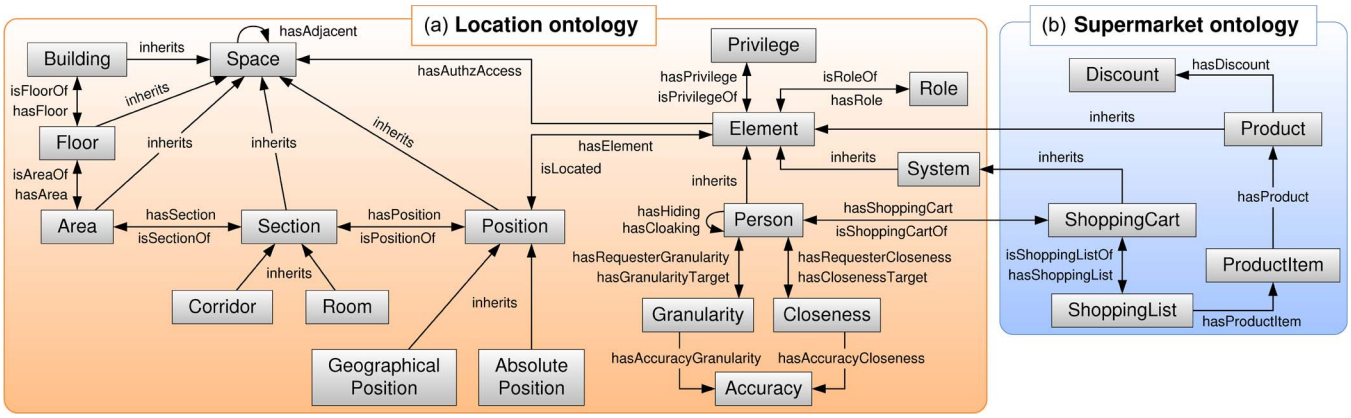

Fig. 2. Ontologies of SeCoMan. (a) *Location* ontology, common for all contexts. (b) *Supermarket* ontology shaping a specific context.

Spaces according to the persons' privileges; location policies generate the *hasCloaking* and *hasHiding* properties between two persons to mask and hide their location, respectively; and *hasRequesterGranularity* and *hasRequesterCloseness* are established by the location policies to link Persons to each other with a specific Granularity and Closeness.

D. Motivating Example

As a proof of concept to show the ease of adding new ontologies to SeCoMan, we implemented a smart application called *eCoMarket* (further details in Section VI). It offers a smart service to customers of a supermarket according to their location. eCoMarket defines an ontology called *Supermarket*, shown in Fig. 2(b), that shapes the supermarket context.

The top-level class in the Supermarket ontology is *Product*, representing an article of the supermarket. This class inherits from Element of the Location ontology, so that this relation is the connection between both ontologies. As it can be observed, a new ontology only has to inherit from the Element class of the Location ontology, or from one of its subclasses, in order to create a link between the two ontologies. In the supermarket context, Products can have *Discounts*, and some of them can belong to a *ShoppingCart* through a *ShoppingList* that contains one or more *ProductItems*.

For clarity, Fig. 3 shows a graphic representation of a basic example about a given instance of a supermarket, in order to clearly follow all elements introduced here. It is worth noting that we defined the corresponding data properties for all classes in the two ontologies shown in Fig. 2. For example, the name and the price of any article of the supermarket were modeled as data properties in the Product class. Although they were not drawn in Fig. 2 for simplicity, the complete definition of both ontologies—classes and object and data properties—can be accessed and downloaded from [13].

In this example, we created entities of the classes defined in the Location and Supermarket ontologies. In this sense, we have a supermarket with one Floor, two Areas, two Corridors, and five Positions. At the supermarket place, there are four persons who can use different Roles, Privileges, Granularities, Closeness, and Accuracies. Specifically, *Peter* has the *GoldCustomer* and *Hidden* roles (R), and he is located at *Position1*.

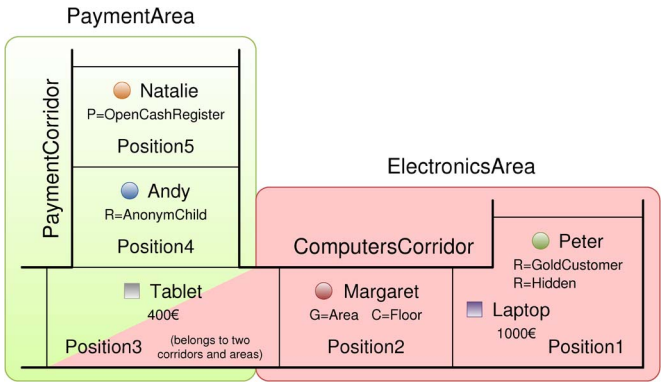

Fig. 3. Map of the supermarket example.

This position belongs to the *ComputersCorridor*, which is in turn located at *ElectronicsArea*. Peter has a ShoppingCart containing a *Laptop*. On the other hand, *Natalie* is located at *Position5* and she has the *OpenCashRegister* privilege (P). Position5 belongs to the *PaymentCorridor* that is located at *PaymentArea*. *Andy* has the *AnonymChild* role, and he is located at *Position4* belonging to PaymentCorridor. Finally, *Margaret* is located at *Position2*, which belongs to the ComputersCorridor, and she has granularity (G) of Area and closeness (C) of Floor. Furthermore, the supermarket has two products: a *Laptop* whose price is 1000€ and is located at Position1; and a *Tablet* whose price is 400€ and is located at *Position3*, belonging to two different corridors and areas.

IV. SeCoMan Policies

Our framework dynamically controls users' privacy, their authorization to stay in certain spaces, and the context information, and generates new knowledge by using semantic rules, which form policies. SeCoMan uses rules that consist of two lists of predicates: the antecedent and the consequent parts of a rule. If all predicates of the antecedent part take the Boolean value true, all predicates in the consequent part are evaluated. It is important to know that, in our semantic rules, the predicates in the consequent part establish new relationships between entities of the ontologies and does not generate new entities.

The policies in SeCoMan are composed of the following elements: *Type* is the kind of policy; *Maker* is the person who

defines the policy (possibly being the same as the Target); *Target* is the person whose information is managed by the policy; *Requester* is the person, or group of persons, who request information about the Target; *Place* is the region of the environment in which the policy is applied; and *Result* determines the relationship that the Requester will have about the Target information. Note that Result is the consequent part of the semantic rule, whereas the remaining fields belong to the antecedent part. SeCoMan also illustrates the possibility of including extensions to make richer and more powerful policies. For example, Makers may improve their policies by tuning elements like *Role*, *Privilege*, *Date*, or *Context*.

In the context of SeCoMan, our framework architecture manages three kinds of policies: *Operational policies*, defined by the Application Administrators to generate new knowledge related to the users' context; *Authorization policies*, defined by the Framework Administrator to decide the authorization of the users to stay or not in a specific space; and *Location policies*, defined by application-independent Users to specify the privacy preferences about their location. We show below an example for each of these policies, making use of the supermarket scenario defined in Section III-D.

A. Operational Policies

Operational policies are used to manage the information of the smart applications, generating new knowledge related to the context-aware ontologies. The antecedent part of the policy is composed of entities belonging to the collection of ontologies of SeCoMan, whereas the consequent one establishes relationships between two entities of which, at least, one of them has to belong to the Location ontology. In the supermarket example, let us suppose that the Application Administrator of the eCoMarket application $(Maker)$ defines that "On July 2013, Persons who have the GoldCustomer role will obtain a 21% off in products located at ComputersCorridor," i.e.,

$$\begin{aligned} &Person(?target) \\ &\wedge isLocated(?target, \#ComputersCorridor) \\ &\wedge hasRole(?target, \#GoldCustomer) \\ &\wedge Product(?product) \\ &\wedge isLocated(?product, \#ComputersCorridor) \\ &\wedge greaterThan\,(\#Today, date(2013, 06, 30)) \\ &\wedge lessThan\,(\#Today, date(2013, 08, 01)) \\ &\rightarrow hasDiscount21(?target, ?product). \end{aligned}$$

Applying this rule to the supermarket use case, Peter $(Target)$ gets a 21% off $(Result)$ on July 2013 in the Laptop he has in his ShoppingCart and in the Tablet located at Position3 as Peter has the GoldCustomer role and both products belongs to ComputersCorridor (see Section III-D).

B. Authorization Policies

Authorization policies are based on privileges to allow users to stay in certain locations according to their privileges [14]. By default, SeCoMan denies the authorization in the absence of rules. These policies are independent of the context; therefore, the consequent part in this kind of rule only generates relationships between entities belonging to the Location ontology (common for all contexts). In the supermarket example, let us suppose that the Framework Administrator $(Maker)$ defines that "Persons located at PaymentArea with the OpenCashRegister privilege have authorized access to be there," i.e.,

$$\begin{aligned} &Person(?target) \wedge isLocated(?target, \#PaymentArea) \\ &\wedge hasPrivilege(?target, \#OpenCashRegister) \\ &\rightarrow hasAuthzAccess(?target, ?position). \end{aligned}$$

Applying this rule to the supermarket use case, Natalie $(Requester$ and $Target)$ has authorized access $(Result)$ to stay at PaymentArea, as she has the OpenCashRegister privilege.

C. Location Policies

Location policies generate new knowledge related to users' location privacy, these being independent of the context. Location policies are divided into four groups: $Cloaking$, $Hiding$, $Granularity$, and $Closeness$. In this kind of policy, the $Target$ is the same person as the $Maker$; therefore, he/she can define rules for the same requester specifying different roles, locations, dates, or times. We explain below in detail these four kinds of location policies managed by the framework.

1) Cloaking: Masking or cloaking is intended to generate one or more fictitious positions for a particular user; therefore, other users cannot distinguish the real position where the target is located. As an example, a cloaking policy applied to the supermarket scenario could be as follows:

$$\begin{aligned} &Person(\#Andy) \wedge hasRole(\#Andy, \#AnonymChild) \\ &\wedge isLocated(\#Andy, \#PaymentArea) \\ &\wedge greaterThan\,(\#Now, time(08, 59)) \\ &\wedge lessThan\,(\#Now, time(14, 00)) \wedge Person(\#Peter) \\ &\rightarrow hasCloaking(\#Andy, \#Peter). \end{aligned}$$

Applying this rule to the supermarket use case, if Peter $(Requester)$ asks about the position of Andy $(Target)$ between 9:00 A.M. and 2:00 P.M., SeCoMan will generate one or more masking positions $(Result)$ for Andy. This is due to Andy being at PaymentArea and he has the AnonymChild role. This provokes that Peter cannot distinguish if Andy is at Position4 (his real position) or at a fictitious one, such as Position3, for example.

2) Hiding: Users can define hiding policies when they do not want to release their location to others, thereby avoiding that requesters know the position of the target. A hiding policy example is shown in the following for the supermarket scenario:

$$\begin{aligned} &Person(\#Peter) \wedge hasRole(\#Peter, \#Hidden) \\ &\wedge isLocated(\#Peter, \#ComputersCorridor) \\ &\wedge Person(?requester) \\ &\rightarrow hasHiding(\#Peter, ?requester). \end{aligned}$$

Applying this rule to the supermarket use case, if someone $(Requester)$ asks about Peter's location $(Target)$, SeCoMan will not return his location $(Result)$, as Peter is at ComputersCorridor and he has the Hidden role. This position is hidden, as requested by Peter, through the $hasHiding$ property.

3) Granularity: Granularity policies are used to indicate the maximum accuracy at which users want to be located. As stated in Section III-B, there are various levels of granularity that can be applied to a given location, namely: $Position$, $Section$, $Area$, $Floor$, and $Building$. An example of policy of this type in the supermarket scenario could be as follows:

$$Person(\#Margaret) \wedge Person(?requester) \wedge Granularity(?granularity) \wedge hasGranularityTarget(\#Margaret, ?granularity) \wedge hasAccuracyGranularity(?granularity, \#Area) \rightarrow hasRequesterGranularity(?granularity, ?requester).$$

Applying this rule to the supermarket use case, nobody $(Requester)$ is able to know that Margaret $(Target)$ is at Position2, because she has Granularity of Area. Therefore, other users can only know that Margaret is located at ElectronicsArea $(Result)$ as she does not want to be located with a Granularity below Area (established through $hasRequesterGranularity$).

4) Closeness: Closeness policies are defined to indicate the minimum level of nearness at which persons want to be located. Nearness levels correspond to the same values defined for the granularity policies. An example of this kind of policy, applied to the supermarket scenario, is given by

$$Person(\#Margaret) \wedge Person(?requester) \wedge Closeness(?closeness) \wedge hasClosenessTarget(\#Margaret, ?closeness) \wedge hasAccuracyCloseness(?closeness, \#Floor) \rightarrow hasRequesterCloseness(?closeness, ?requester).$$

Applying this rule to the supermarket use case, if someone $(Requester)$ wants to know who is in adjacent positions, corridors, areas, or floors, he/she will not know that Margaret $(Target)$ is located at his/her Floor, as she established Floor as her maximum level of closeness to be located $(Result)$ through the $hasRequesterCloseness$ property.

V. SeCoMan Queries

This section presents a set of queries allowing smart applications to provide the space and context information to their customers. Customers will be able to obtain such information, but they cannot define their own queries in order to avoid that they gain private information from others. Queries consider the information shaped in the SeCoMan ontologies described in Section III-B and the policies defined in Section IV.

To define the space queries in SeCoMan, we used the four categories of the taxonomy defined in [15], namely: *position* or *location*, *range*, *nearest neighbor* or *closeness*, and *navigation*. In addition, SeCoMan also provides specific information of the environment and authorization decisions about users to stay in a place through operational and authorization queries, respectively. We describe in detail in the following the six queries predefined in our framework, ending with a way of defining queries composed by the Application Administrators in order to provide advanced features to their smart application(s).

A. Operational Queries

Operational queries allow Users to get information related to them and the environment in which they are located, taking into account the operational policies defined in Section IV-A. Continuing with the supermarket example of Section III-D, we show in the following a function that provides information about the products contained in the requester's shopping cart.

```
1. productInfoList shoppingCartProducts(Person
   requester) {
2.    cart ← SupermarketOnt.hasShoppingCart(requester)
3.    productList ← SupermarketOnt.hasShoppingList(cart)
4.    for (Product product : productList) {
5.       name ← SupermarketOnt.hasId(product)
6.       amount ← SupermarketOnt.hasAmount(product)
7.       price ← SupermarketOnt.hasPrice(product)
8.       discount ← SupermarketOnt.hasDiscount(requester,
         product)
9.       price ← price * discount
10.      productInfoList.add(name, amount, price)
11.   }
12.   return productInfoList
13. }
```

We implemented some methods in an external class, called *SupermarketOnt*, in order to gather information shaped in the Supermarket ontology. The *shoppingCartProducts* function receives the requester's shopping cart and the information about its products (lines 2–7). The operational policies are then taken into account to apply discounts (line 8) for each of the products deposited in the shopping cart.

Applying this query to the supermarket use case, if Peter (*Requester*) wants to know the information about the products contained in his cart, he will get that it holds a Laptop whose price is 790€ (*Result*). Although the price of the Laptop is 1000€ (see Section III-D), Peter has a 21% off when applying the operational policy defined in Section IV-A.

B. Authorization Queries

Queries related to authorization allow Users and the Framework Administrator to know if the customers have authorization or not to stay in a given space. SeCoMan offers the *authorizationAccess* and *unauthorizedPersons* functions to obtain authorization information.

The *authorizationAccess* function, defined in the following, provides Users with information about the authorization to stay in their position. It receives as parameter the *Person* who requests the information about himself/herself and returns an authorization response (allowed or denied). Note that this and subsequent functions will make use of some methods implemented in an external class called *LocationOnt*, which provides information shaped in the Location ontology. The authorizationAccess function obtains the requester's space (line 2) and will return *allowed* or *denied* depending on whether the requester has access to stay in his/her current space (see Section IV-B).

```
1. authorizationResponse authorizationAccess(Person
   requester) {
2.    space ← LocationOnt.hasPosition(requester)
3.    authorization ← LocationOnt.hasAuthzAccess
      (requester, space)
4.    if (authorization == true)
5.       return allowed
6.    return denied
7. }
```

Considering the supermarket use case, if Natalie (*Requester*) asks about her authorization to stay where she is, at PaymentArea, the response will be *allowed*. This is due to the authorization policy, defined in Section IV-B, allowing Natalie to stay there as she has the OpenCashRegister privilege.

As opposed to the previous query, the *unauthorizedPersons* function, defined below, provides the list of persons without authorization to stay in a given space. The goal behind this function is to avoid that persons from hiding their position when they are in unauthorized spaces, not having into account the policies that defined them. The unauthorizedPersons function receives as parameter the *Space* in which the requester is interested and returns the list of unauthorized persons staying there.

```
1. unauthorizedPersonList unauthorizedPersons(Space
   space) {
2.    personList ← getElements(space, "Person")
3.    unauthorizedPersonList ← emptyList
4.    for (Person target : personList)
5.       if (!LocationOnt.hasAuthzAccess(target, space))
6.          unauthorizedPersonList.add(target)
7.    return unauthorizedPersonList
8. }
```

Considering the supermarket use case, if the Application Administrator (*Requester*) asks about the list of unauthorized persons located at PaymentCorridor, he/she will obtain that Andy does not have authorization because he does not have the OpenCashRegister privilege. Natalie does not appear in that list because she does have such a privilege.

C. Location Queries

In SeCoMan, location queries can be used by Users to get the elements' position that form part of the environment, taking into account the policies defined in Section IV-C.

As an example, we defined the *elementLocation* function (given in the following), which returns a list of spaces in accordance with the two parameters received: the *Person* who requests the information, and the *Element* about which the requester is interested to obtain its position. It first checks the type of element of the target (line 2). If the target is a Person (line 3), the function checks whether he/she has a hiding, cloaking, or granularity policy with the requester. Hiding policies will return an empty list (lines 4 and 5), whereas the maximum accuracy at which the target wants to be located will be established by invoking the *getPositionsApplyingGranularity* function (lines 7 and 8). Instead, if there is a cloaking policy, the *getPositions ApplyingCloaking* function will return the real position of the target, as well as some fictitious positions to mask the former one (lines 10 and 11). In case the target is an object, no policy is applied (lines 14–16).

```
1. spaceList elementLocation(Person requester, Element
   target) {
2.    switch (LocationOnt.elementType(target)) {
3.      case "Person":
4.        if (LocationOnt.hasHiding(target, requester))
5.          spaceList ← emptyList
6.        else {
7.          if (LocationOnt.hasGranularityTarget(target,
            requester))
8.            spaceList ← getPositionsApplying
              Granularity(target, requester)
9.          else spaceList ← LocationOnt.hasPosition(target)
10.           if (LocationOnt.hasCloaking(target, requester))
11.             spaceList ← getPositionsApplying
                Cloaking(spaceList, target)
12.         }
13.       break
14.     case "Object":
15.       spaceList ← LocationOnt.hasPosition(target)
16.       break
17.   }
18.   return spaceList
19. }
```

Applying this query to the supermarket use case, if Peter (*Requester*) wants to know where Andy (*Target*) is, Peter will obtain that Andy has two locations, Position4 and Position3. This result is due to Andy having a cloaking policy, as defined in Section IV-C1, that returns Position3 as fake position.

D. Range Queries

Range queries can be used to identify all elements placed at a location meeting a certain criteria. As the previous one, these also consider hiding, cloaking, and granularity policies.

As an example, we defined the *rangeSpaceElements* function (given in the following), which provides the elements placed at a given space. This function returns a list of elements in accordance with the three parameters received: the *Person* who requests the information, the *Space* in which the requester is interested, and the *ElementType* that the requester wants to obtain. This function checks if the type of element is a Person (line 3), and if so, the function considers his/her policies; otherwise, the elements contained in the space are returned without applying any policy (lines 10–12).

```
1. rangeElementList rangeSpaceElements(Person
   requester, Space space, ElementType elementType) {
2.   switch (elementType) {
3.     case "Person":
4.       rangeElementList ← emptyList
5.       elementList ← getElements(space, elementType)
6.       for (Person target : elementList)
7.         if (!LocationOnt.hasHiding(target, requester) &&
           !LocationOnt.hasCloaking(target, requester) &&
           (getGranularity(target, requester) <= space))
8.           rangeElementList.add(target)
9.       break
10.    case "Object":
11.      rangeElementList ← getElements(space,
         elementType)
12.      break
13.  }
14.  return rangeElementList
15. }
```

Considering this query in the supermarket use case, if Andy (*Requester*) wants to know the Persons (elementType) located at ComputersCorridor (*Space*), he will obtain that nobody is located there. This result is due to Peter defining a hiding policy (see Section IV-C2) and Margaret having a granularity policy with a Granularity more than Area (see Section IV-C3).

E. Closeness Queries

Closeness queries can be used to find the nearby elements to Persons with a given level of proximity. The *hasAdjacent*, *hasPosition*, and *isPositionOf* properties, defined in the Location ontology [see Fig. 2(a)], aim to provide neighborhood and hierarchical relationships. This kind of query takes into account the policies defined by the target.

As an example, we defined the *closeElements* function (given in the following), which returns a list of nearby elements to the requester in accordance with the three parameters received: the *Person* who performs the query, the *Accuracy* indicating the proximity level at which the requester wants to get the elements, and the *ElementType* that the requester wants to obtain. This function invokes the *getAdjacentSpaces* function to retrieve the adjacent spaces to the requester's location (line 3). Then, if the type of the desired element is a Person (line 5), the function obtains the persons located at the spaces previously obtained (line 6) and applies their policies. Otherwise, if it is an object (line 11), the elements close to the requester are obtained without considering any policy (lines 11–13).

```
1. closeElementList closeElements(Person requester,
   Accuracy accuracy, ElementType elementType) {
2.   closeElementList ← emptyList
3.   spaceList ← getAdjacentSpaces(requester, accuracy)
4.   switch (elementType) {
5.     case "Person":
6.       elementList ← getElements(spaceList, elementType)
7.       for (Person target : elementList)
8.         if (!LocationOnt.hasHiding(target, requester) &&
           !LocationOnt.hasCloaking(target, requester) &&
           (getCloseness(target, requester) <= accuracy))
9.           closeElementList.add(target)
10.      break
11.    case "Object":
12.      closeElementList ← getElements(spaceList,
         elementType)
13.      break
14.  }
15.  return closeElementList
16. }
```

Considering this query in the supermarket use case, if Natalie (*Requester*) wants to know who is close to her with proximity of Corridor, she will obtain that Andy, located at PaymentCorridor, and Peter, located at ComputersCorridor, are close to her. Instead, Margaret does not appear in that list because she defined a closeness policy with a Closeness level of Floor, as defined in Section IV-C.4.

F. Navigation Queries

Navigation queries allow Users to find the path leading to the desired place or element. If the destination is a Person, his/her privacy policies are taken into account to get the path. This kind of query also considers the same policies as the ones required by the location queries (defined in Section V-C).

As an example, we defined the *getMinimumPaths* function (given in the following), which provides the list of spaces to reach the target from the requester's location in accordance with the two parameters received: the *Person* who wants to go from his/her current position to the destination position, and the *Element* indicating the destination of the path. This function obtains the spaces of the source and the destination invoking the *elementLocation* function (lines 2 and 3), as defined in the location queries of Section V-C. Once having the spaces, the function invokes the *pathFinder* function for each destination (line 6). pathFinder is a recursive function that checks if the source and the destination are in the same space. If so,

pathFinder will return the response; otherwise, it recursively calls itself using each adjacent space to the source as the next unvisited source, keeping track of paths to avoid cycles. Finally, the path is returned to the user (line 7), if any.

```
1. pathList getMinimumPaths(Person requester, Element
   destination) {
2.    sourcePositionList ← elementLocation(requester,
      requester)
3.    destinationPositionList ← elementLocation(requester,
      destination)
4.    pathList ← emptList
5.    for (Position destinationPosition : destinationPosition
      List)
6.       pathList.add(pathFinder(sourcePositionList[0],
         destinationPosition))
7.    return pathList
8. }
```

Applying this query to the supermarket use case, if Peter (*Requester*) wants to know the path to go from his current position (Position1) to the Andy's position (Position4), getMinimumPaths will return to him a list of spaces with two alternatives as Andy has a cloaking policy (where Position3 is a fake position generated in Section IV-C1): $\langle Position1, Position2, Position3, Position4\rangle$ and $\langle Position1, Position2, Position3\rangle$. Another example is the case when Andy wants to know the path to go from his current position (Position4) to the Margaret's position (Position2). The response will be $\langle PaymentArea, ElectronicsArea\rangle$, as she holds a Granularity of Area (defined in Section IV-C3).

G. Composed Queries

Application Administrators can define more sophisticated queries by combining some of those described earlier and by subsequently filtering their output. Therefore, the output of a query is the input for the next one.

As an example, we defined the *complexPathsToOffers* function (given in the following), which provides routes to products on offer nearby the requester's current location, without going through the position where a given person is located. This function returns a list of minimum paths in accordance with the three parameters received: the *Person* who requests the information, the *Space* where the requester wants to get the products with certain discounts, and another *Person* to whom the requester wants to avoid in the path. complexPathsToOffers invokes the *elementLocation* function to obtain the user's location(s) to be avoided (line 3), and then obtains the products with discounts placed at the desired space by taking into account the operational policies defined in Section IV-A (line 4). For each product, the *getMinimumPathAvoidingPosition* function is invoked (line 6) to obtain the minimum path from the requester's location to the product without going through the position(s) where the unwanted user is located.

```
1. pathList complexPathsToOffers(Requester requester,
   Space space, Person avoidPerson) {
2.    pathList ← emptyList
3.    avoidPositionList ← elementLocation(requester,
      avoidPerson)
4.    productsOnOffer ← getProductsOnOffer(requester,
      space)
5.    for (Product product : productsOnOffer)
6.       pathList.add(getMinimumPathAvoidingPosition
         (requester, avoidPositionList, product))
7.    return pathList
8. }
```

Applying this query to the supermarket use case, consider that Peter (*Requester*) wants to know the minimum path(s) to products with some discount and located at the same Floor (*Space*), without having to go through the Andy's position. The function responses that there is a Laptop at the Peter's position (Position1), and there is no possible way of going to the Tablet article without passing through the Andy's position. This is due to the cloaking policy of Andy defined in Section IV-C1.

VI. Deployment of a Context-Aware Smart Application

We developed a smart application, called *eCoMarket*, that offers advanced services in supermarkets to validate the proper functioning of SeCoMan. Furthermore, the deployment of this application was also performed for measuring the throughput and scalability of SeCoMan. These results are subsequently presented in Section VII.

The eCoMarket application provides customers (Users) with the products' location and their information, the position of shopping carts and information about their products, products on offer, nearby friends with several levels of granularity, the path to reach people or products, the customers' authorization to stay at a given place, and privileges and roles of customers. Customers of the supermarket will be able to obtain previous information using an Android application that interacts with the eCoMarket application using the REST technology. With REST, users can use devices with limited computing resources to make their requests as such devices will only have to handle queries, receive responses, and then display them. On the other hand, remote method invocation is used to separate the Application and Context Management layers, thus balancing the workload across multiple computers in order to avoid bottlenecks, among others.

Semantic rules that form policies are expressed in Semantic Web Rule Language (SWRL) [16]. SWRL includes a type of axiom, called Horn clause logic, of the form $if \ldots\ then \ldots$, and it is the most used in Semantic Web. The space and context information is shaped in the Location and Supermarket

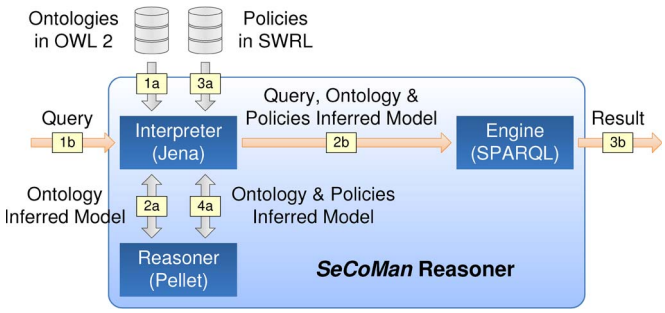

Fig. 4. Reasoning and query processes.

ontologies, respectively. Both of them are defined in Web Ontology Language (OWL 2) [17] and have been generated with the Protégé tool [18]. We have chosen OWL 2 rather than other languages, such as Resource Description Framework (RDF), RDF Schema, or DARPA Agent Markup Language+Ontology Interchange Language (DAML+OIL), because OWL 2 is more expressive than the rest. It was specifically designed as an ontology language, it is an open standard, and it is the main ontology language used nowadays in Semantic Web.

In order to infer new knowledge, all processes related to the *Reasoner* module of SeCoMan (shown in Fig. 1) are depicted in Fig. 4. The Reasoner component uses Pellet [19], which receives ontological models generated by the *Interpreter* component and returns inferred models with new knowledge. The Interpreter uses the Jena API [20] to generate ontological models with the information shaped in the ontologies and policies. Finally, the *Engine* component is in charge of translating the queries performed by the users into SPARQL queries [21], which are applied to the inferred model to get the result.

The Interpreter generates an ontological model from the Location and Context-aware ontologies (step 1a in Fig. 4). This model is sent to the Reasoner to obtain the inferred model with new information inferred from the ontologies (step 2a). Once the Interpreter receives the inferred model, it updates it with the new information provided by the Reasoner according to the policies defined in the system (steps 3a and 4a). Note that the previous process is made when new information from the environment is provided. Therefore, when queries are performed, SeCoMan always has available a consistent and updated inferred model, thus avoiding users having to wait the reasoning time shown later in Section VII-A. When users make a query (step 1b), the Interpreter invokes the *Engine* with the latest inferred model and the query (step 2b). The Engine component applies the appropriate SPARQL queries to the inferred model and returns the result (step 3b).

We developed two plug-ins in SeCoMan to obtain the space and context information. The first one obtains the space information through a REST client, which communicates with an indoor location system based on Wi-Fi. This location-based system obtains the environment map and the location of its elements, combining the fingerprinting technique with pictures about the environment [22]. The second plug-in obtains the context information through a REST client, which communicates with a radio-frequency identification (RFID) middleware [23]. This middleware is capable of getting the information of the products contained on the shopping carts. The space and context-aware information are provided to the Context Management layer by using REST.

TABLE I
INDIVIDUAL DISTRIBUTION OF POPULATION

Element	Amount	Percentage	Element	Amount	Percentage
Buildings	1	0.05%	Persons	200	8.2%
Floors	4	0.2%	Roles	10	0.6%
Areas	20	0.8%	Privileges	40	1.2%
Sections	80	3.4%	Others	90	3.55%
Positions	2,000	82%	**Total**	**2445**	**100%**

VII. EXPERIMENTAL RESULTS

We conducted some experiments with the aim of measuring the throughput and scalability of our SeCoMan proposal. These experiments were intended to deal with three questions.

1) Is the computing time of reasoning acceptable?
2) How does it scales with different amount of information, such as the number of individuals and policies?
3) How does the query time varies when taking into consideration the previous premises?

As experimental setting, the SeCoMan framework and the conducted tests were carried out in a dedicated PC with an Intel Core i7-3770 3.40-GHz, 16-GB of RAM, and an Ubuntu 12.04 LTS as its operating system. The results shown in this section have been obtained by executing the experiments 100 times and computing their arithmetic mean.

A. Reasoner Performance

The Reasoner is an important part of SeCoMan as it *greatly* affects to the framework performance. In order to check the reasoning time and its scalability, several experiments were conducted. A way to measure the SeCoMan performance is making executions with different complexity. This complexity is related to the number of statements hold in the knowledge base, which depends on the number of individuals present in the ontology and the number of semantic rules that form the policies. Increasing the number of individuals and semantic rules will provoke an increment on the number of statements and thus on the complexity of the executions.

The number of individuals contained in our ontologies is referred as *population*. This was randomly generated for the experiments, but in a controlled way, in order to achieve the desired distribution for simulating a scenario as real as possible. Table I depicts the number of elements used in our environment and the percentages obtained for them.

Another issue to evaluate the Reasoner scalability is the way in which the population sizes are established. In this sense, we defined an initial population of 15 000 individuals, and we increased this population with other 15 000 individuals in each step. In order to show the complexity of our ontology, Table II shows the relationships between the individuals and the statements generated by the Reasoner. As observed, the number of statements (obtained after the reasoning process) is proportionally increased according to the number of individuals. Each population group will be used to later obtain the time

TABLE II
NUMBER OF INDIVIDUALS AND STATEMENTS PER POPULATION

Population	0	1	2	3	4	5	6	7	8	9
Individuals	15,000	30,000	45,000	60,000	75,000	90,000	105,000	120,000	135,000	150,000
Statements	130,457	259,804	389,775	519,463	649,115	778,984	908,714	1,038,238	1,168,188	1,297,503

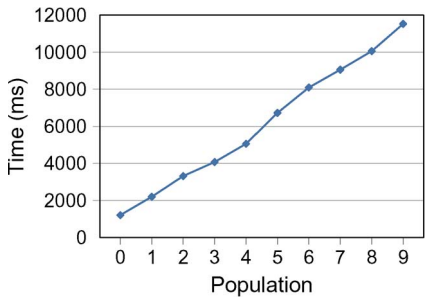

Fig. 5. Consistency checking time.

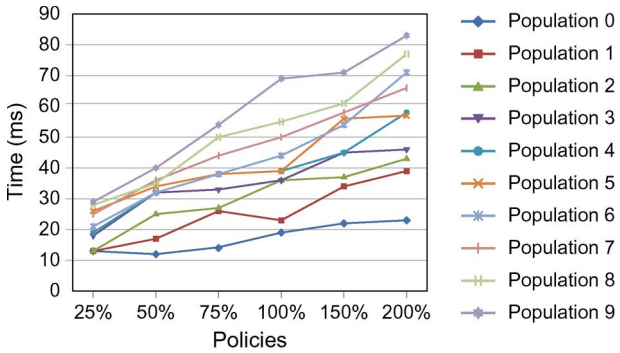

Fig. 6. Reasoning time for different populations and policies.

that SeCoMan needs to check the knowledge base consistency and infer new information.

Fig. 5 depicts the time, measured in milliseconds, used by the Reasoner to validate the ontology considering different population groups (see Table II).

Comparing the increase in individuals and statements with the reasoning time, we can observe that SeCoMan can support a very large number of individuals or statements within a reasonable reasoning time. Furthermore, the linearity property behind these results allows us to deduce that a better computer system setting would obtain a lower reasoning time.

The previous experiment has demonstrated a linear relationship between individual/statements and the reasoning time, but without considering policies. Thus, the main goal behind the next test is to check how policies can affect the reasoning time. In this sense, we defined several percentages of policies related with the persons contained in our population groups.

Fig. 6 depicts how the reasoning time varies depending on each population group (see Table II) and the percentages associated to the policies.

Policies have a very low impact in the reasoning time of our framework. For all populations, the difference between having a 25% and a 200% of policies is around a few milliseconds.

As main conclusion of this section, we have demonstrated with the previous experiments that when the number of individuals/statements is linearly increased in our ontology, the reasoning time also increases linearly. Furthermore, the semantic rules that form the policies do not have an important impact on the reasoning time.

B. Queries Performance

We want now to check how the query time varies when considering different sizes in the population and the number of policies. In this sense, we defined an experiment per each query defined in Section V: authorization, location, range, navigation, and closeness. These experiments consist on checking how the amount of individuals and the percentage of policies affect to the query response time.

Fig. 7 shows the results for each query. The x-axis corresponds to a given population group, the y-axis is the time in milliseconds, and each line symbolizes a given percentage of policies. Note that the closeness query time is not shown because this kind of query is composed of location and range queries (the response time would be the sum of both).

In order to obtain the query times for each query, we used the *unauthorizedPersons* function (defined in Section V-B) to check the authorization query time, whose results are shown in Fig. 7(a); the *elementLocation* function (defined in Section V-C) to check the location query time, whose results are shown in Fig. 7(b); the *rangeSpaceElements* function (defined in Section V-D) to check the range query time, whose results are shown in Fig. 7(c); and *getMinimumPaths* and *pathFinder* functions (defined in Section V-F) to check the navigation query time, whose results are shown in Fig. 7(d). pathFinder was implemented using the breadth-first search (BFS) algorithm. BFS is a graph search algorithm that begins with the source position and explores all the adjacent positions, examining each of the unvisited ones until finding the destination.

As shown in Fig. 7, the response time for all queries is mainly influenced by the population, as when we increased the population, the response time also increased. This is because there are more statements in the knowledge base; therefore, the complexity to answer the query is higher. Furthermore, we can observe that policies do not have a great impact in the response time as policies generate statements associated to persons, and as shown in Section VII-A, they are the 8.2% of the individuals contained in each population.

Fig. 7(b) shows that the location query time is much lower than for the rest, due to its complexity being lower. As shown in Fig. 7(d), the response time for the navigation query is much higher than for the rest of queries. We consider that its times are not an affordable time for answering a query. As we have demonstrated with the previous queries, this problem is not how to represent the information, but the complexity of the algorithm. Thus, improving the pathFinder function in order to decrease the time response is defined as future work.

As main conclusion of this section, we have demonstrated that for different kinds of queries the policies do not have a

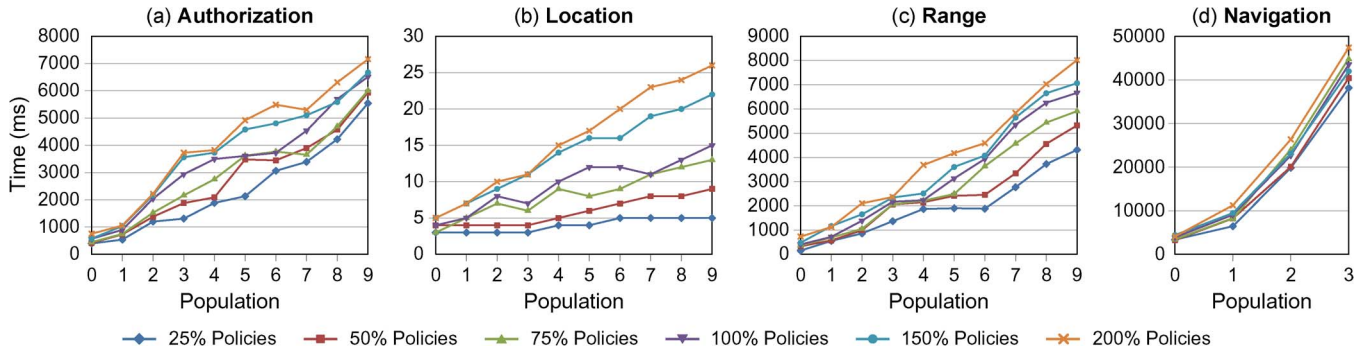

Fig. 7. Query time variation when considering different populations and policies. (a) Authorization. (b) Location. (c) Range. (d) Navigation.

significant impact in the framework performance. As it can be observed in Fig. 7, the kind of algorithm and the amount of individuals/statements are the reason for the increased time in queries and reasoning processes.

VIII. Discussion

As stated in Section II, not all context-aware systems manage policies, and those that manage policies are for different purposes, such as to protect the users' privacy. Here, we compare SeCoMan with those works presented in Section II that allow users to manage their privacy, showing in detail the policies provided by each one.

CoBrA allows users to protect their privacy through policies, indicating the personal information that they want to reveal. In SeCoMan, personal information is a topic that is beyond the scope of the Location ontology. SeCoMan preserves the privacy of the users by shaping their personal information in a context ontology whose information is managed using operational policies. On the other hand, the policies defined in CoBrA take into account the users' location and the context in which they are located, whereas SeCoMan allows actors to define richer policies than CoBrA. In our framework, users can share their location to the right users, at the right granularity, at the right place, and at the right time. Instead, the users of CoBrA cannot define policies to manage their location privacy, which is considered an important requirement in context-aware systems. SeCoMan provides four kinds of policies to allow users to manage their location privacy (see Section IV-C).

CoPS solves the inability of CoBrA to manage the users' location privacy, although the former does not consider the personal information privacy. Using policies, CoPS allows users to decide to whom and at which precision they want to share their location and context information. The policies defined in CoPS are composed of several fields: subject, context, time, precision, application, and result. The structure of these policies has certain similarities with the SeCoMan policies. However, comparing field by field, we can select in SeCoMan a given subject, or a group of subjects, depending on their roles or privileges; context and time information are included in our policies explicitly; precision corresponds in our policies to a place, or a list of places, with different granularity; and the application and result are shaped in our policies to establish an internal classification to generate new knowledge. Therefore, SeCoMan covers users' location privacy of CoPS through the hiding and granularity policies. Furthermore, our framework allows users to generate fictitious positions for specific users and manage the level of closeness at which they want to be located by other users.

TABLE III
Comparative of Systems in Managing Policies to Preserve Users' Privacy

Policies		CoBrA	CoPS	PPCS	SeCoMan
Operational		√		√	√
Authorization					√
Location	Cloaking				√
	Hiding		√	√	√
	Granularity		√	√	√
	Closeness				√

Finally, PPCS addresses the weaknesses of CoBrA and CoPS. Specifically, PPCS protects users' information and their location by allowing users to decide the granularity at which they want to share their information, to whom, and under what conditions. The time during which they want to reveal the information, or the place(s) where they want to share information, is also taken into account when users define their policies. However, and in addition to the cloaking, hiding, granularity, and closeness policies, SeCoMan grants or denies access to users to stay in certain locations depending on their privileges. In addition to that, SeCoMan also manages the context-aware information through operational policies.

Table III shows a comparison of the policies supported by the systems analyzed earlier. The rest of the proposals that were shown in Section II but were not included in Table III do not manage policies to protect users' privacy. Even solutions such as Feel@Home, Hydra, and CroCo define a module indicating that the users' privacy is protected but do not define how to develop it.

Regarding the SeCoMan performance, as we have demonstrated in Section VII, policies do not have a significant impact in reasoning and query times, allowing users to define as many policies as they want without degrading the performance. When we linearly increased the number of individuals/statements in our ontology, the reasoning time also increased linearly. These requirements should be supported by the works commented earlier. In this sense, CoBrA does not present experiments to know how these aspects affect to the system performance and scalability; CoPS demonstrates that semantic rules do not have a direct impact in the time of answering questions, as well as

showing that the query time increases linearly when the system receives simultaneous queries; and PPCS demonstrates that the reasoning time is linearly increased when users increases linearly.

Furthermore, and setting CoBrA aside for not providing performance measures, the authors of CoPS and PPCS argued that query times increase linearly as the number of users also grow (similar conclusions to ours). Yet, none of them offers users further security aspects in comparison with SeCoMan, as shown in Table III and thoroughly discussed in this section.

IX. Conclusion and Future Work

In this paper, we have shown that, to the best of our knowledge, there is no framework that accomplishes all the essential requirements to develop context-aware smart applications using a semantic-oriented IoT vision. To this end, we presented a context-aware framework called *SeCoMan* that allows developing smart applications where users can share their location to the right users, at the right granularity, at the right place, and at the right time. Queries based on location, context awareness, and authorization are predefined in the framework to provide smart applications with the space and context information. Ontologies are the key for modeling the context, inferring new knowledge through semantic reasoners, and sharing this knowledge with independent systems. Moreover, the framework functions are defined in a manner that accommodate IoT requirements, and they neither affect system performance nor introduce excessive overheads.

As next steps of this research, we plan to integrate SeCoMan in the world of cloud computing [24]. Our idea is to offer the Context Management layer of SeCoMan as middleware, located at the Platform as a Service (PaaS) layer of the cloud architecture. This layer will provide the information needed by different context-aware applications located in the Software as a Service (SaaS) layer. Furthermore, we will benefit from other advantages of cloud computing, such as elasticity, monitoring, auditing, load balancing, and security issues. We also plan to improve users' privacy by adding anonymity and hashing policies to hide and disguise the identity of a user [25].

Finally, the support and implementation of outdoor based-location services is another research topic for future work, where global positioning systems, such as GPS or Galileo, can be used to get the position of people and objects in order to offer services based on outside locations.

References

[1] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[2] N. Guarino, D. Oberle, and S. Staab, "What is an ontology?" in *Handbook on Ontologies*. Berlin, Germany: Springer-Verlag, 2009, ser International Handbooks on Information Systems, pp. 1–17.

[3] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the Internet of things: A survey," *IEEE Commun. Surveys Tuts.* [Online]. Available: http://dx.doi.org/10.1109/SURV.2013.042313.00197

[4] A. Katasonov, O. Kaykova, O. Khriyenko, S. Nikitin, and V. Terziyan, "Smart semantic middleware for the internet of things," in *Proc. 5th Int. Conf. Inf. Control, Autom. Robot.*, May 2008, pp. 169–178.

[5] B. Guo, L. Sun, and D. Zhang, "The architecture design of a cross-domain context management system," in *Proc. 8th IEEE Int. Conf. Pervasive Comput. Commun. Workshops*, Apr. 2010, pp. 499–504.

[6] A. Badii, M. Crouch, and C. Lallah, "A context-awareness framework for intelligent networked embedded systems," in *Proc. 3rd Int. Conf. Adv. Hum.-Oriented Pers. Mech., Technol. Services*, Aug. 2010, pp. 105–110.

[7] S. Pietschmann, A. Mitschick, R. Winkler, and K. Meissner, "CroCo: Ontology-based, cross-application context management," in *Proc. 3rd Int. Workshop Semantic Media Adapt. Pers.*, Dec. 2008, pp. 88–93.

[8] T. Gu, X. H. Wang, H. K. Pung, and D. Q. Zhang, "An ontology-based context model in intelligent environments," in *Proc. Commun. Netw. Distrib. Syst. Model. Simul. Conf.*, Jan. 2004, pp. 270–275.

[9] H. Chen, T. Finin, and A. Joshi, "An ontology for context-aware pervasive computing environments," *Knowl. Eng. Rev.*, vol. 18, no. 3, pp. 197–207, Sep. 2003.

[10] D. Ejigu, M. Scuturici, and L. Brunie, "CoCA: A collaborative context-aware service platform for pervasive computing," in *Proc. 4th Int. Conf. Inf. Technol.*, Apr. 2007, pp. 297–302.

[11] P. Jagtap, A. Joshi, T. Finin, and L. Zavala, "Preserving privacy in context-aware systems," in *Proc. 5th IEEE Int. Conf. Semantic Comput.*, Sep. 2011, pp. 149–153.

[12] V. Sacramento, M. Endler, and F. N. Nascimento, "A privacy service for context-aware mobile computing," in *Proc. 1st Int. Conf. Security Privacy Emerging Areas Commun. Netw.*, Sept. 2005, pp. 182–193.

[13] University of Murcia, Murcia, Spain, Complete definition of the SeCoMan ontologies. [Online]. Available: http://reclamo.inf.um.es/secoman

[14] J. M. Marín Pérez, J. Bernal Bernabé, J. M. Alcaraz Calero, F. J. Garcia Clemente, G. Martínez Pérez, and A. F. Gómez Skarmeta, "Semantic-based authorization architecture for grid," *Future Gen. Comput. Syst.*, vol. 27, no. 1, pp. 40–55, Jan. 2011.

[15] C. Becker and F. Dürr, "On location models for ubiquitous computing," *Pers. Ubiquit. Comput.*, vol. 9, no. 1, pp. 20–31, Jan. 2005.

[16] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, SWRL: A semantic web rule language combining OWL and RuleML, May 2004, W3C Member Submission.

[17] B. Motik, P. F. Patel-Schneider, and B. Parsia, OWL 2 web ontology language: Structural specification and functional-style syntax, Dec. 2012, W3C Recommendation.

[18] Stanford Center for Biomedical Informatics Research, Stanford, CA, USA, Protégé: A free, open source ontology editor and knowledge-base framework. [Online]. Available: http://protege.stanford.edu

[19] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner," *Web Semantics, Sci., Services Agents World Wide Web*, vol. 5, no. 2, pp. 51–53, Jun. 2007.

[20] The Apache Software Foundation, Forest Hill, MA, USA, The Apache Jena2 ontology API. [Online]. Available: http://jena.apache.org/documentation/ontology

[21] E. Prud'hommeaux and A. Seaborne, SPARQL query language for RDF, Jan. 2008, W3C Recommendation.

[22] A. LaMarca, Y. Chawathe, S. Consolvo, J. Hightower, I. Smith, J. Scott, T. Sohn, J. Howard, J. Hughes, F. Potter, J. Tabert, P. Powledge, G. Borriello, and B. Schilit, "Place lab: Device positioning using radio beacons in the wild," in *Proc. 3rd Int. Conf. Pervasive Comput.*, May 2005, pp. 116–133.

[23] Trascends, Glastonbury, CT, USA, Rifidi - Connect the Internet of Things. [Online]. Available: http://sourceforge.net/projects/rifidi

[24] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[25] E. Paintsil, "Evaluation of privacy and security risks analysis construct for identity management systems," *IEEE Syst. J.*, vol. 7, no. 2, pp. 189–198, Jun. 2013.

Alberto Huertas Celdrán received the M.Sc. degree in computer science from the University of Murcia, Murcia, Spain.

He is currently a Research Associate with the Department of Information and Communication Engineering, University of Murcia. His scientific interests include security, semantic technology, and policy-based context-aware systems.

Félix J. García Clemente received the M.Sc. and Ph.D. degrees in computer science from the University of Murcia, Murcia, Spain.

He is currently an Associate Professor of computer networks with the Department of Computer Engineering, University of Murcia. His research interests include security and management of distributed communication networks.

Manuel Gil Pérez received the M.Sc. degree in computer science from the University of Murcia, Murcia, Spain.

He is currently a Research Associate with the Department of Information and Communication Engineering, University of Murcia. His scientific activity is mainly devoted to security infrastructures, trust management, and intrusion detection systems.

Gregorio Martínez Pérez (M'80) received M.Sc. and Ph.D. degrees in computer science from the University of Murcia, Murcia, Spain.

He is currently an Associate Professor with the Department of Information and Communication Engineering, University of Murcia. His research interests include security and management of distributed communication networks.