Journal of Universal Computer Science, vol. 19, no. 17 (2013), 2503-2525
submitted: 15/2/13, accepted: 22/4/13, appeared: 1/11/13 © J.UCS

MECCANO: a Mobile-Enabled Configuration Framework to Coordinate and Augment Networks of Smart Objects

Ana M. Bernardos, Luca Bergesio, Josué Iglesias, José R. Casar
(Universidad Politécnica de Madrid
ETSI Telecomunicación, Madrid, Spain
{abernardos, luca.bergesio, josue, jramon}@grpss.ssr.upm.es)

Abstract: In this paper, we exploit the capabilities of mobile devices as instruments to facilitate interaction in spaces populated with smart objects. We do this through MECCANO, a framework that supports an interaction method for a user to perform physical discovery and versatile configuration of behaviors involving a network of smart objects. Additionally, MECCANO guides the developer to easily integrate new augmented objects in the smart ecosystem. Behaviors are rule-based micro-services composed by a combination of events, conditions and actions that one or more smart objects can trigger, detect or perform. Each object owns and publishes its capabilities in a software module; this module becomes available when a user physically lies in the area of influence of the smart object. The capabilities provided by a specific object can be merged with those in other objects (including those in the user's mobile device itself) to configure a behavior involving several objects, adapted to the user's needs. On operation, the behavior is run within the mobile device, serving the device as orchestrator of the involved objects. The framework also facilitates sharing micro-services in such a way that users can act as prosumers by generating their self-made behaviors. New behaviors are associated to the classes of objects that are needed to execute them, becoming ready for other users to download. The proposed interaction method and its tools are demonstrated both from the developer's and the end-user's points of view, through practical implementations.

Keywords: Ubiquitous computing, smart objects, interaction, reasoning, mobile technologies, recommendation, user generated services
Categories: H.1.2, H.5.1

1 Introduction

The issue of how to make a space 'smart' is shifting from the problem of how to coordinate sensors or other elements in the space to how to provide a consistent interaction between the space and the final user [Dahl, 08]. A smart space can be described as a crowd of smart objects, which have full meaning when they are put in relation one to each other, thus the interaction problem can be formulated as the question of how to make easier the coordination and customization of the available network of objects for a given user.

The term 'smart object' is used to designate a wide variety of physical entities, equipped with sensing, processing and, sometimes, communication and interfacing capabilities. The intelligence of these objects is heterogeneous: there are objects that are considered 'smart' just for having an NFC tag attached, while others integrate processors and memory resources that provide them with some autonomy. In brief, a smart object refers to *'a computationally augmented tangible object with an established purpose that is aware of its operational situations and capable of providing supplementary services without compromising its original appearance and interaction metaphor'* [Kawsar, 08]. In this paper, we consider that a smart object preserves its original basic function as an object, but can be augmented to offer some supplementary services, both physical and virtual, when some specific operational situations occur.

The use of smartphones as instruments to interact with objects in the environment is not a new concept: existing proposals enable interaction through gestures or by touching, pointing or scanning the objects (see Section 2). These interaction ideas are feasible because smartphones are currently equipped with a wide amount of sensors (e.g. refer to [Martín et al., 13]) that may provide information about the current user status and its situation with respect to the environment. Aside from the number of possibilities that mobile sensing provides, current mobile technology has also empowered the user as a content generator [Jensen et al., 08], facilitating the collection and publishing of daily life experiences (e.g. videos, pictures or messages posted to social networks). Additionally, contents can be easily tagged with context information from both mobile embedded sensors (e.g. location) and user provided data. The framework described in this paper considers this new role of the user as a 'prosumer' [Tacken et al., 10], capable of producing, consuming and sharing not content, but micro-services [Zhao et al., 09] related to the physical spaces and objects, through a mash-up tool to both configure and publish behaviors for the smart objects in an easy way.

Thus, this work takes advantage of smartphones as sensors and potential content managers to propose *a framework that supports a novel mobile-instrumented interaction method that empowers the user to easily configure personalized behaviors by networking smart objects and to deliver these behaviors as micro-services to be consumed by other users.* The framework is called MECCANO, which stands for 'Mobile-Enabled Configuration framework to Coordinate and Augment Networks of smart Objects'. It is known that a *meccano* is a construction set of miniature parts that can fit among them and build larger mechanical models. With the same philosophy, our system facilitates fitting software parts coming from different smart objects to build a new behavior.

In brief, the interaction concept relies on smart objects that are capable of publishing their capabilities, i.e. which type of events they can detect, which conditions they are able to check and which actions they are prepared to perform. Capabilities are shown to the user through his smartphone, when in the vicinity of the objects. Users can combine these capabilities using rules that define some kind of coordinated automatic behavior (what we refer as a *meccano*), involving one or more objects. This is enabled through the MECCANO client, which manages the whole interaction cycle between the user and the objects. The client facilitates rule configuration and executes behaviors in a stand-alone offline manner. When online, it can access the MECCANO service, or to the objects themselves in a peer-to-peer mode, to retrieve new capability modules and pre-configured *meccanos*.

Through *meccano* mobile-based interaction, a user will be able to:

- Perform intuitive configuration of a network of objects through the smartphone (e.g. 'if *I turn the world globe left*, *lock the door*').
- Configure specific behaviors involving the smartphone itself, both to a) organize action-triggering in objects when an event/condition is fired from the smartphone (e.g. 'if *phone shaken*, *put the music on*') or b) manage the behavior of the mobile device depending on external events (which are generated by other smart objects). E.g. 'if *the light switches off*, *the room is silent* and *it is Monday midnight*, *configure my mobile alarm clock in working settings*'.
- Merge smart object-triggered actions with external service initiation (e.g. 'if *scanning a food packet*, *then check discount coupons and generate an alert when in the supermarket*).
- Generate new *meccanos* and share these micro-services through the framework. Any new *meccano* related to a class of objects will be available to every object belonging to this class, thus it will be exportable to other objects.

All in all, MECCANO allows to quickly deploy customized and scalable smart spaces, adapted to the preferences of each user. Additionally, the framework makes possible for the developer to easily integrate new augmented objects to be available for the public.

In this paper, we aim at describing our mobile-instrumented interaction method, the system architecture and technologies that make possible its implementation and both developer and end-user cases that illustrate the framework operation. The paper is structured as follows. Section 2 reviews the existing literature on mobile-instrumented interaction and anticipates some MECCANO features. Section 3 explains the proposed interaction method using an application scenario and from it, it derives the requirements that the framework has to provide. Section 4 describes MECCANO architecture and its implementation details for its deployment on Android clients. Section 5 describes how a developer may integrate new objects in the proposed framework, while Section 6 reviews some examples of the interaction method in a real setting. Conclusions are gathered in Section 7, together with further steps.

2 Related work

The potentiality of mobile devices to interact with the environment has been considered both in literature and in commercial applications from some years now. For example [Beigl, 99] explored how to use traditional mobile messaging to configure an scenario in which some objects were capable of sending SMS with simple commands to facilitate their remote control and operation from a mobile device and [Want et al., 99] show how to benefit from inexpensively RFID tagged objects through portable computers.

From the commercial point of view, there are now a number of applications designed to act as control centers for 'smart homes'. Currently, it is feasible to use a smartphone to control the house lights through the wireless-equipped LIFX light bulbs, or make legacy infrastructures governable with Belkin's presence-equipped WeMo switches. In-house climate is mobile controllable through e.g. the Nest Learning Thermostat, that learns about user's habits. SmartHome offers a wide set of

ZWave devices, which make possible for the user to control any kind of pluggable device, but also doors and windows, and TV-media sets from iPhone. This TV-media remote control application is probably one of the most popular, as demonstrated by the existing offer in different mobile marketplaces. There are also different solutions to control camera-based infrastructures from mobile devices for security (e.g. iZon Remote Room Monitor, iBaby). In the gaming world, it is easy to find many examples of mobile-governed 'smart objects': e.g. Lego Mindstorms EV3 robots or AR.Drones are ready to be controlled through iOS. Another scenario in which mobile devices are also commercially linked to smart objects is health: the current offer of health sensors (weight scales, pulse meters, oxymeters, etc.) includes mobile applications to manage the sensor and capture its data (e.g. Withings devices or Fitbit activity sensor).

All these solutions provide application-based interfaces to support a set of limited actions on smart objects, making possible for the user to switch the devices on and off or configure a set of alerts. In some occasions, the user has to be connected to the same wireless network than the smart object, but in general objects the services are thought to enable external remote control.

Apart from these commercial proposals, a number of research works have addressed the development of system architectures to make possible the interaction between mobile devices and objects. Some works focus on taking advantage of mobile sensors to design physical interaction methods between the device and the object, e.g. using touching, scanning or pointing solutions [Rukzio et al., 07]. For example, [Pohjanheimo et al., 05] propose a system to implement 'TouchMe', a concept that allows accessing virtual services (such as e.g. virtually searching for a book at the library) by touching the involved objects (e.g. a book) with a RFID reader/scanner attached to the mobile device. [Lampe et al., 06] use a similar approach and equips a mobile device with a Bluetooth RFID reader to enhance the gaming experience in the 'Augmented Knight Castle', in which the mobile device serves to interact with the physical characters on the board. In the same line, [Kawsar et al., 08] address the design of a mobile framework, which facilitates interaction with NFC tagged physical objects to access web services. [Hardy and Rukzio, 08] propose an architecture that also uses NFC to interact with displays, in order to overcome the screen limitations of mobile devices and establish a bidirectional channel between displays and devices through actions such as 'select & pick' or 'select & drop'. 'Touch & Compose' is a platform [Sánchez et al., 09] that supports a model for interaction with the smart environment based on assembling applications from resources (devices, services, files, etc.) that a user is able to manually select by touching them with his mobile device; RFID-tagged icons represent resources. The EMI^{2}lets multi-agent architecture [de Ipiña et al., 06] is designed to control objects through basic actions, from a mobile device or a web service. The architecture is demonstrated with objects tagged with visual markers. The potential of web technologies for smart object orchestration is show in [Pintus et al., 2010]. Authors implement things as Web Services (using WSDL) and logical connection between things are modeled as Web Service orchestrations (using WS-BPEL). Through NFC technology and a graphical interface to 'point-click-compose', authors demonstrate how the user can combine objects to perform actions.

In [Biegl, 99], a stand-alone 'remote control' is used to get the control information from other devices in order to allow operational interaction through a simple user interface. In a similar way, [Broll et al., 09] propose to use a laser-equipped device to retrieve a set of control commands from an object: when the object detects the laser beam, it sends the control description to the master device by using infrared. [Raskar et al., 04] does not use a mobile device, but a hand held projector to point at passive RFID tags equipped with an additional photo-sensor to perform geometric operations (such as 3D location) and navigate or update information related to tagged objects. Other proposals suggest using augmented reality (AR) to interact with existing physical objects, by touching virtual information balloons or objects that are represented over the real image (e.g. like in well-known AR browsers, such as Layar or Junaio). In this direction, [Iglesias et al., 12] describe a system based on an AR application for a tablet PC that delivers information about selected objects in the environment but also facilitates simple control by superimposing virtual menus over the objects (e.g. switch a light control on/off or select a content to play on the TV); this work considers that camera focus persistence may be a way to select an object in the smart space to trigger interaction. Augmented reality is also the basis for the 'grab-carry-release' interaction concept [Cheng et al., 11]. Through different gestures, users are able to virtualize and move a real object. Gesture-based interaction with smart objects is also in Gestures-Connect [Pering et al., 07], a system that uses both NFC and acceleration-based gesture recognition to make selection and action on a object: i.e. a user can capture the information about the playing in a stereo system by scanning a NFC tag on it and flicking the wrist to the left. In order to contextualize interaction with smart objects, 'awareness marks' bundle usage information in NFC tags [Hervás et al., 2011].

Many of the described proposals are simply designed to facilitate the access to information or services, not to control the objects themselves. In those cases that include some control mechanism, control options are to manage a single object. MECCANO goes a step further, making it possible to easily customize new behaviors by networking the objects in the space. Behaviors are built on well-defined logic parts that are published directly by the objects. Moreover, with respect to object discovery, most existing frameworks are dependent on a specific technology, in contrast to MECCANO, which may be integrated with any kind of discovery technology, just depending on the final service (NFC-based touching, BT/ZigBee proximity discovery, inertial-based gestures, etc.). The same occurs with the level of intelligence of the objects to integrate: whereas other approaches focus on a specific kind of smart object, MECCANO allows integration of diverse types of objects. Additionally, none of the reviewed frameworks manage the concept of relating user-generated services to smart spaces. Another feature that makes MECCANO different is that most of the proposals rely on a server infrastructure, while MECCANO is mobile-centric. Although the client is connected to an external service (MECCANO service), it can work in a stand-alone manner, as behaviors are orchestrated from the smartphone.

In the next Sections, these singularities of MECCANO are motivated and its implementation, detailed.

3 The interaction method

Let us consider the following scenario: *'Patrick is a 70-years old man, fond of gadgets and technology. He has installed the MECCANO client in his smartphone to* *interact with different already existing smart objects in his house. For his birthday, his daughter has given to him a smart lamp and a pair of smart power clips; each of these can be attached to a power supply in order to detect if the plugged artifact is on or off. When Patrick touches the lamp with his mobile phone for the first time, he receives in his phone a new module describing the lamp capabilities through MECCANO application. It seems that his new lamp is able to 1) detect when it is switched on/off (event), 2) check if the light level is below a threshold (condition), 3) switch itself on/off (action), 4) rise or lower the light level (action) and 5) change the light color (action). Patrick is interested in remotely managing the lamp, so he decides to download the module. As he is starting to suffer from hearing loses, he thinks that it can be useful to configure some light signs to announce specific events, such as the ring of the bell or the telephone. Thus he attaches a power clip to the doorbell cable and another one to the telephone's cable. He touches one of the clips and downloads its available capabilities, which are 1) detecting that clip X fires on/off (event), 2) check whether clip X is on/off (condition), 3) check clip X state (action). Patrick then configures the following grammar: 'if the doorbell clip (A) fires on or the telephone clip (B) fires on, switch the lamp light on to blue'. He activates his newly created meccano to put it to work. Once he has tried the behavior, he thinks that this solution can be useful to somebody else. Thus he signs and shares his new meccano, uploading it to the service. When he touches for the second time the lamp with his mobile phone, he receives a recommendation of a new meccano to install, which says 'if lamp B switches on, then make lamp A blink twice'. He smiles and thinks about buying another lamp for his daughter; when connecting both objects, he will be able to know when she arrives home after work and exchange a light greeting.'*

This short scene shows the key aspects of the proposed interaction method. The whole interaction relies on a smartphone application, which serves as an intermediary between the user and those objects in his/her vicinity. The application facilitates the configuration of networked behaviors (two power clips and a lamp, in this case), but also the real time orchestration of a community of those. Objects' capabilities are usually stored into a dedicated infrastructure, either in a cloud service or in a specific server. Behaviors may be bundled as micro-services, ready to be shared in the system.

This concept is to be applicable in very different scenarios, supporting easy personalization of diverse spaces through the objects in them. For example, it can serve to adapt a room in a hotel for a frequent traveller and to afterwards export its configuration, but also to provide a customized experience in a museum or to take advantage of collective knowledge to command a greenhouse.

The stages to build such an application-agnostic implementation are shown in Figure 1. Entities in Figure 1 are decisions (diamond shapes), cognitive processes (in a rounded box) and actions (square box) [Queen, 06]. The main stages are the following:

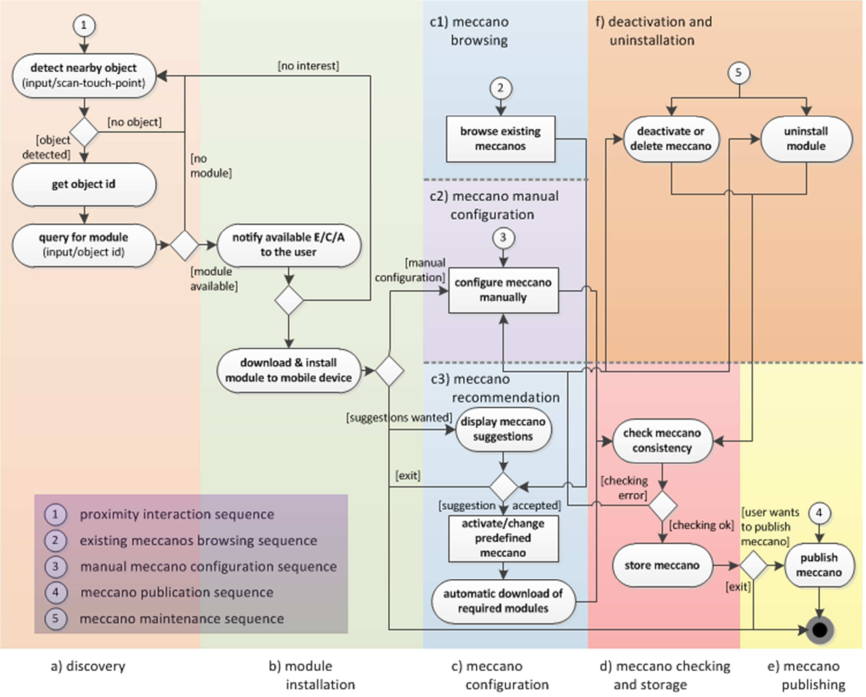

c1) meccano browsing
f) deactivation and uninstallation
1
detect nearby object (input/scan-touch-point)
[no interest]
[no object]
[object detected]
get object id
[no module]
query for module (input/object id)
notify available E/C/A to the user
[module available]
download & install module to mobile device
2
browse existing meccanos
c2) meccano manual configuration
3
[manual configuration]
configure meccano manually
c3) meccano recommendation
[suggestions wanted]
display meccano suggestions
[exit]
[suggestion accepted]
activate/change predefined meccano
automatic download of required modules
5
deactivate or delete meccano
uninstall module
check meccano consistency
[checking error]
[checking ok]
store meccano
[user wants to publish meccano]
4
publish meccano
[exit]
1 proximity interaction sequence
2 existing meccanos browsing sequence
3 manual meccano configuration sequence
4 meccano publication sequence
5 meccano maintenance sequence
a) discovery
b) module installation
c) meccano configuration
d) meccano checking and storage
e) meccano publishing

Figure 1: MECCANO Interaction Method

a) *Discovery*. The interaction begins when the user enters the influence area of a smart object. In Patrick's scenario, the object defines its influence area through NFC technology, thus touching the lamp with the smartphone is needed to start the interaction. Nevertheless, depending on the application, different ranges for proximity detection technologies can be used to define a smart object's influence area.

b) *Module installation*. The touching action triggers the retrieval of the object's univocal offering that represents the augmented capabilities of the lamp (five, in this case). As previously said, capabilities may assist on: 1) event detection (e.g. an object can detect when it is open or closed), 2) condition checking (e.g. an object can offer a logic to check if the temperature is in a given range) and 3) action execution (e.g. an object can be switched on or off). In this scenario, capabilities are stored in an external service, thus once the user agrees, the download of the software module starts and it is automatically installed in the smartphone, making the capabilities ready to use.

c) *Meccano configuration*. With the downloaded capabilities, it is possible to build a behavior or *meccano*. Capabilities from an object can be combined among themselves and with others provided by other objects through event-condition-action (ECA) rules. The smartphone is also considered a particular kind of smart object, with a set of preloaded components that can be used in *meccano* configuration. Once the *meccano* is activated, it coordinates the response of the network of smart objects on execution in the mobile device. As it is explained in Patrick's scenario, not only capabilities, but also pre-configured *meccanos* can be downloaded from an object directly browsing existing *meccanos* or accepting proposed recommendations. The recommender is a functional element in the infrastructure that includes simple logic (based on most downloaded *meccanos* and location).

d) *Checking and storage.* On creation, each *meccano* grammar is checked to verify that it is consistent prior to being locally stored in the smartphone. On activation, execution compatibility of *meccanos* is also checked in order not to perform contradictory or incompatible behaviors. This can be done even if the smartphone is not connected to the external server.
e) *Publication.* The user may decide to share his self-made behaviors. To do so, he must sign his creation and upload it to the service hosting *meccanos*. There, the new *meccano* will be checked and linked to one or more objects to make it available to other users on interaction with those specific objects.
f) *Deactivation and uninstallation.* The user can always deactivate or uninstall a *meccano* or an object module, but these ones can also be deactivated due to other requirements: e.g. in case that they do not make sense if the user is not close to the given object or due to service restrictions.

The interaction method is to be implemented through a mobile-centric architecture and a set of tools that will be supported by a service (in a dedicated server or hosted in the cloud) and by an object-embedded logic. This architecture will have to address the requirements that have been indirectly mentioned up to now, which are summarized in Table 1.

4 MECCANO architecture

This Section details the architecture that makes possible the implementation of the interaction method (Figure 2) and its specific requirements. In brief, the architecture and its description are organized in three separated units: 1) *the client*, ready to run in a smartphone; 2) *the service*, which hosts and coordinates the components/*meccanos* offering; and 3) *the smart objects*, which have a variable logic depending on their level of intelligence.

4.1 MECCANO Client

MECCANO architecture is device-centric, as it is based on a smartphone application that controls the full interaction flow: object discovery, capacity modules download and *meccano* configuration, execution, publishing and uninstallation. Once the capabilities of an object have been downloaded, the client can work in a stand-alone manner, even if the smartphone is not connected to the MECCANO service.

	Requirements	Description
R1	Easy integration of new objects	To include a new smart object in the ecosystem, its capability module has to be developed by following defined directives and a naming structure. The module can be hosted in the smart object itself (if it has storage capabilities) or in a central repository. The mobile device has to be able to retrieve it by a univocal identifier.
R2	Usable interface for micro-service configuration	A mobile mash-up tool has to enable the user to easily configure *meccanos*.
R3	Coordination of networks of objects	Actions involving a given smart object can be linked to events/conditions/actions available in other objects in the space. The tool may enable easy configuration of networked behaviors.
R4	User-empowered *meccano* generation	The framework has to empower the user to create and publish new *meccanos*. New *meccano* needs to be tagged as members of a smart object class. User-configured *meccanos* are to be stored in an external service.
R5	Publication of object capabilities	A smart object can publish a set of event, condition and action capabilities as a bundled module.
R6	Smartphone-object proximity discovery	The starting point for the discovery of capabilities of a smart object is physically being in its area of influence. When a smartphone enters a smart object's area of influence, it automatically receives the module/*meccanos* offering related to the object. Proximity may also be used to allow *meccanos* (de)activation or execution, depending on the service needs.
R7	Recommendation of a customized offering	The offering of *meccanos* may grow dramatically when users start generating *meccanos*, thus a tool to integrate recommendation algorithms for *meccanos* has to be included in the framework.
R8	Automatic download of *meccano* modules	When a user decides to download a *meccano* related to an object, he will be informed about the additional modules that are needed and not enabled in the smartphone. If the user continues with the *meccano* download, missing components will also be installed.
R9	Embedded rule-based reasoning	An action will be triggered when an event occurs and 'conditions' are fulfilled. The smartphone will be in charge of executing the active *meccanos*, thus checking the rules real-time, even if it is not connected to the Internet.
R10	Module life-cycle management	The framework has to offer mechanisms for automatic and dynamic module download, installation and uninstallation. User feedback and relative location may be used to control the lifecycle: e.g. downloaded modules can be deleted if the user has not configured them and exits the object's area of influence.
R11	Consistency checking	Consistency checking has to be done: a) on configuration, to avoid *meccanos* that do not make sense, b) on activation, to check that a *meccano* is compatible with other active ones, c) on publication, to check that the *meccano* is well-formed and ready to share.
R12	Portability	The framework has to provide tools for object classification to allow identifying and grouping similar smart objects, thus making both modules and *meccanos* portables among entities of the same class.

Table 1: List of requirements to consider in MECCANO architecture

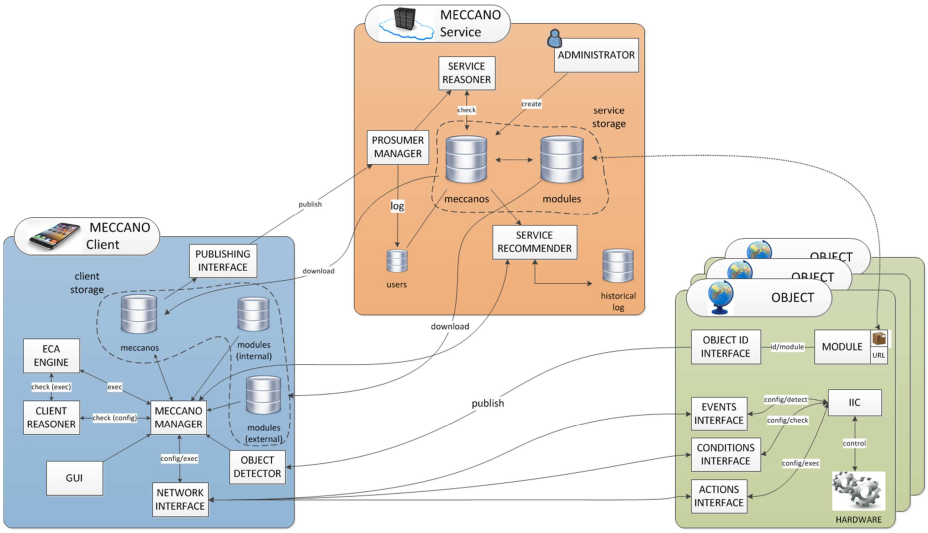

MECCANO Service
ADMINISTRATOR
SERVICE REASONER
check
create
service storage
PROSUMER MANAGER
meccanos
modules
log
users
SERVICE RECOMMENDER
historical log
publish
download
MECCANO Client
PUBLISHING INTERFACE
client storage
meccanos
modules (internal)
ECA ENGINE
check (exec)
exec
CLIENT REASONER
check (config)
MECCANO MANAGER
modules (external)
config/exec
OBJECT DETECTOR
GUI
NETWORK INTERFACE
download
publish
OBJECT
OBJECT
OBJECT
OBJECT ID INTERFACE
id/module
MODULE
URL
EVENTS INTERFACE
config/detect
IIC
config/check
control
CONDITIONS INTERFACE
config/exec
ACTIONS INTERFACE
HARDWARE

Figure 2: Overview of MECCANO architecture

The client, in our system, is composed by seven main components (see Figure 2):

- The *meccano manager*, which handles modules and *meccanos* lifecycle: it dynamically retrieves *meccanos* from the service or the objects, saves them in the *external module storage* and loads them into memory. It exchanges data among all the components within the smartphone and with the *service recommender* hosted by the infrastructure (refer to Section 4.2) in order to coordinate the download of new *meccanos* and modules.
- The *ECA engine*, which drives the *meccano* execution phase. The *meccano* manager transfers active *meccanos* from the storage component to the *engine*; it afterwards waits for the events that trigger the *meccanos* and executes them.
- During the configuration stage, the *client reasoner* checks incompatibilities, inconsistencies and possible dangerous configurations. This component does the checking in two different situations: 1) when configuring a *meccano,* while the user selects each event, condition or action, by blocking the choice of some of them and 2) when the configuration is finished, by examining the whole *meccano*. To prevent errors on operation, the reasoner checks *meccanos* when they are activated.
- The *network interface* is the component responsible for the communication with objects. During the configuration of a *meccano*, it sends orders to the objects to configure them (see Section 4.3). During the execution phase it receives events and sends and receives data related to condition checking; it also sends commands to execute actions. Internally, it communicates directly with the *meccano manager*.
- The *object detector* is the component that discovers objects using some communication technology (e.g. NFC, WiFi, Bluetooth, ZigBee). Objects may

directly publish their capability *modules* or a URL from where they can be downloaded. The object detector reads and transfers the identifiers to the *meccano manager*, which downloads the *modules*.

- The *publishing interface* is the component that allows a prosumer to publish *meccanos* as micro-services. The publishing interface controls *meccanos* signature and uploading. It is also responsible for deleting any sensitive data from a *meccano* before publishing it.
- The *Graphical User Interface* (GUI) is composed of several parts. A general structure allows the user to configure, share and download *meccanos* and to download, install and delete modules. Additionally, each module contains some elements to integrate the object capabilities, in particular a set of icons and also the interfaces that are needed to configure specific parameters for the capabilities.

The *client storage* keeps all available modules and *meccanos* in the application. The *meccano storage* stocks up the ECA rules created by the user using the smartphone and also those downloaded from the service or from the objects. Modules are stored separately depending if they are linked to the smartphone capabilities (*internal modules*) or retrieved from objects (*external modules*). Internal modules control the smartphone capabilities, either software or hardware, related to calls or messaging, sensors or applications (see Table 2 in Section 6). As part of the smartphone, these capabilities cannot be shared, but they can be manually installed and uninstalled. External modules bundle objects' capabilities, which are offered on proximity and installed and uninstalled manually and automatically.

4.2 MECCANO Service

The second part of the architecture is the MECCANO service infrastructure. In brief, MECCANO service hosts the offering of both capability *modules* and *meccanos*, to allow their publication and download. Each *meccano* may depend of an external service to control its normal execution flow and extend its functionalities; this aspect remains out-of-scope of MECCANO service.

MECCANO service can be implemented in dedicated servers or equivalent cloud elements. On abstraction, it is composed by three different entities: the *service recommender*, the *prosumer manager* and the *service reasoner*, and supported by the *server* storage, that hosts *meccanos* and *modules*.

The *service recommender* uses a simple recommendation algorithm based on popularity and user location, but it is ready to be enhanced with specific techniques that will make possible advance context-aware, profile and history-based module and *meccano* delivery.

The *prosumer manager* and the *service reasoner* are designed to provide security when uploading new *meccanos*. Since a *meccano* can be configured containing personal information (pictures, email addresses, facebook account details, etc.), a subsequent download by other users can lead to a not desirable/malicious behavior (e.g. publish unwanted photos, send emails automatically or having access to a personal facebook account). MECCANO Service offers two functionalities in order to address this security issue: *(i)* an authentication method that forces the user to digitally sign each *meccano* before being published (the *prosumer manager* checks that each *meccano* has been created by a valid existing user) and *(ii)* a *meccano* validation process held in the *server reasoner* aiming at detecting personal information to be erased and inconsistencies in the *meccano* definition. The *server reasoner*, that follows an internal architecture equivalent to the one running in the mobile device, is also planned to include intelligence to detect different levels of secure/insecure behaviors (e.g. a *meccano* controlling an irrigation system may be potentially unsecure as a it may lead to a water flood if inappropriately configured); this security level information associated to each *meccano* will be available for the users when downloading behaviors.

The *module storage* saves the software components that allow controlling smart objects' capabilities. When a new module is developed, it must be saved in this area and linked with a URL for download. In the case that the object itself is able to store and to handle a complete download, it is not necessary to copy the module in the *module storage*, but it can be done to permit a manual download from the server. The *module* and *meccano storages* are linked one to each other, as the first one hosts the components to configure each *meccano*.

Finally, it is worth mentioning that not only prosumers can create new bundles, but also an *administrator* can, directly in the server.

4.3 MECCANO 'smart object'

The logic in the smart object makes possible to discover its capabilities, to remotely configure it with specific settings and to trigger its response on *meccano* execution. This is done through four different components (see Figure 2):

- The *module* is the software that bundles the object's capability components. As explained, each module includes a set of events, conditions and actions related to the object. Depending on the communication and storing capabilities, *modules* may be stored in the objects themselves or externally, in MECCANO service.
- The *object id interface* is in charge of handling object discovery and *module* identification. Proximity to the smartphone may be detected either using a communication technology available both in the smartphone and the object (e.g. NFC, Wi-Fi, Bluetooth, 3G) or using a gateway, in case the object communication technology is not in the smartphone (e.g. ZigBee). The discovery technology and the object capacity to store the *module* define the implementation of the *object id interface.* If the *module* is stored in MECCANO service, the *object id interface* includes an URL or an object identifier that may be converted into an URL by the *meccano manager* (the identifier can be a Bluetooth MAC or a Wi-Fi network interface). If the object hosts the module itself, the *object id interface* provides the communication data to transfer the *module* through the *object detector* of MECCANO client.
- To configure the object to perform specific behaviors (e.g. to check that the temperature is above a threshold) and also to make it respond when *meccanos* are executing (e.g. switching an object on or off), each object needs to implement a set of common *interfaces* for events, conditions and actions. These *interfaces* communicate with the *intelligence & control component*, which implements object control in its native language, and with the smartphone, directly or through a gateway, through the *network interface* in the client.
- The *intelligence & control component* (ICC) implements object-dependent software to manage the object. It is a sort of driver that adapt signals coming from the mobile to signals to control the object's hardware

4.4 Some implementation details

Up to now, the general architecture enabling MECCANO interaction method has been described. In this Section, some aspects regarding our real implementation are addressed: i) the class structure for the client, together with some peculiarities of significant components, including the Graphical User Interface, ii) the data model to describe *meccanos* and *modules*, iii) some details on the server hosting MECCANO service.

The MECCANO client has been developed in Java using Android 4.1 SDK. Figure 3 shows the main classes in the code that maps the client architecture: the application classes (Figure 3.a); in Figure 3.b, the GUI classes; the class to active/deactivate *meccanos* (Figure 3.c); Figure 3.d, the communication interface, and in Figure 3.e, an example *module* coming from an object.

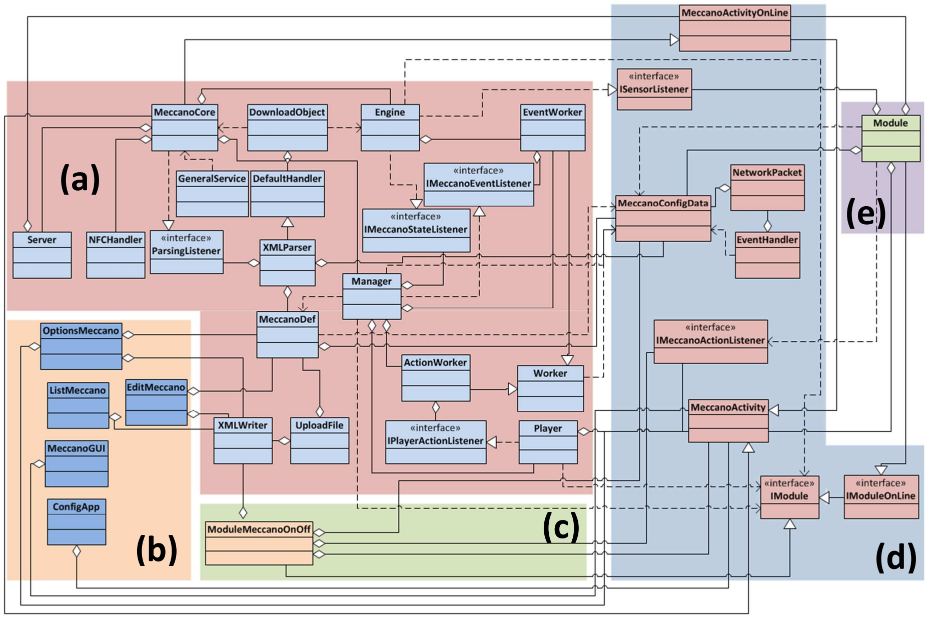

MeccanoActivityOnLine
«interface» ISensorListener
MeccanoCore
DownloadObject
Engine
EventWorker
Module
(a)
GeneralService
DefaultHandler
«interface» IMeccanoEventListener
NetworkPacket
MeccanoConfigData
(e)
«interface» IMeccanoStateListener
Server
NFCHandler
«interface» ParsingListener
XMLParser
EventHandler
Manager
MeccanoDef
OptionsMeccano
«interface» IMeccanoActionListener
ActionWorker
Worker
ListMeccano
EditMeccano
XMLWriter
UploadFile
«interface» IPlayerActionListener
Player
MeccanoActivity
MeccanoGUI
«interface» IModule
«interface» IModuleOnLine
ConfigApp
ModuleMeccanoOnOff
(c)
(b)
(d)

Figure 3: Class diagram for meccano client

Each *module* is implemented in an APK (Android Application Package) compressed file that contains Java code to implement the common interface and other resources for visualization, i.e. icons and the object GUI. The *module* publishes the object's capabilities in the *module* constructor. *Modules* are loaded at runtime using the dynamic class loader provided by Java and also available for Android. A module can share variables with other modules (e.g. the contact for the last incoming call or the smartphone Bluetooth MAC address) sending the variable to a hash table stored in the *meccano manager*.

The *communication interface* (blocks in red in Figure 3) defines the data structure for the communication between the *meccano manager* and the *modules*. Together with the *communication interface*, there are also some classes (e.g.

MeccanoConfigData class) to organize the internal structure of a *meccano*. These classes simplify coding. *iSensorLister*, *iMeccanoActionListener* and *iModuleOnLine* publish the methods to be implemented by the *modules* and used by the *meccano manager* to build and execute *meccanos*.

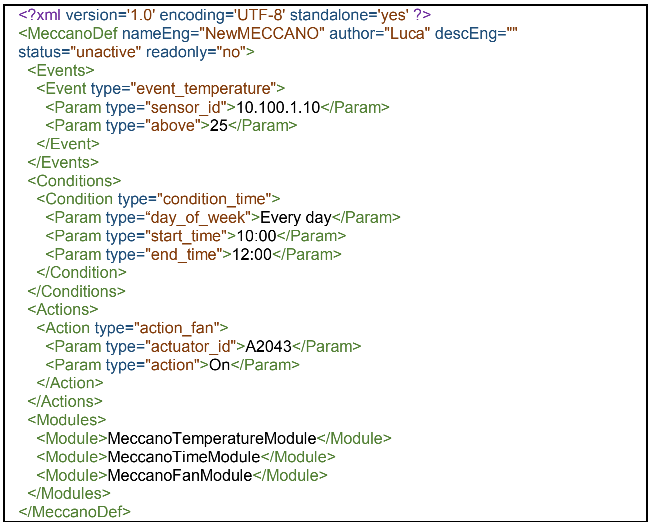
The *ECA engine* receives events from *modules* and dispatches the check of the conditions and the execution of the actions to the object *interfaces*, through the *meccano manager*. The engine uses one queue to manage events, conditions and actions in bundles and another queue to manage *meccanos*, so the behavior is always strictly sequential. In order to allow data sharing among modules, there is a hash table of objects in the *meccano manager*, where variables from the modules are stored and read. Finally the *meccano manager* includes a *server* to receive events coming from objects.

With respect to the Graphical User Interface, it enables easy configuration of new behaviors by combining icons, building a kind of ECA sentence, as it is shown in Figure 5 and Figure 6b. Yellow icons represent events; conditions are in green and actions in blue. When a module is downloaded, the user can change its configuration by tapping on the icon; then, if more than one module is available, a horizontal scroll with other modules of the same category will appear and the user will be able to select a new one. If the new module needs a specific configuration, a new screen will open in order to allow the user to set these parameters. To change the parameters, a long tap on an icon will open its parameter screen, if available; if not, a short vibration will advert the user that the capability has not any configurable parameter. To add new blocks or delete an existing one there is a '+' button on the right and a '-' button on the left: the user can simply tap on the desired button (+/-) and then tap on the icon he wants to add or delete; in the 'add' case the new capability will be added below the selected one.

For each module, the developer must deliver customized icons identifying the object's capabilities (using different colors). Additionally, if the capability can be configured (e.g. any of its parameters can be set), one or more additional screens are to be developed in order to guide the user through the configuration process (refer to Section 6 for additional details).

As it has been already mentioned, a semantic data model supports MECCANO functionalities, guaranteeing successful *meccano* composition, execution and sharing. The model enables semantic reasoning to assist the user in the construction of *meccanos* by filtering the modules configuration options according to their capabilities restrictions, to automatically detect inconsistencies in the behaviors defined in the pool of active *meccanos* for each particular user (both at inter and intra-*meccano* level) and to support a future context-based *meccano* recommendation mechanism based on both, user and *meccano* context. The model has been built as an extension of our previous work [Iglesias et al, 12], and it is implemented as an OWL2 implemented ontology, composed by four sub-ontologies (an in-depth description of this model is out of the scope of this paper). In particular, *meccano* model is implemented through an XML file. The structure of the XML file is divided into two parts: the first part includes the *meccano* author (which enables *meccano* signature when a user configures and uploads a new one) and the behavior itself, as a combination of events, conditions and actions; the second part includes a list of the modules necessary for execution. An example of XML is shown in Figure 4, it is a *meccano* that describe the implementation of modules and the interaction between a smart fan and temperature sensor: ‘When the *temperature is above 25 degrees*, if it is *between 10:00 and 12:00 AM*, then *turn the fan on*’. In the smartphone, *meccano*’s XML are stored in memory, and their pointers in a hash table.

For demonstration purposes, MECCANO service has been implemented in a specific server (Linux Ubuntu Server 8.04 LTS). The *server recommender* is implemented in an Apache web server, while databases are over MySQL. In particular, the *modules storage* database contains the reference path to download modules. In the same way, *meccanos* are stored in the form of XML files in the server. They are also listed in the *meccanos storage* database with the relative URL to provide an address to download them to the smartphone. When a user generates a new *meccano*, it is uploaded through an FTP server, that implements control access and stores the uploader’s username of who has uploaded a bundle directly in the database. The *recommender* composes the *meccano* offering for a given user depending on his location and past downloads from other users. A static table correlates objects and *meccanos* to recommend depending on nearby objects.



```xml
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<MeccanoDef nameEng="NewMECCANO" author="Luca" descEng=""
status="unactive" readonly="no">
 <Events>
  <Event type="event_temperature">
   <Param type="sensor_id">10.100.1.10</Param>
   <Param type="above">25</Param>
  </Event>
 </Events>
 <Conditions>
  <Condition type="condition_time">
   <Param type=“day_of_week">Every day</Param>
   <Param type="start_time">10:00</Param>
   <Param type="end_time">12:00</Param>
  </Condition>
 </Conditions>
 <Actions>
  <Action type="action_fan">
   <Param type="actuator_id">A2043</Param>
   <Param type="action">On</Param>
  </Action>
 </Actions>
 <Modules>
  <Module>MeccanoTemperatureModule</Module>
  <Module>MeccanoTimeModule</Module>
  <Module>MeccanoFanModule</Module>
 </Modules>
</MeccanoDef>
```

Figure 4: XML meccano syntax

5 The developer’s view: how to integrate new smart objects into MECCANO

MECCANO aims at providing a framework that makes possible to grow the number of smart objects to work with in different application scenarios. This Section is focused on demonstrating how Requirement 1 on ‘easy integration of new objects’ (Table 1) has been accomplished, by describing the procedure for a developer to integrate new smart objects into the framework. To illustrate it, two simple smart objects has been taken: a *wireless smart fan*, which publishes a capability action module to remotely control its power on/off, and an *environmental monitor*, which publishes event and condition capabilities related to the temperature and light that it is able to measure. When the procedure of integration is successfully completed, these two objects will be ready to be networked by using MECCANO client (Figure 5).

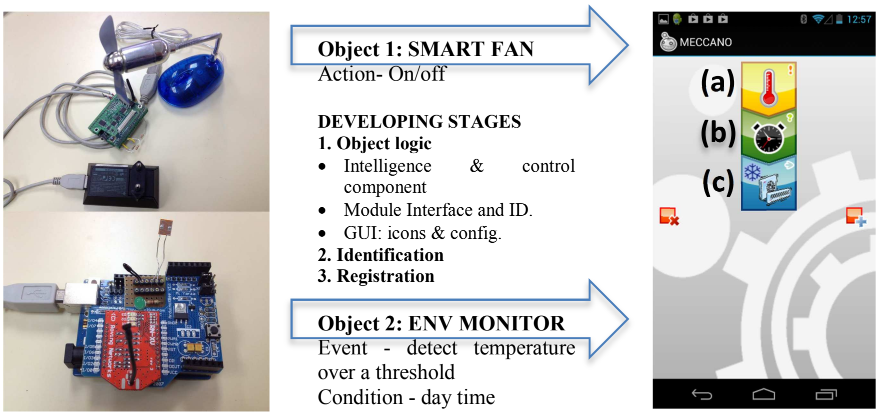

Object 1: SMART FAN
Action- On/off
DEVELOPING STAGES
1. Object logic
Intelligence & control component
Module Interface and ID.
GUI: icons & config.
2. Identification
3. Registration
Object 2: ENV MONITOR
Event - detect temperature over a threshold
Condition - day time
MECCANO
12:57
(a)
(b)
(c)

Figure 5: Smart objects and GUI for meccano configuration and activation. (a) icons are for events, (b) for conditions and (c) for actions

5.1 Stage 1 - Making an object smart

Nowadays, it is relatively easy to smartify a standard object with additional capabilities. In order to create a wireless smart fan, we have equipped an USB fan with a ZigBee communication interface, by connecting it to a MEMSIC MicaZ mote. The *control component* (ICC) that gets the orders from the object's *action interface* is implemented in TinyOS; it is programmed to receive a packet containing a switch on/off order and to convert it to a simple actuation by opening or closing the supply to the fan. As regular smartphones do not include any IEEE 802.15.4 interface, a MicaZ base station connected to a computer is used as a gateway. In the client, the object *module* must be capable of sending an order to the gateway through WiFi to switch the fan on or off.

The second smart object for this experiment is an *environmental monitor* that includes several sensors (e.g. a temperature one), which are built on an Arduino UNO equipped with an Xbee shield and a WiFly transceiver to give it a Wi-Fi communication interface. In this case, the *ICC* reads data from a port and forwards them to the mobile application. Once there, the client checks the configured conditions; no additional logic is coded within the sensor itself.

5.2 Stage 2 - Coding the object module

Once the basic control for the object has been set up, the developer needs to implement in the *module* the methods of the *communication interface* that enable communicating the mobile application and the fan (Figure 3.d). For the fan case, this code has to notify the *meccano manager* that an action has been triggered, then to build a data packet containing the action instruction (on/off) in order to send it over WiFi and to notify the end of the action.

Once the *module* logic is implemented, it is necessary to include the object network address (IP, MAC Bluetooth, IEEE 802.15.4 ID…) in the *meccano* description. In case of having multiple objects of the same type, a list of them has to be defined (i.e. if there were several fans that share the same module, it is necessary to identify which object to use). In the XML in Figure 4, the *actuator_id* includes the fan's address. Note that in this case, since the smartphone must pass through a gateway to communicate with the fan, the module must always send data to the gateway IP address and include the fan's id into the packet.

Within the *module*, the developer must also provide the GUI components that permit the integration of the object with the visual interface: the configuration screens and illuminating icons that visually describe events, conditions and actions. In this case, as the object enables a single action, the developer only has to provide an action icon (Figure 5.c) and a GUI to change the gateway IP address. For debugging purposes, the framework delivers a test class that allows executing the module as if it were a standalone application. When the *module* is finished, it must be compiled as an Android library; this process generates an APK archive.

For the second object (i.e. the temperature monitor), the *module* bundles two types of capabilities: event triggering and condition checking. The Android *module* for the object is similar to the fan's one, it includes four icons (events/conditions for temperature/light) and the GUI to configure them. Since it has a Wi-Fi transceiver, it uses IP addresses to identify the object (Figure 4, *sensor_id*). The configuration process is however different to the fan's. For example, as we want to configure the sensor to generate an event when the temperature goes over a threshold, we must configure this threshold and we also need to specify the address to send the event through the object's *event interface*. On *meccano* activation, the *module* sends the threshold to the Arduino, which stores it, together with the smartphone IP address that will be receiving the events (this is done through the *events interface*). Other conditions (i.e. conditions not depending on the object itself, e.g. those related to checking the time of the day) are evaluated in the mobile client (*ECA manager*) on *meccano* execution.

5.3 Stage 3 - Registering the module

As previously said, *modules* can be stored in MECCANO services or directly in an object. In this case, the external service is used, thus the object publishes an URL from where the smartphone can download the module. The developer must register the *module* in MECCANO service and prepare the object to publish the URL, by copying the APK file in a directory controlled by Apache and its path (as Apache external URL) into the *module storage* database. Since the fan only has an IEEE 802.15.4 interface and it is not compatible with the communication technologies available in the smartphone, an NFC tag with the object URL is used to detect vicinity. The reading triggers module download and installation in MECCANO client. For the second smart object, we also use a NFC tag to publish the URL.

Once the modules for both smart objects are downloaded, it is possible to implement the *meccano* described in Figure 5: *'when the temperature rises above 25 C, if it is between 10:00 AM and 12:00 AM, then switch the fan on'*. This *meccano* may be associated to both objects (i.e. fan and monitor), thus users can access it on object discovery.

6 Demonstrating the interaction method in a real setting

To demonstrate the proposed interaction method and its implementation from the end-user point of view (Requirements 2-4, Table 1), let us consider again Patrick's home scenario.

Our elderly's house is now populated with the following MECCANO smart objects: a lamp, an entrance door and its mat, a weight scale, several electricity plugs, an HVAC system, a television set and a fridge. Additionally, a set of Bluetooth proximity nodes (some of them embedded in specific objects) enables room-based location detection.

Table 2 shows a selection of some of the mentioned objects. Through them, we will exemplify how Patrick might use the available setting. Table 2 also includes the discovery technology for the objects, together with the capabilities published in their downloadable *modules.* In particular, it consider four objects, apart from the smartphone:

- The entrance door has been smartified with a kit that includes: a) an NFC tag to identify the user when opening/closing the door, b) a Bluetooth sensor for short-range proximity detection and c) an automation mechanism with ZigBee interface that permits opening and closing the door, checking its state and counting how many people are inside the house. To offer this last capability, the door works in coordination with a ZigBee pressure mat, located at the home landing. Thus the door is able to trigger an event when it is open or closed or when a person enters or exits the house. It is also able to check some conditions related to the door state, the identity of the person involved in the events above and the house occupancy (extracted from the number of entering/exiting events). The available actions include opening or closing the door automatically. Additionally, it is linked to an on-line weather service, thus it can offer weather conditions related to weather state checking.
- The room acts as a smart object itself. The smartphone detects its location through Bluetooth screening (as previously mentioned, rooms are to be populated with objects with capabilities of presence detection, that can be associated to specific locations). When the mobile detects the room-tagged access point, it builds an URL by using the Bluetooth MAC address from the AP and uses it to download the capabilities related to that specific space from MECCANO service. These capabilities facilitate the detection of entering/exiting events, associated or not to a person's identity, and checking the number of people in the room. The user may build location-based *meccanos* on these capabilities.
- The Bluetooth weight scale with proprietary technology cannot be programmed, thus the system uses a static URL that the mobile application's logic completes by adding an identifier coming from the object. This object can detect when a (given) user is weighting and when it was the last time that the user did it. The scale offers the possibility of sending the weight to different services.

S. O.		***Capability components***	***Example meccano's*** **offering**	***Discovery tech.***
Entrance door	E	On open/close door Enter/exit the house*	M1. 'When nearby the entrance door, if the weather forecast is rainy, open it in the browser and make phone vibrate' M2. 'On entrance door closing, if mat is on (exit event) and nobody at home, turn general HVAC system and lamp off' M3. 'On entrance door opening, if mat is on (enter event) and TV off, turn TV on in the news channel'	NFC tag

	C	If door is open/closed
		If (nobody/x people)^c at home*
		If (user identity)^c is
		If daily weather is (type)^c		

	A	Open/close door		
Room	E	Enter/exit the room	M4. 'When in the room, automatically maintain the room temperature at 25ºC' M5. 'When in the room, turn on/off the HVAC system by shaking the phone' M6. 'When in the room, on phone shaken, if between 16:00 and 16:30, set mobile in vibration mode and switch HVAC on and light off'	Bluetooth ID
	C	If (somebody/specific device)^c in the room If (room occupancy<>threshold)^c **		
	A			
Weight scale	E	On (user X)^c weighting	M7. 'When BT weight scale is detected, pair it with the smartphone' M8. 'On weighting, if last measure age>24h, send weight by email' M9. 'On weighting, if weight over 80, post the weight to the fridge's screen'	Bluetooth ID (needs pairing)
	C	If (user X's)^c (last measure age <>=threshold)^c If (user X's weight)^c <>=threshold)^c		
	A	Send (user X's)^c weight by (email/to a device)^c Store weight in a service		
Plug	E	Detect device plugging	M10. 'Notify incoming calls on TV'	USB connector
	C	If (battery level <>threshold)^c		
	A	Check battery state		
		* enabled if door mat; ** infrastructure BT scan is needed; ^{c}configurable parameters		
Smartphone	E	Receive SMS Incoming call BT devices detected Enter/Leave (GPS+radius, cell)	Alarm fired Periodic timer Device shaken/turned Start moving	Profile change NFC tag reading USB plugged/unplugged
	C	If SMS/call from a specific number/a number in a list If a specific BT MAC detected If in a tagged place	If time between two time references is <>= threshold	If device profile is If device is charging
	A	Call a number Take a photo Start/stop BT detection Pair devices	Vibrate (normal, short, twice) Open a website / GET Shift call/event to a compatible device	Change profile Play a song Publish in twitter Send an email

Table 2: Example of some smart objects that offer their capability components and associated meccanos. The smartphone is treated as a smart object containing a set of configurable modules

- For the electricity plug, the identifier is also a static value associated to the USB port (a standard electrical socket that can only transport electric energy, not signal - otherwise, a dynamic identifier could be attached). The plug may detect when a device is plugged (event), perform differently depending on the device battery load (condition) and check the battery state (action).

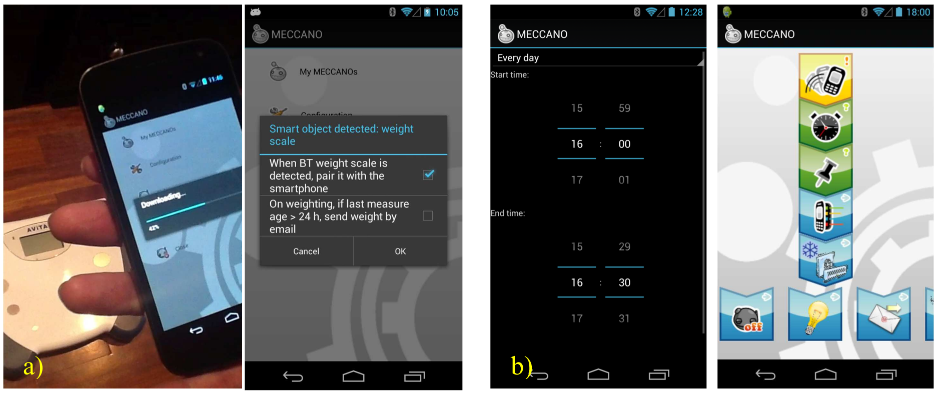
- Finally, the MECCANO application provides a number of default capabilities involving the smartphone: e.g. when the device is shaken (event), check the origin of an incoming call (condition) or take a photo (action) are some of the components ready to be merged with those from other objects.

In this ecosystem, Patrick can configure a long list of *meccanos* and also benefit from already existing ones that are offered by the MECCANO service. Next we describe some examples of configurations that Patrick may find useful to support his daily living activities, for example to access valuable information, set preferred home ambiences, avoid forgetting things or support him to maintain a healthy lifestyle.

Example 1: next to the smart door - Patrick frequently forgets his umbrella, thus he has configured a *meccano* to receive a mobile haptic alert of a rainy weather forecast when he approaches the entrance door (M1). When he arrives home and nobody else is in, he feels lonely, thus he has configured another *meccano* that switches the TV on in his preferred news channel to have some welcome background noise (M3). Additionally, he has browsed the offering of available *meccanos* and has found a new compatible one not to leave the lights and the air conditioning on when going out (M2).

Example 2: keeping an eye on weight - Some months ago, Patrick's doctor insisted on controlling his weight, thus he bought a Bluetooth wireless scale (Figure 6a). When setting it up, he downloaded a configuration *meccano* that automatically pairs his smartphone with the scale. This way, Patrick is automatically identified each time that he weights (M7). To avoid following a strict diet, Patrick has decided to set a 'persuasive' *meccano* that makes his weight visible in the fridge's door panel whereas he goes over his recommended threshold (M9). It seems that this idea works for him, so he has even shared it in the MECCANO service.

Example 3: customizing the daily environment - Other type of use that Patrick gives to *meccanos* is to configure atmospheres at home. For example, he likes to take a daily nap in his sofa, which is in the living room. Therefore he has personalized a *meccano* (Figure 6b, M6) to set his optimal conditions of temperature, noise and light during his rest slot.



Smart object detected: weight scale

When BT weight scale is detected, pair it with the smartphone ☑

On weighting, if last measure age > 24 h, send weight by email ☐

Cancel OK

Every day

Start time: 15 59 / 16 : 00 / 17 01

End time: 15 29 / 16 : 30 / 17 31

Figure 6: a) Default meccano offering for weight scale, b) configuring the time capability and nap-setting meccano

Example 4: practical stuff - Patrick is used to plug his smartphone in his bedroom to recharge it; he usually forgets where the phone is, not listening to the incoming calls anymore. Since his daughter always complains, he has configured a *meccano* that allows notifying incoming calls on an IP device as his TV set (M10).

Patrick's scenario is obviously fictitious, but MECCANO framework has been fully implemented (Figure 5 and 6 shows some real implementations) and some videos showing how capabilities/*meccanos* are offered and how *meccanos* are configured in similar real settings to the list described here are available in [GPDS, 13].

7 Conclusions

In this paper, a framework to support mobile-instrumented interaction with smart objects is presented. The framework, named MECCANO, facilitates an interaction method that is based on the composition of combined behaviors, by aggregating capabilities of smartphones, smart objects and smart spaces. The result is a tool that enables the user to create, customize, generate and publish his preferred behaviors in a simple and intuitive way and the developer to easily integrate new smart objects in the ecosystem.

The framework is supported by an architecture that makes possible the practical implementation of the interaction method's requirements. This architecture, which includes client, object and infrastructure components, relies on a data model, that guarantees the consistency of *meccano* creation in different stages of the bundle lifecycle. In particular, the data model becomes especially relevant to guarantee the quality and coherence of user-created *meccanos*.

The paper refers how a developer may integrate a new smart object in MECCANO by using the available framework. The procedure, although unavoidably requires some specific coding of drivers and objects functionalities, is well defined and facilitated by a set of programming interfaces. Object identification and registration are also system-guided. This simplifies scaling the framework capabilities with new smart objects that can be included in different application scenarios.

From the end user perspective, MECCANO is demonstrated in a possible real setting, i.e. a home equipped with several smartified objects (legacy objects that have been equipped to be smart). This scenario allows showing the feasibility of the interaction method and the diversity of possible objects to be integrated in MECCANO. Depending on the object, the technology used for smart object's discovery and query initiation might be different (NFC, USB, Bluetooth, etc.). The demonstration exercise cover both the configuration of *meccanos* from different event-condition-action components coming from different objects, but also the recommendation of preconfigured *meccanos* depending on the target smart object.

As it is commented in Section 3, the interaction method that is proposed in the work is a prescriptive one, which could be tuned after deep user evaluation. This task is in our further work, together with a metric-based study about discoverability, learnability, user efficiency and productivity, system response time and easiness of use of MECCANO. Additionally, we aim at enhancing specific features of the architecture, in particular for the recommender in the server: as user-based generation can make the MECCANO components' pool grow dramatically, we are developing a specific recommendation algorithm to give priority to those components or bundles that can be more adapted to the user needs, depending on his profile and context.

Acknowledgements

This work has been supported by the Government of Madrid under grant S2009/TIC-1485. The authors also acknowledge fruitful related discussions on the ECA paradigm on mobile devices with consultants from Deimos-Space Co., under the THOFU initiative, financed by the Spanish Center for the Development of Technology.

References

[Beigl, 99] Beigl, M.: Point & Click – Interaction in Smart Environments. *Procs. of the First Int. Symposium on Handheld and Ubiquitous Computing*, LNCS 1707, Springer-Verlag, pp. 311-313, 1999.

[Broll et al., 09] Broll, G., Paolucci, M., Wagner, M., Rukzio, E., Schmidt, A., Hubmann, H.: Perci: Pervasive Service Interaction with the Internet of Things. *IEEE Internet Computing*, vol. 13, no. 6, pp. 74-81, 2009.

[Cheng et al., 11] Cheng, K-Y., Lin, Y-H., Lin, Y-H., Chen, B-Y., Igarashi, T.: Grab-carry-release: manipulating physical objects in a real scene through a smart phone. *Proceedings of SIGGRAPH Asia 2011 Emerging Technologies* (SA '11). ACM, New York, NY, USA, Article 13, 1 page, 2011.

[Dahl, 08] Dahl, Y.: Redefining Smartness: The Smart Home as an Interactional Problem. *Proceedings of the IET 4th International Conference on Intelligent Environments*. IEEE, pp. 1-8, 2008.

[de Ipiña et al., 06] de Ipiña, D.L., Vázquez, J.I., García, D., Fernández, J., García, I., Sainz, D., Almeida, A.: EMI2lets: a reflective framework for enabling AmI. *Journal of Universal Computing Sci* (JUCS) 12(3): 297-314, 2006.

[GPDS, 13] MECCANO demonstration videos, 2013.
http://www.grpss.ssr.upm.es/index.php/es/investigacion/67-research-resources, 2013.

[Hardy and Rukzio, 08] Hardy, R., Rukzio, E.: Touch&Interact: Touch-based Interaction of Mobile Phones with Displays. *Proceedings of the 10th international conference on Human Computer Interaction with Mobile Devices and Services*. ACM, New York, NY, USA, pp. 245-254, 2008.

[Hervás et al., 11] Hervás, R., Bravo, J., Fontecha, J.: Awareness marks: adaptive services through user interactions with augmented objects. *Personal Ubiquitous Computing*, vol. 15, pp. 409-418, 2011.

[Iglesias et al., 12] Iglesias, J., Gómez, D., Bernardos, A.M., Casar, J.R.: An attitude-based reasoning strategy to enhance interaction with augmented objects. *Proceedings of the International Workshop on Extending Seamlessly to the Internet of Things*, IEEE, pp. 829 – 834, 2012.

[Iglesias et al., 12] Iglesias, J., Bernardos, A.M., Bergesio, L., Cano, J., Casar, J.R.: Towards a lightweight mobile semantic-based approach for enhancing interaction with smart objects, *Advances in Intelligent and Soft Computing*, Springer-Verlag, vol. 151, pp. 185-196, 2012.

[Jensen et al., 08] Jensen, C.S., Vicente, C.R., Wind, R.: User-Generated Content: The case of Mobile Services. *Computer,* Vol. 41, Iss. 12, pp. 116-118, 2008.

[Kawsar et al., 08] Kawsar, F., Fujinami, K., Nakajima, T.: Prottoy Middleware Platform for Smart Object Systems. *International Journal of Smart Home*, vol. 2, no. 3, 2008.

[Lampe et al., 06] Lampe, M., Hinske, S., Brockmann, S.: Mobile Device-based Interaction Patterns in Augmented Toy Environments. *Proceedings of the Third International Workshop on Pervasive Gaming Applications*, pp. 109-118, 2006.

[Martín et al., 13] Martín, H., Bernardos, A.M., Iglesias, J., Casar, J.R.: Activity logging using lightweight classification techniques in mobile devices. *Personal Ubiquitous Computing*, Vol. 17, Issue 4, pp 675-695, April 2013.

[Pering et al., 07] Pering, T., Anokwa, Y., Want, R.: Gesture Connect: Facilitating Tangible Interaction with a Flick of the Wrist. *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*, ACM, New York, NY, USA, 259-262, 2007...

[Pintus et al., 10] Pintus, A., Carboni, D., Piras, A., Giordano, A.: Connecting smart things through web services orchestrations. *Proceedings of the 10th international conference on Current trends in web engineering* (ICWE'10), Florian Daniel and Federico Michele Facca (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 431-441, 2010.

[Pohjanheimo et al., 05] Pohjanheimo, L., Keränen, H., Ailisto, H.: Implementing TouchMe Paradigm with a Mobile Phone, *Proceedings of the joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies*. ACM, New York, NY, USA, 87-92, 2005.

[Queen, 06] Queen, M.: Interaction Modeling: User State-Trace Analysis, http://boxesandarrows.com/interaction-modeling/, 2006. [retrieved on Jan 2013]

[Raskar et al., 04] Raskar, R., Beardsley, P., van Baar, J., Wang, Y., Dietz, P., Lee, J., Leigh, D., Willwacher, T.: RFIG Lamps: Interacting with a Self-Describing World via Photosensing Wireless Tags and Projectors. *ACM Journal Transactions on Graphics* (TOG) - Proceedings of ACM SIGGRAPH, Volume 23 Issue 3, pp. 406-415, August 2004.

[Rukzio et al., 07] Rukzio, E., Broll, G., Leichtenstern, K., Schmidt, A.: Mobile Interaction with the Real World: An Evaluation & Comparison of Physical Mobile Interaction Techniques. *Proceedings. of AmI 2007*, LNCS 4794, Springer-Verlag, pp. 1-18, 2007.

[Sánchez et al., 09] Sánchez, I., Riekki, J., Pyykkönen, M.: Touch&Compose: Physical User Interface for Application Composition in Smart Environments. *Proceedings of the First International Workshop on Near Field Communication*, IEEE, pp. 61-66, 2009.

[Tacken et al., 10] Tacken, J., Flake, S., Golatowski, F., Prüter, S., Rust, C., Chapko, A., Emrich, A.: Towards a Platform for User-Generated Mobile Services. *Procs. IC on Advanced Information Networking and Applications Workshop*, IEEE, pp. 532-538, 2010.

[Want et al., 99] Want, R., Fishkin, K.P., Gujar, A., Harrison, B.L.: Bridging Physical and Virtual Worlds with Electronic Tags. *Procs. of the Conf. Human Factors in Computing Systems*, ACM Press, pp. 370-377, 1999.

[Zhao et al., 09] Zhao, Z., Laga, N., Crespi, N.: A Survey of User Generated Service. *Proceedings of IC-Network Infrastructure and Digital Content.* IEEE, pp. 241-246, 2009.