MALMÖ UNIVERSITY ELECTRONIC PUBLISHING

This is an author produced version of a paper published in Software Architecture : Proceeding of 12th European Conference on Software Architecture, ECSA 2018. This paper has been peer-reviewed but does not include the final publisher proof-corrections or journal pagination.

Citation for the published paper:
Alkhabbas, Fahed; Spalazzese, Romina; Davidsson, Paul. (2018). ECo-IoT : An Architectural Approach for Realizing Emergent Configurations in the Internet of Things. Software Architecture : Proceeding of 12th European Conference on Software Architecture, ECSA 2018, p. null

URL: https://doi.org/10.1007/978-3-030-00761-4_6

Publisher: Springer

This document has been downloaded from MUEP (https://muep.mah.se) / DIVA (https://mau.diva-portal.org).

ECo-IoT: an Architectural Approach for Realizing Emergent Configurations in the Internet of Things

Fahed Alkhabbas[1,2] *, Romina Spalazzese[1,2], and Paul Davidsson[1,2]

[1] Department of Computer Science and Media Technology
[2] Internet of Things and People Research Center
Malmö University, Sweden
{fahed.alkhabbas,romina.spalazzese,paul.davidsson}@mau.se

Abstract. The rapid proliferation of the Internet of Things (IoT) is changing the way we live our everyday life and the society in general. New devices get connected to the Internet every day and, similarly, new IoT services and applications exploiting them are developed across a wide range of domains. The IoT environment typically is very dynamic, devices might suddenly become unavailable and new ones might appear. Similarly, users enter and/or leave the IoT environment while being interested in fulfilling their individual needs. These key aspects must be considered while designing and realizing IoT systems.
In this paper we propose `ECo-IoT`, an architectural approach to enable the automated formation and adaptation of Emergent Configurations (ECs) in the IoT. An EC is formed by a set of things, with their services, functionalities, and applications, to realize a user goal. ECs are adapted in response to (un)foreseen context changes e.g., changes in available things or due to changing or evolving user goals. In the paper, we describe: (i) an architecture and a process for realizing ECs; and (ii) a prototype we implemented for (iii) the validation of `ECo-IoT` through an IoT scenario that we use throughout the paper.

Keywords: Internet of Things · Emergent Configurations · Self-adaptive Systems · Software Architecture.

1 Introduction

The rapid proliferation of the Internet of Things (IoT) is changing the way we live and work. New things, (smart) connected objects and devices with their services, functionalities, and applications become available everyday. Leveraging such things, new IoT systems are continuously developed providing new types of services and applications in various fields such as home automation, transportation and health-care to mention a few [7, 12].

The context of IoT systems continuously changes as things, which are possibly resource-constrained and mobile, can join at anytime or become suddenly unavailable. The high dynamicity of the context makes it hard, if not impossible, to fully specify at design time which things constitute IoT systems, which things contribute to perform what tasks, and in which order. Therefore, to enable the IoT, it should be possible to

* Corresponding author.

automatically form IoT systems based on dynamically discovered things and adapt the systems in response to emergent user needs and unforeseen context changes. To meet the aforementioned requirements, we exploit the concept of Emergent Configurations (ECs) to engineer IoT systems. The term EC refers to a set of things that connect and cooperate temporarily to achieve a user goal. A thing is any (smart) connected object or device, with its functionalities and services or applications [2, 5].

A concrete use case that we use throughout the paper is the smart meeting room scenario presented in [2]. Imagine a person who enters an unknown smart meeting room and intends to deliver a presentation. The smart room is equipped with several things including light and temperature sensors, curtains and light actuators, a smart screen, and a smart projector. The user expresses her/his goal to deliver a presentation e.g., via an application installed on her/his smartphone. The goal is interpreted and a set of suitable things are automatically chosen to form an EC which satisfies it. For instance, the EC constituents could be the smartphone, the smart projector, the light sensor and the curtains actuator. The smartphone connects and streams the presentation to the projector which illustrates it while curtains are closed automatically due to the high light levels detected by the light sensor. Suppose that, during the presentation, the projector turns off suddenly. The failure is automatically detected and the user is proposed to continue the presentation using the available smart screen.

`ECo-IoT` consists of: (1) a process which enables the automated formation and adaptation of ECs in response to dynamic context changes; (2) an architecture which enables the realization of ECs and refines the abstract architecture presented in [2]. We also present a first prototype implementing both the architecture and the process and validate the feasibility of `ECo-IoT` through the smart meeting room scenario. In this paper, we *assume* that: (i) the user goal specification and interpretation process is already performed and correctly terminated, i.e., we take as input a decomposed goal (we plan to investigate this as future work); (ii) ECs are formed, enacted and adapted within well-defined spatial boundaries: ECs goals are achieved within locations e.g., room, building. Consequently, the number of things involved in forming and adapting an EC is not expected to be massive. This notably mitigates the well-known IoT scalability problem [12]. (iii) ECs are realized to achieve goals within non-critical time constraints: we envision that ECs are realized within the timescale of seconds; (iv) ECs are formed and enacted at runtime: ECs are realized to achieve user goals expressed at runtime; (v) ECs constituents share a context ontology.

The remainder of this paper is organized as follows. Section 2 discusses related works. Section 3 presents an overview about ECs. Section 4 introduces the `ECo-IoT` approach. Section 5 presents the prototype implementation. Section 6 presents an experiment which validates the feasibility of the approach. Finally, Section 7 concludes the paper and outlines future work directions.

2 Related Work

In the context of architectures, a number of works have been proposed. The IoT-A project presented a service-based reference model architecture for the IoT [8]. The architecture we propose in this paper is compliant with the IoT reference architecture

and refines the process management and the service organization layers presented in the reference architecture functional view. Kramer et al. [11] proposed an architectural reference model to support automatic (re)configuration of self-managed systems. The model relies on a set of predefined plans to achieve system goals. When new goals are introduced, new plans are generated in a timely consuming process. Aura is an architectural framework designed to enable users to continue their tasks in mobile contexts where they can move between different environments and to adapt progressing computations apropos the dynamic availability of services [22]. User tasks are precompiled at design time and appropriate services are (re-)assigned at runtime. Another architecture which adopts a similar approach is SIA, a service oriented architecture designed to enable the integration of the IoT in enterprise services [18]. In SIA, business processes are modelled at design time using an extended version of BPEL which allows dynamic assignment of services during the processes execution. Although services can be (re)assigned at runtime, specifying execution flows at design time limits systems flexibility as updating execution flows automatically, in response to unforeseen context changes or emergent user needs, is not supported in both Aura and SIA. Dar et al. [10] proposed a high level service oriented architecture designed to enable adaptive service composition for the IoT. The reconfiguration of the composed services is performed at design time through user interfaces. Thus, automated adaptations in response unforeseen context changes are not supported.

Hussein et al. [14] proposed a model-driven approach to enable IoT systems to adapt at runtime. A set of system states and adaptation triggers are modelled at design time based on anticipated context changes. When an adaptation is triggered, the system is switched from one state to another based on the designed models. Ciortea et al. [1] proposed an agent-based approach for composing goal-driven IoT mashups. IoT things are modelled offline as agents or artefacts according to their capabilities. Agents rely on predefined plans which specify how their goals are achieved. In cases where goals cannot be achieved individually, agents interact and cooperate, in a network like system called STN, to compose IoT mashups which achieve the goals. Marrella et al. [15] presented SmartPM, a framework for enabling automated adaptation of processes using situation calculus and AI planning. Processes are defined by designers using a graphical editor. Events and exceptions which disrupt the processes enactment are automatically detected and recovery procedures are automatically generated to adapt faulty processes. Seiger et al. [21] proposed another framework for enabling workflow-based Cyber-physical Systems to self adapt. The framework utilizes the MAPE-K loop notion to automatically adapt workflows in response to detected failures. Relying on predefined plans in [1], process (or workflows) in [15, 21] or models of possible system states and adaptation triggers in [14] limit systems flexibility as it is hard to foresee at design time IoT systems constituents in dynamic and mobile contexts.

The MobIoT is a service-oriented middleware designed to address the heterogeneity, interoperability and scalability in mobile IoT contexts [19]. User requests are achieved by applying an ontological-based composition approach which exploits the notation of probabilistic registration and lookup mechanisms. The approach addresses specific types of requests related to real world measurement in the physics and chemistry domains. Mayer et al. [20] proposed an approach to achieve user goals by dynamically

composing service-based IoT mashups. IoT things are described by means of semantic services. User goals are described in a machine understandable way and can be expressed via a user interface. Given the semantic description of the user goal and a list of services, a plan which comprises a set of services is generated to achieve the goal. The proposed approach supports adaptation apropos the dynamic availability of services. Compared to our approach, we consider additional intrinsic contextual properties of IoT things including connectivity status, operational status (i.e., on/off) and if they rely on batteries. In addition, exploiting the notion of events, our approach possesses more effective reasoning capabilities about performed adaptations.

3 Emergent Configurations Background

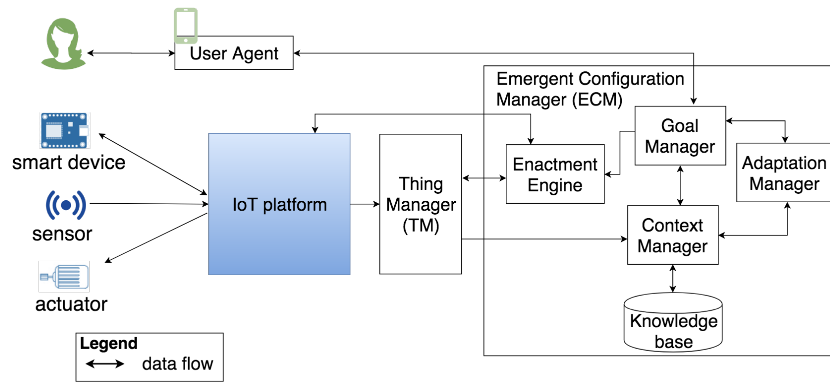


Fig. 1: A high level architecture for realizing ECs

In [2], we presented an abstract architecture for ECs describing how they are automatically formed and adapted. In this section, we first overview the proposed architecture, then refine it by presenting the `ECo-IoT`. The abstract architecture, illustrated in Figure 1, comprises a set of components. The *User Agent* (UA) is an application running on one of the existing smart devices (e.g., smartphone) and used to enable users to interact with the system. The *Emergent Configuration Manager* (ECM) is responsible for automatically forming ECs to achieve user goals (if possible) and for adapting ECs in response to runtime context changes. The *Thing Manager* (TM) is responsible for discovering and registering available IoT things, monitoring their statuses and reporting any changes to the ECM. Finally, a *set of (IoT) things* which, following indications of the ECM, communicate and collaborate to realize ECs. The ECM comprises the following subcomponent:

(i) *Goal Manager*: is responsible for interpreting user goals and forming ECs which achieve them;

(ii) *Adaptation Manager*: is responsible for adapting ECs in response to context changes. ECs are adapted by executing the Monitor-Analyze-Plan-Execute plus Knowledge (MAPE-K) loop -better described in Section 4.1;
(iii) *Context Manager*: is responsible for maintaining ECs context;
(iv) *Enactment Engine*: is responsible for enacting ECs by commanding or requesting ECs constituents to perform functionalities in specific orders;
(v) *System Knowledge Base*: is the container of the context of ECs.

In this paper, we mainly focus on the realization of the ECM component while the realization of the TM and UA are planned for future works.

4 The ECo-IoT Approach

In this section, we present the `ECo-IoT` approach for realizing ECs. More specifically, we present a process and a refined architecture developed to enable the automatic formation and adaptation of ECs.

4.1 The ECo-IoT Process

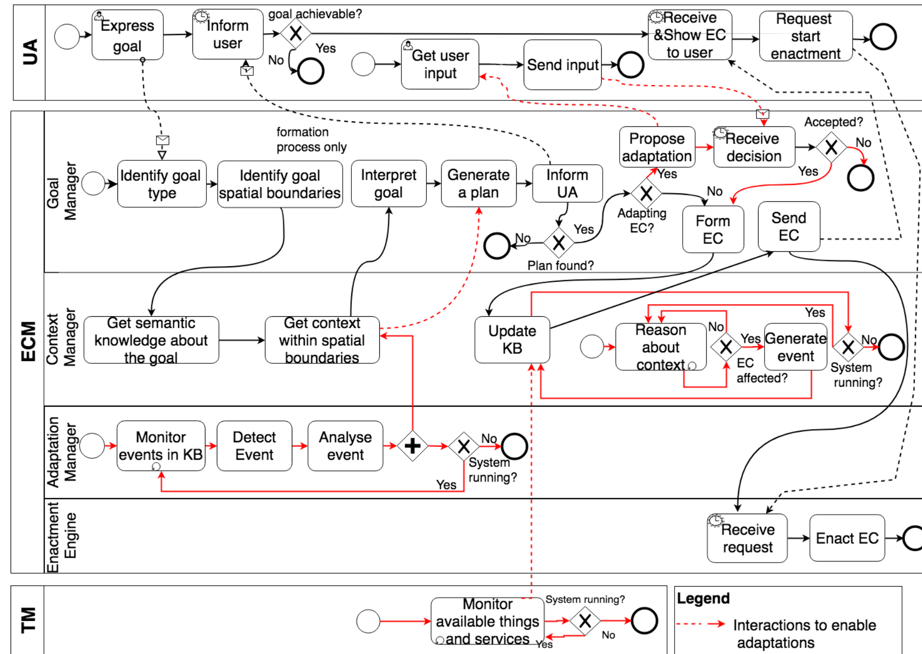


Fig. 2: ECs formation and adaptation process

Figure 2 shows the ECo-IoT process[3] for enabling the automated formation and adaptation of ECs. The *EC formation process* starts with a user interacting with the goal manager (via the user agent) to express her/his goal by sharing a goal description. The goal manager expects that the shared description comprises (at least) a goal type (e.g., deliver presentation) and spatial boundaries (e.g., a specific room). After analyzing the goal description, the goal manager requests the context manager to provide the semantic knowledge about the goal type and the context of the specified spatial boundaries including, for instance, available things and their capabilities. In cases where the goal type or the spatial boundaries are not known to the context manager, the goal interpreter engages in a complex process which involves interactions with the user to better analyze her/his goal description. It also interacts with the user to identify e.g., her/his preferences (when applicable). Based on the goal interpretation and the retrieved context, the goal manager tries to generate a plan which achieves the user goal when enacted. The user is informed whether or not the requested goal is achievable. If a plan is found, the goal manager forms an EC and forwards it to the context manager which updates knowledge base. Afterwards, the goal manager sends the EC to the user agent which illustrates needed info to the user (e.g, EC constituents). Then, the user agent requests the enactment engine to enact the EC. The user goal interpretation and ECs enactment processes are out of the scope if this paper and planned for future works.

To enable the automated adaptation of ECs, the *EC adaptation process* exploits the notion of Monitor-Analyze-Plan-Execute plus Knowledge (MAPE-K) loop adopted from the field of Self-adaptive Systems. In general, ECs are adapted due to: (i) changes in available things and their status; (ii) evolving user goals. Several components *monitor* various parts of the context. The thing manager continuously *monitors* available things and their status. It mainly *monitors* things connectivity status, operational status (i.e, on/off), locations and battery levels (when applicable). The thing manager periodically reports detected changes to the context manager which updates the knowledge base. The context manager continuously *monitors* the context and *analyzes* if context changes affect any running EC. If an EC is affected, the context manager generates events about the detected changes and updates the knowledge base. In this paper, we consider the following types of events: (1) an IoT thing is disconnected; (2) an IoT thing is no longer situated within the spatial boundaries of an EC which comprises it; (3) an IoT thing is running out of battery. As can be noted, these events model only the loss of IoT things. We plan to consider events which model the availability of new things and evolving user goals in our future work.

The adaptation manager continuously *monitors* events in the knowledge base. When an event is detected, the adaptation manager *analyzes* it and decides a proper adaptation procedure. Based on the detected event type, adaptation procedures can be *reactive* or *proactive*. Reactive adaptations procedures are applied in response to events of type (1) and (2), while proactive adaptations are applied in response to events of type (3). An example of a reactive adaptation is to propose the user to continue the presentation using an available smart screen after the sudden loss of the used smart projector. An example of a proactive adaptation is to propose the user to switch to an available laptop

[3] For presentation purposes, in Figure 2, we omit some details. The process is modelled using the standard Business Process Model and Notation (BPMN) http://www.bpmn.org

to stream the presentation as the battery level of the used smartphone is less than a specific threshold and expected to turn off soon. The number of proactive events that can be generated about a thing battery level is configurable in order not to overwhelm the user with many messages. The goal manager then tries to find a new plan that maintains the achievement of the user goal. If a plan is found, the goal manager proposes the adaptation to the user and asks for her/his approval. If the user accepts the proposal, the goal manager forms a new version of the EC, changes the status of the former version and links both ECs via the event which triggered the adaptation process. The status of an EC is *ready for execution* when it is newly formed, *in execution* when it is being enacted, *adapted* in case it is adapted, *enacted successfully* in case it achieves the goal and *failed* in case it cannot be adapted to maintain the goal achievement. The EC versioning subprocess enables the reasoning about all performed adaptations. The process then continues as described in EC formation process.

4.2 A Refined Architecture for Realizing ECs

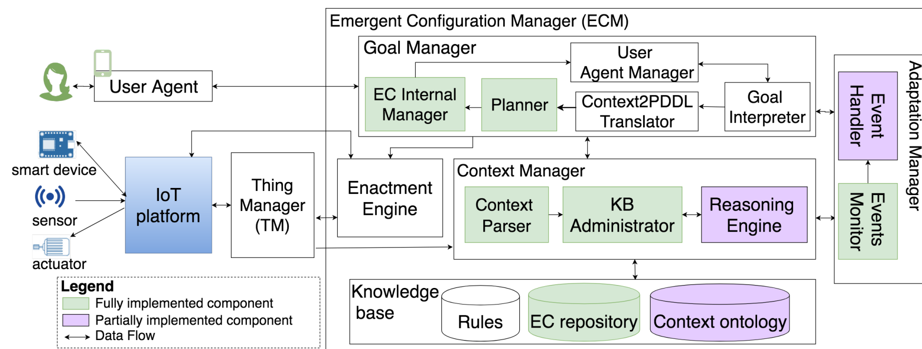


Fig. 3: A refined architecture for realizing ECs

Figure 3 illustrates a refined architecture based on the abstract architecture and the ECs realization requirements formulated in [2]. The architecture also complies with the well known architectural design principles such as separation of concerns and modularity. In the following, we describe in more details the ECM subcomponents which we mainly focused on for the realization of ECs in this paper.

4.3 Managers Components

Goal Manager. This component comprises five subcomponents responsible for forming ECs which achieve user goals when enacted. The *User Agent Manager* is responsible for interacting with the User Agent. The *Goal Interpreter* is responsible for analyzing goals descriptions in the context of their spatial boundaries. The *Planner* is responsible for generating (if possible) plans which achieve the goals when enacted. It

supports the system to cope with dynamic (unforeseen) context changes and to utilize heterogeneous IoT things. Generated plans comprise a set of actions (i.e., tasks) which might have different complexity levels based on the autonomy of available things. The *Context2PDDL Translator* is responsible for generating files needed for the planning process by translating goals interpretations and the context of identified spatial boundaries to the Planning Domain Definition Language (PDDL) [6].

The *EC Internal Manager* is responsible for updating the user agent manager about ECs formation and adaptation processes, instantiating ECs when plans are found by the planner, maintaining their statuses and versioning them when they are adapted.

Context Manager. This component comprises three subcomponents responsible for maintaining and reasoning about ECs context. The *Context Parser* is responsible for receiving and parsing information about available things and their statuses. This information is forwarded to the *Knowledge Base Administrator* which is responsible for manipulating the knowledge base. The *Reasoning Engine* comprises two subcomponents, the *Semantic Reasoner* and the *Rule-based Engine*. The semantic reasoner is responsible for querying the context ontology and inferring semantic knowledge. The rule-based engine is responsible for monitoring the KB and generating events when the conditions of the rules defined in the rules repository are met.

Adaptation Manager. This is an event-based component which comprises two subcomponents responsible for adapting ECs in response to context changes. The *Event Monitor* is responsible for detecting events created by the reasoning engine. The *Event Handler* is responsible for analyzing how detected events affect running ECs and for triggering proper adaptation processes when needed.

4.4 Knowledge Base Component

Context Ontology. This component contains the semantic representation of ECs context. The context of IoT systems can be represented by various means such as graphical based modelling, markup scheme based modelling, key-value based modelling and ontology based modelling to mention a few [3]. Ontologies are composed of a set of concepts represented by classes, relations represented by properties and concept instances represented by individuals. They are considered among the most suitable techniques to maintain systems contexts [3, 23]. Reasons for this include: their expressiveness, representation of shared understanding of knowledge among involved parties and the availability of several tools and reasoning engines which support their usage [3, 17, 24]. Figure 4 illustrates the main classes of the ECs context ontology.

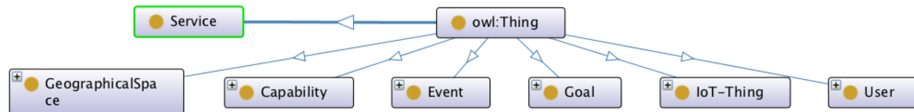


Fig. 4: Main classes of the ECs context ontology

The *owl:Thing* class is the superclass for all classes in the ontology. The *IoT-Thing* class models the types of IoT things supported by the system. It has three subclasses, *Sensor*, *Actuator* and *SmartDevice*. The *Service* class models external services which can be consumed by the system. For instance, a service which can be used to convert a presentation to a format supported by a smartphone. The *User* class models system users. The *Goal* class models the types of goals which are supported by the system. The exploration of this class is out of the scope of this paper and is planned for future works. The *Capability* class models the types of functionalities provided by things types. Functionalities types are modelled as subclasses e.g., senseLightLevel. The *Event* class models specific types of events in the context of ECs. It has three subclasses *GoalBasedEvents*, *AvailabilityEvents* and *UnavailabilityEvents*. The UnavailabilityEvents class has three subclasses which model the events described in Section 4.1. The GoalBasedEvents and AvailabilityEvents classes model respectively events related to evolving user goals and the availability of new things. The exploration of these classes is planned for future works. The *GeographicalSpace* class models available locations. The structure of this class is adopted from the SOUPA space ontology [4].

The ontology also comprises object properties which model the following relations: (1) users have goals and locations; (2) IoT things have capabilities and locations; (3) services have capabilities; (4) IoT things are connectable; (5) IoT things are connected; (6) Geographical spaces have sub spaces. The ontology also comprises a set of data properties which model: (1) things operational status; (2) things connectivity status; (3) if a thing relies on batteries or not; (4) a thing battery level when applicable; (5) capabilities preconditions and effects written in the PDDL language; (6) timestamps of triggered events.

Our approach requires that a developer models offline the mentioned context ontology. However, note that several parts of the knowledge (e.g., things individuals, locations, battery levels, connectivity statuses, etc) can be *automatically* populated from the data sent by the thing manager with the support of an IoT platform. For instance, the Amazon AWS-IoT platform[4] supports the management and runtime monitoring of registered things. Geographical spaces (i.e., locations) can be populated automatically by consuming services in a smart building. Still, developers need to extend the ontology when introducing new types of things and defining things capabilities.

EC repository. This component is a container of active and archived ECs. An EC comprises a user goal and its' spatial boundaries, a set of things and capabilities, the plan generated to achieve the goal, the EC version and status.

Rules repository. This component comprises business rules which are application specific and generic rules which are required to realize ECs. An example of a business rule is to lower light levels in a room if a presentation is ongoing via a projector. An example of a generic rule is that two things are connectable if they have Wifi capabilities.

5 Prototype Implementation

In this section, we present some implementation[5] details about the prototype we developed to validate the feasibility of `ECo-IoT`. The prototype is implemented in Java

[4] https://aws.amazon.com/iot/

[5] The prototype code is available at https://github.com/iotap-center/eco-iot

(version 1.8) and integrates the OWLAPI (version 5.1.3)[6] with the JavaFF planner (version 2.1.5)[7]. The context ontology is represented by OWL [16] and the EC repository is realized by a relational PostgreSQL database.

5.1 The Knowledge Base

The Context Ontology. Let us consider that "room19" is a smart meeting room which contains a set of IoT things. Table 1 and Table 2 illustrate (partially) how the available things are modelled in the context ontology. As capabilities preconditions and effects are modelled in PDDL, they are explored in Section 5.4. We created an individual of the user class and set the user location to "room19".

Table 1: Representation of some object properties in the context ontology

individual	**individual class**	**hasCapability**	**capability class**	**hasLocation**
smart_projector_a	SmartProjector	illustrate_presentation_a	IllustratePresentation	room19
smart_screen_b	SmartScreen	illustrate_presentation_b	IllustratePresentation	room19
smart_phone_c	SmartPhone	stream_presentation_c	StreamPresentation	room19
laptop_d	Laptop	stream_presentation_d, illustrate_presentation_d	StreamPresentation, IllustratePresentation	room19

Table 2: Representation of some data properties in the context ontology

individual	**individual class**	**isConnected**	**hasStatus**	**reliesOnBattery**	**hasBatteryLevel**
smart_projector_a	SmartProjector	true	false	false	N/A
smart_screen_b	SmartScreen	true	false	false	N/A
smart_phone_c	SmartPhone	true	true	true	20%
laptop_d	Laptop	true	false	true	60%

5.2 The Context Manager

To enable the ECM to receive status updates about available things from the thing manager, we implemented the *context parser* component based on the Publish/Subscribe model and subscribed it to topics defined at the simulated thing manager. The KB administrator utilizes the OWLAPI to maintain the knowledge represented in the context ontology.

To dynamically generate events, we implemented a thread which periodically checks the EC context ontology for new changes in the status of available things (see Section 4.1). We plan to integrate a rule-based engine to enable the dynamic manipulations of rules at runtime. This will enable developers and end users to define business rules at runtime to better configure their smart environments.

[6] http://owlapi.sourceforge.net [7] http://personal.strath.ac.uk/david.pattison/#software

5.3 The Adaptation Manager

In this first prototype implementation, the *event monitor* is implemented as a thread which continuously checks if new events are created in the KB. The *event handler* implements the part of adaptation process which handles the events detected by the event monitor.

5.4 The Goal Manager

In the implemented prototype, we integrated the JavaFF planner which is an open source planner based on [9]. The JavaFF planner requires two PDDL files to generate a plan, namely, the domain file and the problem file. For the prototype, we have manually defined both files. We plan to implement the Context2PDDL component to automatically generate these files. The domain file comprises three basic parts, namely, types, predicates and actions. Types represent the hierarchy of information structure in the planning domain. More specifically, they are the translation of the classes hierarchy in the context ontology. For instance, as illustrated below, smart devices, sensors and actuators are sub-types of the type IoT-Thing.

```
(:types ... smartDevice sensor actuator — IoT—Thing ...
```

Predicates are the result of translating data and object properties. For instance, as illustrated below, the ontology data property *hasStatus* is translated to the hasStatus predicate in the domain file. The symbol ? is used to declare a variable (e.g., ?t) of the type which directly follows it (e.g., IoT-Thing).

```
(:predicates ... (hasStatus ?t — IoT—Thing ?st — status) ...
```

Finally, actions represent the translation of things capabilities modelled in the ontology. We recall that capabilities data properties are already specified in strings that are described in PDDL. As illustrated in below, actions are described by parameters, preconditions and effects. Parameters of a capability are represented by variables. Preconditions specify the conditions needed to perform actions. For example, it is only possible to turn on a thing if it is already turned off. Effects represent changes in the state of the world if actions are executed successfully. For instance, executing successfully the action defined below results in turning on a specific IoT thing.

```
(:action turnThingOn
      :parameters (?thing — IoTThing)
      :precondition (hastatus ?thing off)
      :effect (hastatus ?thing on))
)
```

The planning problem file comprises three basic parts, namely, objects, initial states and desired states. As illustrated below, objects represent instances of the types defined in the domain file. The set of instantiated objects in the problem file is the same set of individuals in the context ontology.

```
(:objects ... smart_projector_a — smartProjector...
```

The initial state of the world is a set of predicates representing a particular situation. As illustrated below, the status of the smart_projector_a object is *false* meaning that the smart projector is turned off.

```
(:init ...(hasStatus smart_projector_a false)...
```

Desired states are described by means of predicates which specify desired changes in the state of the world. Desired states represent the translation of goals interpretations. The desired state in the smart meeting room scenario is illustrated below. The *EC internal manager* is implemented to perform its responsibility described in Section 4.1.

```
(:goal (illustrate smartProjector1  presentation1))
```

6 Experimenting ECo-IoT

In this section we validate the feasibility of our approach by putting in action the prototype we implemented. We recall that we already realized a number of the `ECo-IoT` components in this first prototype, as shown in Figure 3. The additional effort is planned for future work. This means that here we validate the feasibility of `ECo-IoT` while gaining initial insights about the performances of a subset of it.

6.1 Forming ECs

We assume that the user has expressed her/his goal to be "deliver a presentation in room19" and that the goal is received by the goal manager which derived the goal type (i.e., deliver a presentation) and the spatial boundaries (i.e., room19). In the current implementation, the goal interpretation process is prespecified and is dedicated to run the smart meeting room scenario. The goal manager queried context manager about the (capabilities of) things which are situated in room19. The context manager responded with the set of individuals presented in the Tables 1 and 2. We envision that the output of the goal interpretation correlates decomposed goals with available capabilities. More specifically, to specify which of the existing devices can store a presentation file, can stream it and which of them can illustrate it.

To specify where the presentation file is stored, the goal interpreter interacted with the user through the user agent manager and the user agent. In addition, it asked the user about the preferred device to stream and illustrate the presentation by highlighting possible options. In the current implementation, the user interacted with the goal interpreter through the console. The user selected her/his smartphone as the source of the presentation and the available smart projector as the preferred illustration device. Based on that, the plan illustrated below was generated to achieve the user goal. First, the thing manager turns the projector on and connects it to the user smartphone. Then, the smartphone streams the presentation to the smart projector which illustrates it. The formation process then continued as described in Section 4.1.

```
(turn_on_thing smart_projector_a)
(connect_things smart_phone_c smart_projector_a)
(stream_presentation presentation1 smart_phone_c smart_projector_a)
(illustrate_presentation smart_projector_a presentation1)
```

6.2 Adapting ECs

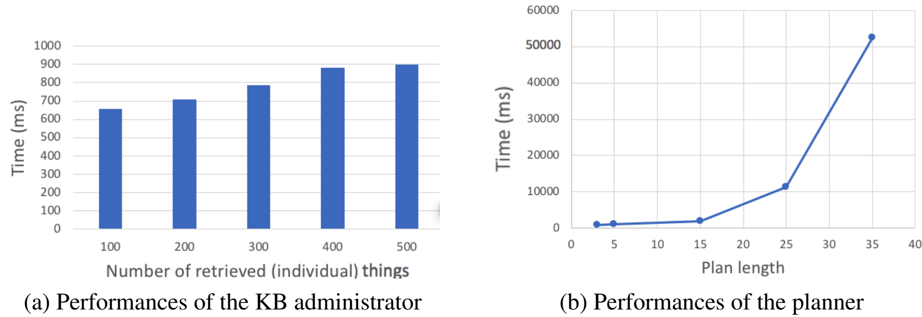
To simulate the enactment of the EC formed in Section 6.1, we updated manually the system knowledge base simulating the supposed execution of the generated plan. For instance, we changed the status of smart_projector_a to be true meaning that the smart projector is turned on. As already mentioned, this process will be automated when the thing manager and the enactment engine are realized. Then, we published two messages to the context manager to trigger *reactive* and *proactive* adaptation processes. We illustrate one scenario per each category. The first message stated that smart_projector_a is disconnected. The message was received by the context manager which generated an event that was detected by the adaptation manager. In response, the adaptation manager triggered the adaptation process which generated a plan that substitutes smart_projector_a with smart_screen_b. The adaptation was proposed to the user (via a printed message on the console) and the EC was versioned properly after the user accepted the proposal.

The second message stated that the battery level of smart_phone_c is 12% which was less than the configured threshold to trigger the proactive adaptation process. The context manager automatically created an event which was detected by the adaptation manager. The adaptation manager triggered the proactive adaptation process as described in Section 4.1. The user was proposed to switch to laptop_d to continue the presentation. The user accepted the proposed adaptation by communicating via the console and the EC was versioned properly.

6.3 Discussion

The dynamicity of IoT contexts and the involvement of the human in the loop require ECs to be responsive. Designing ECs to be goal-oriented, supports meeting this requirement. Indeed, specifying the goal spatial boundaries notably mitigates the well-known scalability issue in the IoT [12]. The implemented prototype presents an evidence about the feasibility of the `ECo-IoT` approach. Although some components are not implemented yet, they do not seem to require intensive resources for performing their responsibilities. The semantic model of the context ontology, having the goal ontology defined, is expected to reduce the complexity of the goal interpretation process. Although it is required that the context ontology be modelled by developers, several parts of the operational (dynamic) knowledge can be populated automatically as described briefly in Section 4.4. From a user perspective, we envision that the user needs only to express her/his goal without being concerned about how available things can achieve it. In addition, the goal interpretation process should not overwhelm the user with many requests. We plan to investigate these aspects in our future work.

In Section 5.4, we explained the mapping between the PDDL files structures and the context parameters. The process of translating context to PDDL is not expected to be computationally complex as we do not expect to deal with a huge number of things due to goals spatial boundaries and to the responsiveness of the context retrieval process (see below). The process of creating events is not complex either, as it creates events when rules preconditions are met. Likewise, the process of detecting events is not complex as it is not more than continuously querying the KB for new events. The process



(a) Performances of the KB administrator

(b) Performances of the planner

Fig. 5: Performances of the KB administrator and the planner

of enacting ECs may impose some complexity in cases where ECs aim to achieve contradicting goals, when ECs compete on available resources or when ECs are formed, enacted or adapted in uncertain contexts.

The fast retrieval of the context contributes to the responsiveness of the system. Therefore, we conducted an experiment to evaluate the knowledge retrieval process when the number of individuals in the context ontology increases. Figure 5a illustrates the response time of a query which retrieves an increasing number of fully specified things situated within specific spatial boundaries. Things were automatically instantiated, specified and persisted. As can be noted, it is evident that the KB administrator scales well when the number of individuals in the ontology increases.

The planner is another key component which is involved whenever an EC is formed or adapted. AI planning is well known for being a resource intensive process [13]. Therefore, we conducted an experiment to evaluate how the planner performs in a smart room that has 200 IoT things which possess 800 capabilities on average. Note that these numbers do not represent an upper bound or a limitation of the approach. To setup the experiment, the planning domain file was automatically generated. To simulate realistic conditions, several initial states were declared, many generated actions were complex with respect to their preconditions and effects and goals were achievable by multiple possible plans. Figure 5b illustrates that the planer responds in the order of a few seconds when generated plans comprise up to 24 actions. All experiments were conducted on a dual-core CPU running at 2.7GHz, with 16 Gb memory.

In the future, we expect to have some additional computational complexity from the components to be developed. In order to mitigate the complexity, we plan to work on proposing intelligent and efficient goal interpretation techniques that reduce, as much as possible, domain and problem spaces of AI planning processes.

7 Conclusion and Future Work

New things get continuously connected and embedded everywhere. This gives to the Internet of Things (IoT) an increasingly important role in all aspects of our society. Devising concrete architectures and approaches is then a needed enabler to support the

effective use of IoT systems. In this paper, we presented the ECo-IoT approach to enable the automated formation and adaptation of Emergent Configurations while dealing with realistic IoT scenarios including runtime changes. We also described our first prototype for the validation of ECo-IoT in terms of its feasibility and performances of some key components.

Some future directions we plan to investigate include: devising a suitable goal language; exploring (intelligent) processes and techniques for effective and efficient goal interpretation; investigating automated mechanisms supporting rule-based reasoning; handling situations where concurrent ECs compete on available resources or aim to achieve contradicting goals; proposing mechanisms to enable the automated adaptation of ECs in response to evolving user goals and the availability of new things; devising mechanisms to enable the formation, enactment and adaptation of ECs in uncertain contexts. Additionally, we aim at: extending the prototype by implementing e.g., a thing manager exploiting an IoT platform, rule engine, a Context2PDDL translator, a goal interpreter, and an enactment engine; conducting trade off analysis among existing IoT deployment models to support decision making about the ECo-IoT deployment; and performing a more extensive evaluation.

Acknowledgment

This work is partially financed by the Knowledge Foundation through the Internet of Things and People research profile (Malmö University, Sweden).

References

1. Ciortea, A. and Boissier, O. and Zimmermann, A. and Florea, A. M.: Responsive Decentralized Composition of Service Mashups for the Internet of Things. In: 6th ACM International Conference on the Internet of Things, pp. 53–61. ACM, 2016.
2. Alkhabbas, F. and Spalazzese, R. and Davidsson, P.: Architecting Emergent Configurations in the Internet of Things. In: IEEE International Conference on Software Architecture (ICSA), pp. 221–224. IEEE, 2017.
3. Perera, C. and Zaslavsky, A. and Christen, P. and Georgakopoulos, D.: Context Aware Computing for The Internet of Things: A Survey. In: IEEE Communications Surveys & Tutorials, pp. 414–454. IEEE, 2014.
4. Chen, H. and Perich, F. and Finin, T. and Joshi, A.: SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications. In: 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, pp. 258–267. IEEE, 2004.
5. Ciccozzi, F. and Spalazzese, R.: MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering. In: 10th Intern. Symp. on Intelligent Distributed Computing, pp. 67–76. Springer, 2017.
6. McDermott, D. and Ghallab, M. and Howe, A. and Knoblock, G. and Ram, A. and Veloso, M. and Weld, D. and Wilkins, D.: PDDL - The Planning Domain Definition Language. 1998.
7. Miorandi, D. and Sicari, S. and De Pellegrini, F. and Chlamtac, I.: Internet of things: Vision, applications and research challenges. In: Ad Hoc Network, pp. 1497–1516. Elsevier, 2012.
8. Bauer, M. et.al.: IoT Reference Architecture. In: Enabling Things to Talk, pp. 163–211. Springer, 2013.

9. Hoffmann, J. and Nebel, B.: The ff planning system: Fast plan generation through heuristic search. In: Journal of Artificial Intelligence Research, pp. 253–302. AI Access Foundation and Morgan Kaufmann, 2001.
10. Dar, K. and Taherkordi, A. and Rouvoy, R. and Eliassen, F.: Adaptable Service Composition for Very-Large-Scale Internet of Things Systems. In: 8th Middleware Doctoral Symposium, pp. 2. ACM, 2011.
11. Kramer, J. and Magee, J.: Self-Managed Systems: an Architectural Challenge. In: Future of Software Engineering, pp. 259–268. IEEE Computer Society, 2007.
12. Atzori, L. and Iera, A. and Morabito, G.: The internet of things: A survey. In: Computer Networks, pp. 2787–2805. Elsevier, 2010.
13. Ghallab, M. and Nau, D. and Traverso, P.: Automated Planning: theory and practice. Elsevier, 2004.
14. Hussein, M. and Li, S. and Radermacher, A.: Model-driven Development of Adaptive IoT Systems. In: 4th International Workshop on Interplay of Model-Driven and Component-Based Software Engineering, pp. 20-27, 2017.
15. Marrella, A. and Mecella, M. and Sardina, S.: Intelligent Process Adaptation in the SmartPM System. In: ACM Transactions on Intelligent Systems and Technology (TIST). ACM, 2017.
16. McGuinness, D. L. and Van Harmelen, F. et al.: OWL Web Ontology Language Overview. W3C recommendation, 2004.
17. Noy, N.F. and McGuinness, D.L. et al.: Ontology Development 101: A Guide to Creating Your First Ontology, Stanford knowledge systems, 2001.
18. Spiess, P. and Karnouskos, S. and Guinard, D. and Savio, D. and Baecker, O. and De Souza, L.M.S. and Trifa, V.: SOA-based Integration of the Internet of Things in Enterprise Services. In: IEEE International Conference on Web Services, pp. 968–975. IEEE, 2009.
19. Hachem, S. and Pathak, A. and Issarny, V.: Service-Oriented Middleware for the Mobile Internet of Things: A Scalable Solution. In: IEEE GLOBECOM: Global Communications Conference. IEEE, 2014.
20. Mayer, S. and Verborgh, R. and Kovatsch, M. and Mattern, F.: Smart Configuration of Smart Environments. In: IEEE Transactions on Automation Science and Engineering, pp. 1247–1255. IEEE, 2016.
21. Seiger, R. and Huber, S. and Heisig, P. and Aßmann, U.: Toward a framework for self-adaptive workflows in cyber-physical systems. In: Software & Systems Modeling, pp. 1–18. Springer, 2017.
22. Sousa, J.P. and Garlan, D.: Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. In: Software Architecture, pp. 29–43. Springer, 2002.
23. Strang, T. and Linnhoff-Popien, C.: A Context Modeling Survey. In: 1st International Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp., 2004.
24. Wang, X. H. and Zhang, D. Q. and Gu, T. and Pung, H. K.: Ontology Based Context Modeling and Reasoning using OWL. In: 2nd IEEE Annual Conference on Pervasive Computing and Communications Workshops, pp. 18–22. IEEE, 2004.