Time-centric and resource-driven composition for the Internet of Things

Zakaria Maamar – *Zayed University, Dubai*
Noura Faci – *Université Claude Bernard, Lyon*
Mohammed Al-Khafajiy – *University of Reading*
Murtada Dohan, *University of Northampton*

Abstract

Internet of Things (IoT), one of the fastest growing Information and Communication Technologies (ICT), is playing a major role in provisioning contextualized, smart services to end-users and organizations. To sustain this role, many challenges must be tackled with focus in this paper on the design and development of thing composition. The complex nature of today's needs requires groups of things, and not separate things, to work together to satisfy these needs. By analogy with other ICTs like Web services, thing composition is specified with a model that uses dependencies to decide upon things that will do what, where, when, and why. Two types of dependencies are adopted, regular that schedule the execution chronology of things and special that coordinate the operations of things when they run into obstacles like unavailability of resources to use. Both resource use and resource availability are specified in compliance with Allen's time intervals upon which reasoning takes place. This reasoning is technically demonstrated through a system extending EdgeCloudSim and backed with a set of experiments.

Introduction

Over the years, Information and Communication Technology (ICT) practitioners have been advocating for different software solutions to achieve distributed and heterogeneous system integration (*aka* interoperability, [1]). Web services are among these solutions that usually acts as wrappers over systems allowing to expose these systems' functionalities in a standard/homogeneous way. Web services have been backed by different standards and specifications such as eXtensible Markup Language (XML), Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP), Universal Description, Discovery, and Integration (UDDI), and Business Process Execution Language (BPEL) [2]. Among all these standards and specifications, BPEL has played a major role in reinforcing the role of Web services as a solution integrator. BPEL composes component Web services into what is usually referred as composite Web services. The complex and changing nature of users' and organizations' needs would require more than one component Web service that would wrap all necessary systems contributing collectively to satisfy these needs.

In conjunction with Web services, there have been a lot of advances in ICT with focus lately, among many others, on the Internet-of-Things (IoT, [3,4]). IoT is about making things/devices like sensors and actuators act over the cyber–physical surrounding so, that, contextualized, smart services are provisioned to users and organizations. According to Gartner,[1] 6.4 billion connected things were in use in 2016, up 3% from 2015, and will reach 20.8 billion by 2020. IoT is a perfect demonstration of Weiser's vision about ubiquitous computing when he states in 1999 that ''*the most profound technologies are those that* ***disappear****. They weave themselves into the fabric of everyday life until they are indistinguishable from it* '' [5]. IoT uses are diverse such as controlling the freshness of goods in warehouses, monitoring elderly people's health, and tracking vehicle flows on highways. Despite the bright side of IoT, many are still skeptical about IoT's benefits to users and organizations because of things' limited cognitive capabilities and privacy invasion. Indeed, Wu et al. compare things to ''*awkward stegosaurus: all brawn and no brains*'' [6] and Green states that IoT needs to be smarter so, that, things would go beyond the regular operations of sensing and sometimes actuating [7].

By analogy with Web services composition [2], we discuss in this paper the necessary steps and means for composing things so they could be integrated into complex business applications. Multiple challenges, like diversity of things' development and communication technologies [8] and passive nature of things [9], are confining things into *silos*, which could deprive them from being the technology of choice for developing advanced cyber–physical systems, for example. To identify these steps and means, the following actions are taken: expose capabilities of things to external stakeholders, identify potential dependencies between things based on things' capabilities, develop mechanisms allowing things to engage in compositions, and technically demonstrate these mechanisms. Contrarily to some existing works that advocate for thing composition from a service perspective [10–12], we argue that this perspective is not appropriate for capturing things' intrinsic characteristics. Things are completely ''immersed'' in cyber–physical surroundings while services are ''immersed'' in cyber surroundings, only. Therefore, a different and novel way to approach thing composition is deemed necessary.

The rest of this paper is organized as follows. Section 2 motivates thing composition using a case study and discusses some related works. Section 3 presents conceptual details about our time-centric, resource-driven approach to compose things. Section 4 suggests ways for enhancing this approach by connecting things using social relations and tackling the challenge of resource unavailabilities. The approach's technical details and results of experiments are presented in Section 5. Section 6 concludes the paper and identifies future research work.

Background

This section presents a case study that motivates examining thing composition and then, discusses some works that touched upon thing composition, as well.

Motivations

We consider a case study that sheds light on the importance of composing things to assist elderly people with their daily activities. Many studies confirm that population ageing is a dominant global

[1] www.gartner.com/newsroom/id/3165317.

demographic trend of the 21st century.[2] In the context of a living room, a straightforward scenario would consist of composing the remote control, the smart TV, and light switches together to offer a seamless watching experience to a group of elderly persons based on their habits and preferences. We could think of a temporary composition since the remote control could also be used for other needs like opening and closing the blinds in the same living room when the smart TV is off. Another scenario would be an elderly person's smart watch that would team up with her dispenser to automatically dispense medicine doses according to her blood-pressure level. The composition between the smart watch and dispenser could be set according to the duration of the treatment but, then, extended, should the treatment need to be renewed. To complete these two scenarios, our approach requires that (i) the capabilities of things should be known (what can the smart TV do?), (ii) the dependencies between things should be identified (how can the smart TV and remote control be ''glued'' together?), (iii) the consistency of these dependencies should be checked (when can the remote control send instructions to the smart TV?), and (iv) the engagement of things in compositions should be regulated (how can the remote control along with the smart TV confirm their participation in a composition knowing that the remote control could participate in other on-going compositions?).

Related work

Many studies have examined thing composition from different perspectives [13–19]. Firstly, Khaled and Helal proposed a programming framework to capture inter-thing relations into IoT applications [13]. The framework's objective is to ensure that, on top of things' services, logical and functional ties between services are not overlooked. The framework introduces three primitives that are thing service, thing relation (e.g., control/controlled by, drive/driven by, and extend/extended by), and recipe. Secondly, Krishna et al. acknowledged the difficult job of composing things due to their heterogeneity and suggest a Web-based tool called IoT Composer [14]. This one assists users with selecting, configuring, and binding things together. An LNT process-algebraic code is generated and then, submitted to a verification toolbox that checks the satisfaction of some properties like comparability and deadlock-free. Thirdly, Åkesson et al.composed services hosted on IoT devices using ComPOS (Composition language for Palcom Oblivious Services) [18]. Native services contain computation and interaction with the physical world, while composition services combine native services into applications, mediating and adapting messages between them. Fourthly, Seiger et al. relied on mix reality to facilitate the exercise of developing IoT processes [19]. They suggested HoloFlows to address the complexity of developing IoT applications for users and domain experts with limited technical background. Using HoloFlows, physical sensors and actuators are mapped onto virtual components to connect together through virtual wires. Finally, Maamar et al. examined thing composition in the context of thingsourcing [15]. They argue that by analogy with people who participate in crowdsourcing, things could ''act in the same way'' leading to the formation and management of a crowd of things from which a select group of things would be composed and assigned users' demands to complete. The authors handled an unexpected traffic diversion by composing multiple things such as traffic light, traffic cone, speed-limit sign, speed camera, and flip-disc display to restore normal circulation. Worth mentioning, though brief, some works that advocate for blending social computing with IoT. For instance, Atzori et al. consider things as intelligent objects that could form social networks of objects [20]. These networks

[2] www.weforum.org/agenda/2019/10/ageing-economics-population-health.

could be built upon relations such as parental, co-location, co-work, ownership, and social. Also, Hussain et al. suggest a software agent-centric semantic social-collaborative network that provides functionality to represent and manage cyber–physical resources in a social network [21].

While existing works including those summarized above support the idea of an IoT marketplace where things would sign-up and sign-off looking for opportunities to complete demands [22], there is a need for mechanisms that would compose those things that would express interest in these demands. We associate these mechanisms with a four-stage approach concerned with capabilities of things, dependencies between things, consistency of dependencies, and engagement of things. More details about these four stages are given in the next sections.

Approach for thing composition

This section details the approach for composing things with emphasis on how to expose capabilities of things, how to use dependencies to both connect things together and make resources available for things, and how to engage things in composition. Reasoning over dependencies is also discussed in this section.

Exposing capabilities of things

In preparation for composing things, we expose their capabilities using the concept of duty presented in [23,24]. A duty depicts what a thing can do and is specialized into atomic and composite (see Fig. 1). On the one hand, atomic duty could be either *sensing (s)* (collecting data), *actuating (a)* (processing data), or *communicating (c)* (distributing data). As per Fig. 1 that targets one thing, only, a duty is either disabled or enabled (0,1) according to the functional/non-functional requirements of the under-development IoT applications. Briefly, a thing senses the cyber–physical surrounding so, that, it produces data. A thing actuates data including those that are sensed. Finally, a thing communicates with the cyber–physical surrounding the sensed and/or actuated data. Accepting data and/or commands from peers, for example, is also taken care by the communicating duty but is not discussed further in this paper.

On the other hand, composite duty puts some atomic duties together according to one of these representative cases: *sac* (sensed data are passed on to actuating; and the data that result from actuation are passed on to communicating for distribution), *sa* (sensed data are passed on to actuating; and the data that result from actuation are finals), *sc* (sensed data are passed on to communicating for distribution), and *ac* (data that result from actuating are passed on to communicating for distribution). Additional cases of composing primitive duties include *ca* and *cas*. It is worth noting that composing duties belonging to separate things exemplifies thing composition and would require standardization between these things to address concerns like semantic and communication protocol heterogeneity. These concerns do not fall into the scope of this paper.

Table 1 includes examples of duties of some things mentioned in the case-study. Duties like *change* and *display* are atomic while others like *configure* and *remind* are composite.

Connecting things together

Composition is always associated with a model that would define how independent components would be put together to achieve joint goals. In the business process community, this model is known as business logic or process model and relies on dependencies (e.g., *prerequisite* and *parallel prerequisite*) that would ensure proper execution arrangement of the activities of a process [25]. By analogy with business process dependencies, we adopt two categories of dependencies to achieve thing composition (C_t): regular in compliance with project management's scheduling techniques[3] and special in compliance with Decker and Lesser's coordination techniques [26]. We apply these two categories of dependencies to duties (d) that they would belong to either the same thing (i.e., d_i, $d_j \in t$) or separate things (i.e., $d_i \in t$ and $d_j \in t'$) where the last case perfectly exemplifies composition of things.

Regular dependencies between two duties (d_i and d_j) are specialized as per Fig. 2 into *start-to-finish (sf)*, *start-to-start (ss)*, *finish-to-start (fs)*, and *finish-to-finish (ff)*:

1. *sf* (d_i, d_j): d_j cannot end until d_i begins; e.g., when the *display* duty begins on a certain date, the ongoing *dispense* duty should end.
2. *ss*(d_i, d_j): d_j cannot begin before d_i begins; e.g., when the *play* duty begins on a certain date, the additional *record* duty should begin too.
3. *fs*(d_i, d_j): d_j cannot begin before d_i ends; e.g., when the ongoing *remind* duty ends on a certain date, the *relax* duty should begin.
4. *ff* (d_i, d_j): d_j cannot end until d_i ends; e.g., when the $remind$ duty ends on a certain date, the ongoing *share* duty should end too.

Special dependencies between two duties (d_i and d_j) are specialized into *facilitate*, *constrain*, *enable*, *cause*, *inhibit*, and *cancel*. According to Decker and Lesser, these dependencies in the context of multiagent systems permit to avoid redundant activities, to shift activities to idle executors, and to provide predictive results. For instance, if a software agent ''realizes'' that after some reasoning that executing an activity would *facilitate* the work of a peer, then the agent would expedite the execution of this activity in support of the peer. To fit the special dependencies into thing composition, we adopt them from a resource perspective where a resource would first, be either hard (e.g., sensor, remote control, and light switch) or soft (e.g., data, CPU time, and bandwidth) and second, be used during the execution of duties.

1. *facilitate*(d_i, d_j): by establishing this dependency, some outcomes of executing d_i are to reinforce the availability of some resources for d_j , should d_j become executed. Example of reinforcement could be extra bandwidth, should d_j be of type communicating.
2. *constrain*(d_i, d_j): by establishing this dependency, some outcomes of executing d_i are to restrict the availability of some resources for d_j , should d_j become executed. Example of restriction could be starting and ending time-period of collecting data from a sensor, should d_j be of type sensing.
3. *enable*(d_i, d_j): by establishing this dependency, some outcomes of executing d_i are to unlock

[3] https://project-management-knowledge.com/definitions

the availability of some resources for d_j, should d_j become executed. Example of unlock could be a token for updating data, should d_j be of type actuating.

4. *cause(*d_i, d_j *)*: by establishing this dependency, some outcomes of executing d_i are to change the features of some resources initially assigned for the execution of d_j, should d_j become executed. Example of change could be CPU-speed increase, should d_j be of type actuating.

5. *inhibit* (d_i, d_j): by establishing this dependency, some outcomes of executing d_i are to reduce the availability of some resources for d_j, should d_j become executed. Example of reduction could be less bandwidth, should d_j be of type communicating.

6. *cancel*(d_i, d_j): by establishing this dependency, some outcomes of executing d_i are to lock the availability of some resources for d_j, should d_j become executed. Example of lock could be withdrawal of data-sharing token, should d_j be of type communicating.

From a design perspective, we first, connect things' duties together using regular dependencies and then, look into how special dependencies could either reinforce resource availabilities (Section 3.3) or handle resource unavailabilities (Section 4.2) for the benefit of these duties. It happens that a duty that executes before another duty uses all the resources, which would inhibit the execution of the other duty. Simply put, we use regular dependencies to ''craft'' the execution chronology of things' duties and special dependencies to manage these resources that these duties would use during execution. To avoid raising violations about resources, we assume that all special dependencies are subject to conditions that are not discussed further in this paper.

Reasoning over dependencies

Reasoning over dependencies becomes critical when planning the availability of resources that duties ($d_{i,j,k,...}$) use during execution. In compliance with our previous work on social coordination of business processes [27], we associate this availability with four properties, limited (l), limited-but-extensible (lx), shareable (s), and non-shareable (ns), and define two time intervals:

- [b, e] corresponds to a pre-defined interval defining the availability time of a resource where b, $e \in \mathbf{N}^+$.
- [α^{ui}, β^{ui}] corresponds to a pre-defined interval defining the requested time of using a resource by a duty d_i where α^{ui}, $\beta^{ui} \in \mathbf{N}^+$. Ideally, we would like to have [α^{ui}, β^{ui}] $\subseteq$ [b, e] but this might not always be the case due to risks of resource unavailabilities.

Let us analyze the relationship between resource properties and availability-time/use-time intervals (Fig. 3) where resource use u_i is associated with duty d_i.

1. limited (l[b, e], Fig. 3-a): any first-time resource use (u_i) happening whether concurrently to other ongoing uses (e.g., u_1 and u_2) or sequentially after previous uses (e.g., u_4 and u_5) is associated with [$\alpha^{u}i$, $\beta^{u}i$] where $\alpha^{u}i \geq b$ and $\beta^{u}i \leq e$. Should there be additional resource uses, e.g., once with $u_{1.1}$ and twice with $u_{3.1}$ and $u_{3.2}$, then any new use-time interval [$\hat{\alpha}^{u}i$, $\hat{\beta}^{u}i$] for an additional resource use should fall into [b, e]. Otherwise, the additional resource use would be rejected by the resource's provider. For

instance, all notifications whether regular or special from the $display$ duty should happen during a 15-min interval from the time of administering medicine.

2. limited-but-extensible (lx[b, e], Fig. 3-b): any first-time resource use (u_i) happening whether concurrently to other ongoing uses (e.g., u_1 and u_2) or sequentially after previous uses (e.g., u_4 and u_5) is associated with [$\alpha^{u}i$, $\beta^{u}i$] where $\alpha^{u}i \geq b$ and $\beta^{u}i \leq e$. Should there be a need for additional resource uses, then any new use-time interval [$\hat{\alpha}^{u}i$, $\hat{\beta}^{u}i$] for an additional resource use should fall into [b, e 0[+δ] $*$] where δ represents the extended availability time and is set at the discretion of the resource's provider, and 0[...] $*$ denotes *zero-2-many* repetitions. Otherwise, the additional resource use is rejected by the resource's provider. For instance, any alarm that is set by the *dispense* duty should last for at least 5 min with the option of extending the alarm by 2 min, if necessary.

3. shareable (s) and non-shareable (ns) would be mixed with limited and limited-but-extensible, as we see fit. Unless stated, all resources are shareable allowing their concurrent uses, e.g., ([$\alpha^{u}i$, $\beta^{u}i$] $\subseteq$ [b, e] and [$\alpha^{u}j$, $\beta^{u}j$] $\subseteq$ [b, e]). For instance, any sensitive video/audio content that the *play* duty displays is kept private from the *share* duty.

We now discuss the reasoning over dependencies using some expressions associated with the availability and use of resources (Table 2). This reasoning is illustrated with Table 1's duties.

***start-to-finish*(d_i, d_j):** after d_j execution is complete (i.e., d_j (r[$\alpha^{u}j$, $\beta^{u}j$])), d_i could be concerned about the post availability-time of a resource r since $p.r(dj \rightarrow di)$ could be **either**

1. l meaning that either d_i(r[$\alpha^{u}i$, $\beta^{u}i$]) $\nsubseteq$ r[$\beta^{u}j$, e] or d_i(r[$\hat{\alpha}^{u}i$, $\beta^{u}i$]) $\nsubseteq$ r[$\beta^{u}i$, e].

or

2. lx meaning that either d_i(r[$\alpha^{u}i$, $\beta^{u}i$]) $\nsubseteq$ r[$\beta^{u}j$, e 0[+ δ] $*$] or d_i(r[$\hat{\alpha}^{u}i$, $\beta^{u}i$]) $\nsubseteq$ r[$\beta^{u}i$, e 0[+ δ] $*$]

then, the reasoning is to ensure that d_k is included in C_t and to execute d_k before d_i starts so, that,

1. should *facilitate*(d_k, d_i) hold, then d_k's execution outcomes would allow to reinforce the availability time of the resource (r[$\beta^{u}j$, e]) that d_j would have left for d_i depending on this resource's property:

(a) l would require ensuring that first, d_i(r[$\alpha^{u}i$, $\beta^{u}i$]) $\subseteq$ r[$\beta^{u}j$, e] and then, d_i(r[$\hat{\alpha}^{u}i$, $\beta^{u}i$]) $\subseteq$ r[$\beta^{u}i$, e], if necessary. Otherwise, r's unavailability would be handled as per Section 4.2.
(b) lx should require adjusting r[$\beta^{u}j$, e] in a way that d_i(r[$\alpha^{u}i$, $\beta^{u}i$]) $\subseteq$ r[$\beta^{u}j$, e 0[+ δ] $*$].

2. should *enable*(d_k, d_i) hold, then d_k's execution outcomes would allow to unlock a new resource (nr to be similar to r) for d_i, if the availability time of the resource (r[$\beta^{u}j$, e]) that d_j would have left for d_i cannot accommodate its use time. Confirming this unlock would mean the following as per this new resource's property:

(a) l would require ensuring that d_i(nr[$\alpha^{u}i$, $\beta^{u}i$]) $\subseteq$ nr[b, e].

(b) lx would require ensuring that $d_i(nr[\alpha^u i, \beta^u i]) \subseteq nr[b, e]$ and then, $d_i(nr[\hat{\alpha}^u i, \beta^u i]) \subseteq nr[\beta^u i, e\ 0[+\delta] *]$, if necessary.

To illustrate *enable*, let us assume *sf* (*display*, *dispense*) and consider the dispenser's screen as a lx resource (r). Should the *dispense* duty last more than expected, e.g., $dispense(r[5, 7{+}2])$, this would make the dispenser's screen's availability- time interval, e.g., $r[7, 9]$, inappropriate for the *display* duty whose use-time interval is set at [7, 10]. Because of *enable*(*play*, *display*), the $play$ duty would make the smart TV's screen (nr) available for the *display* duty in a way that *display* 's use-time interval would be accommodated, e.g., $display(nr[7, 10]) \subseteq nr[7, 12]$.

3. should *cause*(d_k, d_i) hold, then d_k's execution outcomes would allow to change the initial availability-time ($r[b, e]$) of a resource that was assigned to d_i, with the assumption that this availability-time would not satisfy d_i's use- time requirement after d_j execution is complete. Regardless of this resource's property, l or lx, the new time availability $r[b', e']$ would allow to satisfy $d_i(r[\alpha^u i, \beta^u i]) \subseteq r[b', e'\ 0[+\delta] *]$ and $d_i(r[\hat{\alpha}^u i, \beta^u i]) \subseteq r[b', e'\ 0[+\delta] *]$.

start-to-start **(**d_i**,** d_j **):** before d_i and d_j simultaneously start their executions, both could be concerned about the pre availability-time of a resource r that they would share during these executions ($d_i(r[\alpha^u i, \beta^u i]) \subseteq r[b, e]) \wedge d_j(r[\alpha^u j, \beta^u j]) \subseteq r[b, e]$ with $\alpha^u i = \alpha^u j$) since $p.r$ could be **either**

1. l meaning that (either $d_i(r[\alpha^u i, \beta^u i]) \nsubseteq r[b, e]$ or $d_i(r[\hat{\alpha}^u i, \beta^u i]) \nsubseteq r[b, e]$) and (either ($d_j(r[\alpha^u j, \beta^u j]) \nsubseteq r[b, e]$ or $d_j(r[\hat{\alpha}^u j, \beta^u j]) \nsubseteq r[b, e]$).

or

2. lx meaning that (either $d_i(r[\alpha^u i, \beta^u i]) \nsubseteq r[b, e\ 0[+\delta] *]$ or $d_i(r[\hat{\alpha}^u i, \beta^u i]) \nsubseteq r[b, e\ 0[+\delta] *]$) and (either ($d_j(r[\alpha^u j, \beta^u j]) \nsubseteq r[b, e\ 0[+\delta] *]$ or $d_j(r[\hat{\alpha}^u j, \beta^u j]) \nsubseteq r[b, e\ 0[+\delta] *]$).

then, the reasoning is to ensure that d_k is included in C_t and to execute d_k before d_i/d_j simultaneously start so, that,

1. should *facilitate*(d_k, d_i/d_j) hold, then d_k's execution outcomes would allow to reinforce the availability time of the resource ($r[b, e]$) that d_i and d_j would use after their simultaneous executions start. Reinforcing this availability would depend on this resource's property:

(a) l would require ensuring that (either $d_i(r[\alpha^u i, \beta^u i]) \subseteq r[b, e]$ or $d_i(r[\hat{\alpha}^u i, \beta^u i]) \subseteq r[b, e]$) and (either $d_j(r[\alpha^u j, \beta^u j]) \subseteq r[b, e]$ or $d_j(r[\hat{\alpha}^u j, \beta^u j]) \subseteq r[b, e]$). Otherwise, r's unavailability would be handled as per Section 4.2.
(b) lx should require adjusting $r[b, e]$ in a way that (either $d_i(r[\alpha^u i, \beta^u i]) \subseteq r[b, e\ 1[+\delta] *]$ or $d_i(r[\hat{\alpha}^u i, \beta^u i]) \subseteq r[b, e\ 1[+\delta] *]$) and (either $d_j(r[\alpha^u j, \beta^u j]) \subseteq r[b, e\ 1[+\delta] *]$ or $d_j(r[\hat{\alpha}^u j, \beta^u j]) \subseteq r[b, e\ 1[+\delta] *]$).

To illustrate *facilitate*, let us assume *ss*(*play*, *record*) and consider bandwidth as a l resource (r). Should this resource's availability-time interval, e.g., $r[2, 5]$, does not suit these duties after their simultaneous start, e.g., $play(r[3, 7])$ and $record(r[3, 7])$, this would make completing their executions at risk. Because of *facilitate*(*trigger*, *play* /*record*), the execution of *trigger* duty would happen allowing to free additional availability time of the resource for the benefit of both *play* and *record* duties in a way that their respective use-time intervals would be accommodated, e.g., $play(r[3, 7]) \subseteq r[3, 7]$ and $record(r[3, 7]) \subseteq r[3, 7]$.

2. should *enable*(d_k, d_i/d_j) hold, then d_k's execution outcomes would allow to unlock a new resource (nr to be similar to r) for the benefit of d_i/d_j , if the availability time of the resource ($r[b, e]$) that d_i/d_j would use cannot accommodate their respective use time, once their simultaneous times start. Confirming this unlock would mean the following as per this new resource's property:

1. I would have to ensure that (either $d_i(nr[\alpha^u i\ ,\ \beta^u i\ ]) \subseteq nr[b, e]$ or $d_i(nr[\hat{\alpha}^u i\ ,\ \beta^u i\ ]) \subseteq nr[b, e]$) and (either $d_j\ (r[\alpha^u j\ ,\ \beta^u j\ ]) \subseteq nr[b, e]$ or $d_j\ (nr[\hat{\alpha} uj\ ,\ \beta uj\ ]) \subseteq nr[b, e]$). Otherwise, r's unavailability would be handled as per Section 4.2.

2. lx would have to ensure that (either $d_i(nr[\alpha^u i\ ,\ \beta^u i\ ]) \subseteq nr[b, e\ 0[+\ \delta]\ *]$ or $d_i(nr[\hat{\alpha}^u i\ ,\ \beta^u i\ ]) \subseteq nr[b, e\ 0[+\ \delta]\ *]$) and (either $d_j\ (r[\alpha^u j\ ,\ \beta^u j\ ]) \subseteq nr[b, e\ 0[+\ \delta]\ *]$ or $d_j\ (nr[\hat{\alpha}^u j\ ,\ \beta^u j\ ]) \subseteq nr[b, e\ 0[+\ \delta]\ *]$).

finish-to-start **(**d_i**,** d_j **):** should d_j be concerned about the post availability-time of a resource after d_i execution is complete, then the reasoning is to ensure that d_k is included in C_t and to execute d_k before d_j starts so, that,

- should *facilitate*(d_k, d_j) hold, then d_k's execution outcomes would allow to reinforce the resource's availability time that d_i would have left for the benefit of d_j with the assumption that these availability times were not enough for d_j .
- should *enable*(d_k, d_j) hold, then d_k's execution outcomes would allow to unlock new resources for the benefit of d_j with the assumption that d_i would have purged some (or all) of the resources that d_j would need.
- should *cause*(d_k, d_j) hold, then d_k's execution outcomes would allow to change the features of the resource for the benefit of d_j with the assumption that the resource initially assigned to d_j would not satisfy its requirements after d_i execution is complete.

To illustrate *cause*, let us assume *fs*(*remind*, *relax*) and consider CPU as a lx resource (r). Should this resource's availability-time interval, e.g., $r[3, 7]$, after executing the *remind* duty, indicate a low speed that would not be convenient during the *relax* duty's use-time interval, e.g., $relax(r[4, 6])$, this would degrade the performance of *relax* duty. Because of *cause*($post$, $relax$), the execution of $post$ duty would happen allowing to overclock the resource (speed increase) for the $relax$ duty in a way that its execution would be complete during its use-time interval, e.g., $relax(r[4, 6]) \subseteq r[3, 7]$.

finish-to-finish**(**d_i**,** d_j **):** should d_i/d_j be concerned about the ongoing availability-time of a resource before its simultaneous execution-end with d_j /d_i, then the reasoning is to ensure that d_k is included in C_t and to execute d_k before d_i/d_j complete so, that,

- should *facilitate*(d_k, d_i/d_j) hold, then d_k's execution outcomes would allow to reinforce the resource's availability time for the benefit of d_i/d_j with the assumption that d_i/d_j would share the resource, so, that, both would finish simultaneously.

- should *enable*(d_k, d_i/d_j) hold, then d_k's execution outcomes would allow to unlock new resources for the benefit of d_i/d_j with the assumption that d_j /d_i would use the resource that d_i/d_j would need so, that, both would finish simultaneously.
- should *cancel*(d_k, d_i/d_j) hold, then d_k's execution outcomes would allow to lock the availability of the resource for the benefit of d_j /d_i with the assumption that d_i/d_j would need this resource, so, that, both would finish simultaneously. To illustrate *cancel*, let us assume *ff* (*remind*, *share*) and consider data-sharing token as a lx resource (r). Should this resource become obsolete during its availability time, e.g., r[2, 7], and ongoing execution of *remind* and *share* duties, e.g., $remind(r[3, 8])$ and $share(r[5, 8])$, this would make the simultaneous execution-ends of these duties at risk. Because of *cancel*(*post*, *remind*/*share*), the execution of *post* duty would allow to extend the resource's availability time and hence, its validity for the benefit of *remind*/*share* duties in a way that their use-time intervals would be accommodated, e.g., $remind(r[3, 8]) \subseteq r[2, 9]$ and $share(r[5, 8]) \subseteq r[2, 9]$.

Engaging things in composition

Fig. 4 illustrates the four modules that would support the participation of component things (T_i) in composite things (CT_j). These modules are composer, reasoner, monitor, and executor interacting with three repositories, trace, dependency, and resource, and some run-time platforms upon which the composite things will be deployed for execution. In this figure, numbers correspond to the chronology of operations.

It all starts when an IoT engineer specifies the component things in terms of duties and dependencies between duties (0 & 1). Standards like the Web of Things (WoT) Thing Description (WoT-TD, [28]) could be used for specifying things, but this does not fall into the scope of this paper. Next, the composer module screens the available component things (2) so, that, it matches their atomic/composite duties with the needs of users as per the interactions the users would have had with the IoT engineer when expressing their needs. The matching leads to the development of composite things whose execution chronologies would refer to regular dependencies between the duties of the component things that have been selected to participate in these composite things. Upon the user's approval of the definition of a composite thing in terms of component things and execution chronology, the composer module ''transfers'' the composite thing to the composite layer (3) along with informing the executor module of the readiness of this composite thing for deployment on the run-time platforms (4). Prior to initiating the deployment, the executor module ensures that the component things of this composite thing have the necessary resources to execute their duties. To this end, the executor module submits the composite thing's execution chronology (5) to the reasoner module that consults the resource repository (6) (i.e., resources' availability times) and reasons over this chronology's regular dependencies (7) in order to identify the special dependencies (e.g., *enable* and *facilitate*) that could be deemed necessary for executing these duties. Should some special dependencies hold between the duties (duties' time uses *versus* resources' availability times), the reasoner module notifies both the executor module and the composer module about these special dependencies (8) so, that, the necessary changes in the composite thing's execution chronology are made as per Section 3.3. All these interactions happen under the supervision of the IoT engineer/user who are made aware of the needs of duties of resources as well as how these resources are secured thanks to the special dependencies that would allow to release more resources from a availability-time perspective, for example. After the necessary changes are made in the composite thing's execution chronology, the executor module initiates the composite thing, which means invoking the duties of the component things participating in this composite thing (9 to 12).

During invocation, the executor module tracks the execution progress of the composite thing (13) along with asking the monitor module to consult the trace repository where details about this progress are stored. These details are about which duties are executed, which resources are used, and which resources are requested. The objective of consulting the trace repository is to analyze the execution traces of composite things (14) and notify both the reasoner module and the IoT engineer/user of any dependency violation (15). This violation could have many reasons like non-implementation of a special dependency, which has led to resource unavailability for some duties. More details about this unavailability are presented in Section 4.2.

Thing composition enhancement

This section discusses how to weave social relations into thing composition for the needs of thing identification and how to handle resource unavailabilities despite special dependencies.

Making things socialize

In Fig. 4, operation (2) is about the composer module that takes care of identifying the necessary things according to their atomic/composite duties and users' needs. Usually, this identification is known as discovery that could benefit from potential social relations between things [29] and trust between things as well [30]. In the context of the Internet of Social Things (IoST), Atzori et al. mention that models used to study social networks of humans can be extended to social networks of objects/things [20,31]. These networks could be built upon relations such as parental (similar objects built in the same period by the same manufacturer), co-location (objects in the same venue), co-work (objects participating in the same scenario), ownership (objects having the same user), and social (when objects come into contact sporadically or continuously). Atzori et al. also mention the paradigm shift that is happening from human–object interaction to object–object interaction. Based on our previous work on weaving social computing into IoT [32], we adopt three social relations namely *complementary* exposing recommendation between things, *antagonism* exposing opposition between things, and *competition* exposing exclusion between things.

1. Complementary(t_i, t_j) refers to the ''joint'' participation of things, e.g., smart TV and remote control, in satisfying users' demands (ud). Eq. (1) assesses the complementary level between t_i and t_j where $accepted\ Rec_{ud}\,(t_i, t_j)$ is the number of times that t_i's recommendations for t_j are accepted by the IoT engineer and $made\ Rec_{ud}\,(t_i, t_j)$ is the number of times that t_i recommended t_j (including the declined recommendations, $declined\ Rec_{ud}\,(t_i, t_j)$).
2. Antagonism(t_i, t_j) refers to the ''sensitivity'' that exists between things, e.g., coffee machine and espresso machine, when they jointly participate in satisfying users' demands. Eq. (2) assesses the

antagonism level between t_i and t_j where $joint_{ud}$ (t_i, t_j) is the number of times that t_i and t_j jointly participated in satisfying users' demands and $participated_{ud}$ (t_i | ¬t_j) is the number of times that t_i participated in satisfying users' demands without the participation of t_j in these demands and *vice versa*.

3. Competition(t_i, t_j) refers to the ''exclusion'' between things, e.g., either cordless phone or regular phone, as one thing, only, can participate in satisfying a user's demand. Eq. (3) assesses the competition level between t_i and t_j where $selected_{ud}$ (t_i, t_j) is the number of times that t_i is selected over t_j to participate in satisfying users' demands and $possible_{ud}$ (t_i, t_j) is the number of times that both t_i and t_j are potential candidates for participation in satisfying users' demands.

The aforementioned social relations could impact the execution chronology of the composite thing as per the analysis below:

1. Regular dependencies: tapping into the complementary relation could mean inserting new things into the under-development composition of things, which means connecting the new things' duties to those that are already in the composition using either *start-to-start*, *start-to-finish*, *finish-to-finish*, or *finish-to-start* dependency. The IoT engineer is responsible for the connection, as he sees fit. Prior to confirming the insertion, it is recommended to tap into the antagonism relation to avoid any potential ''frictions'' between the newly inserted things and existing things. These ''frictions'' mean conflicts between things and hence, the IoT engineer could decide of not inserting some things into the composition if the antagonism level is above a threshold (Eq. (2)).
2. Special dependencies: because some special dependencies like *enable* and *facilitate* require invoking new duties that could belong to things that are not already included in the under-development composition of things, it is recommended to tap into the antagonism relation to avoid any potential ''frictions'' between the newly included things because of their duties and existing things. Once the newly included things are confirmed, the IoT engineer connects their respective duties to other things' duties using regular dependencies, as he sees fit. It is also recommended to tap into the competition relation, should a necessary duty be offered by many similar competing things and thus, only one thing should be selected.

Handling resource unavailabilities

In Section 3.3, we illustrated the role of regular dependencies in connecting duties of things together and special dependencies in handling resource unavailabilities and reinforcing resource availabilities. However, it happens that special dependencies do not hold (i.e., resource's availability time is neither secured nor reinforced), which confirms resource unavailability impacting the completion of thing composition. In the following, we put forward some potential solutions for handling this unavailability. To start with, we model duty and resource as state diagrams allowing to indicate with respect to specific states when a duty is put on-hold and when a resource is unavailable. The states and transitions in both diagrams are activated in a synchronized way.

In Fig. 5(a), *prepared* state signals that a thing offering a duty is under consideration for possible participation in a composition scenario. In this state, the thing checks the availability of the resources that the duty needs and then, proceeds with enabling either the *activation* transition when the check is positive (Fig. 5(b):*engaged* state) making the duty take on *activated* state or the *pause* transition when the check is negative (Fig. 5(b):*idle* state) making the duty take on *suspended* state. When the duty is in *activated* state, it could transition to either *done* state signaling the success of the duty execution, or *suspended* state signaling the necessity of handling a resource unavailability identified with Fig. 5(b): *suspended* state. Should this handling be possible (Fig. 5(b): from *suspended* state to *engaged* state), the duty transitions back from *suspended* state to *activated* state. Otherwise, the duty transitions from *suspended* state to *failed* state. In a duty's state diagram, *failed* and *done* states lead to the final state. Our objective is to avoid $suspended$ state and, particularly, $failed$ state since this latter would terminate the ongoing thing composition.

In Fig. 5(b), *idle* state signals that a resource is available for use by some things' duties. Following a request-to-use from a thing, the resource transitions to *engaged* state confirming that it is available for use. Should this use complete successfully by a duty (Fig. 5(a):*done* state), the resource transitions back to *idle* state. Otherwise, i.e., incomplete use, the resource becomes unavailable taking on *suspended* state and impacting the duty (Fig. 5(a):from *activated* state to *suspended* state). If the unavailability cannot be handled, the resource takes on *failed* state impacting the duty as well (Fig. 5(a):from *suspended* state to *failed* state). In the resource's state diagram, *idle* and *failed* states lead to the final state. Our objective is to avoid $suspended$ state and, particularly, $failed$ state since this latter would terminate the thing composition.

To mitigate the impact of resource unavailabilities on duties, we track when these unavailabilities would happen, i.e., design-time *versus* run-time, and associate duties with qualitative criteria and transactional properties as per the options below.

Option 1 refers to a set of qualitative criteria that we suggested in [33] to evaluate the criticality of things/duties in a process- of-things. These criteria are known as value-adding (va_t), business value-adding (bva_t), and non value-adding (nva_t) and are built upon similar criteria that the BP community uses [34]. For the needs of our work, we restrict ourselves to value-adding and non value-adding criteria, only.

Option 2 refers to a set of transactional properties that the ICT community uses to decide on the acceptable outcomes of executing transactions. These properties are referred to as pivot, retriable, and compensatable [35], Fig. 6. Pivot means that once an execution successfully completes, its effects remain unchanged forever and cannot be undone. Additionally, this execution cannot be retried following failure. Compensatable means that a successful execution's effects can be semantically undone. Finally, retriable means that an execution is guaranteed to successfully complete after several finite activations. For the needs of our work, we first, suggest semi-compensatable property that would require dropping *failed* state from the compensatable definition and second, combine semi-compensatable and retriable properties together.

We now illustrate the role of qualitative criteria and transactional properties in handling resource unavailabilities. To this end, we adopt two regular dependencies. Let us start with *sf* (d_i, d_j) where the potential resource unavailability impacting d_i would be known at run-time and hence, would prevent the execution of d_i.

- Should both d_i and d_j be value-adding duties, i.e., must be executed, then we recommend at design-time to consider a composite duty $cd_{i,j}$ whose component duties, e.g., cd^1 , cd^2 , $\cdots$, cd^n , would provide the same outcomes as d_j and d_i would do. Prior to launching the execution of the composite duty, we ensure that two components, e.g., cd^k and $cd^{k'}$, at least in the composite duty are connected through *sf* like d_i and d_j and that $cd^{k'}$ is semi-compensatable and *cdkij* is retriable. At run-time, the composite duty (acting as a controller) tracks the execution progress of the different component things' duties. Should $cd^{k'}$ produce resources that are inappropriate for cd^k , the composite duty would make $cd^{k'}$ take on semi-compensatable: *compensated* state so, that, this inappropriateness is handled. Then, the composite duty would initiate the execution of cd^k that will for sure succeed being retriable.
- Should d_i be a value-adding duty, i.e., must be executed, then we recommend at design-time to substitute d_j with an atomic duty d' (or composite duty $cd_{j.}$) that would provide the same outcome as d_j would do. Prior to launching the execution of the substituting duty, we ensure that its transactional property is semi-compensatable. Should d' produce resources that are inappropriate for d_i, d_j' would transition from semi-compensatable: *done* state to semi-compensatable: *compensated* state so that, this inappropriateness is handled.
- Should d_i be a non value-adding duty, then its execution would be skipped when d_j would produce inappropriate resources for d_i.

We now examine $ss(d_i, d_j)$ where the potential resource unavailability impacting both d_i and d_j would be known at design-time and hence, would prevent their executions.

- Should both d_i and d_j be both value-adding duties, i.e., must be executed, then we recommend to adjust the resource's availability time in a way that d_i's and d_j's consumption times are accommodated.
- Should both d_i and d_j be both non-value-adding duties, then their executions would be skipped when the resources made available for them would be inappropriate.

Implementation and experiments

We present the implementation work that was performed to demonstrate the technical feasibility of our time-centric, resource-driven thing-composition approach and then, discuss some experimental results.

Testbed set-up

Our testbed extends the Java-based discrete event simulator EdgeCloudSim that we deployed on top of a Toshiba dynabook with Intel Core i5-825 processor and 8GB of RAM. EdgeCloudSim is one of the most popular simulators for IoT scenarios [36]. The extension was necessary since EdgeCloudSim does not consider multiple Virtual Machines (VMs) for processing nor how tasks (duties in our work) would be connected through (regular and special) dependencies. Fig. 7 depicts the testbed architecture along with the main modules. Extending EdgeCloudSim was associated with a *resource-duty* module (not shown in this figure) that for instance, attaches dependencies (defined as Enums) to duties, isolates things from their duties, and specifies resource availabilities and duty uses of resources as time intervals. The *resource-duty* module includes three others referred to as *duty generator*, *dependency*, and *resource orchestrator*.

- *Core-simulation* module manages the simulator based on a configuration file that includes details such as resources, dependencies between duties, and time intervals related to resources and duties. This module also supports saving simulation results in Comma-Separated Value (CSV) format so, that, numerical data is extracted for plotting.
- *Networking* module does not have a direct impact on the simulation nor output results, but is required for managing the IoT communication components including devices, VMs, and routers. This module also handles transmission delays, via an internal cost-delay function, when uploading/downloading data about duties is communicated to/from the VMs.
- *Duty-generator* module produces the necessary duties with their deadlines for a given configuration working closely with the *dependency* module to attach regular/special dependencies to these duties. In addition, duties' data sizes and duties' needs in terms of resource uses are also set according to a distribution and scheduling mechanism that complies with a Poisson distribution via active/idle duty generation pattern [37]. Since the *duty-generator* module schedules the execution of duties, it also tracks their execution progress as per Section 3.4.
- *Resource-orchestrator* module makes decisions about duty execution with respect to their dependencies. In fact, this module uses the information collected from the *duty-generator* and *dependency* modules to decide on how to handle incoming duties and in what order so, that, dependency requirements are met.

Discussions

The simulation starts by an initialization stage where the *duty-generator* module creates a set of duties according to a Poisson distribution[4] via active/idle duties generation pattern,. considering both regular and special dependencies. Each created duty would have a set of parameters such as dependency type, duty start-time, duty end-time (*aka* deadline), and duty data size (includes both input and output data). These parameters are exponentially distributed based on the simulation time and number of iterations. Afterwards, the duties are sorted based on their start times, dependencies, and other details like network medium between VM resources. It is worth noting that during simulation we used almost a consistent number of regular and special dependencies as per Table 3, where each dependency would have at least 230 simulated instances.

The *duty-generator* module takes in account not, only, the duties' deadlines but, also, the dependencies associated with each duty before submitting a duty for execution on one of the available VMs. The

[4] Poisson distribution is a discrete distribution model that measures the probability of the occurrence of a given number of duties over specified time period during the simulation.

total number of VMs is fixed to 10 for each run-time. Therefore, some duties may wait some time before they get executed to meet their dependency requirements as per Fig. 8 showing the ideal execution time *versus* the actual execution time in milliseconds. The delay in accrued execution is due to the consideration of dependencies during simulation. Moreover, the duty creation and arrival rates can vary during simulation to mimic real-world scenarios where the demand on resources can vary from time to time according to things' needs. Therefore, the previous experiment has been extended to have different arrival rates of duties during simulation; the duties at the start are 100, then increased by x amount (adjustable based on the simulation scenario) till the end of the simulation time or a maximum number of duties to execute, such as 2500 as per Fig. 9.

Another observation is related to resource use during run-time with respect to duties' regular dependencies. The objective here is to simulate cases of limited resources *versus* limited-but-extensible resources. Fig. 10 shows duties grouped by their regular dependencies and the accumulative time (in milliseconds) required for those groups to extend the resources (regardless of which VM) usage during the simulation run-time. It is also worth noting that the number of duties in iteration #1 is fixed to 100 and increased by 200 until it gets to 1000 duties in the final iteration #5. In each iteration, the same ratio of duty dependencies was maintained.

To illustrate the resource consumption with respect to limited resources and limited-but-extensible resources, we observed VMs usage at a specific timestamp (randomly chosen around mid-simulation time) to verify that duties can extend their times on the VM resources when required as per Fig. 11 which shows duties running on VM2, VM4, VM5, VM7, VM8, and VM10 have extended their times to finish, unlike duties running on VM1, VM3, VM6, and VM9 that are done according to the scheduled times, hence no resource usage extension was required.

Conclusion

To sustain the rapid development of the Internet-of-Things, this paper presented a time-centric, resource-driven approach to compose things based on a set of regular and special dependencies. The former specialized into start-to-start, finish-to-finish, start- to-finish, and finish-to-start, allow to define the chronology of executing things and their duties. And, the latter specialized into facilitate, constrain, enable, cause, inhibit, and cancel, allow to tackle the challenge of resource unavailabilities during the execution of things and their duties. Resources were assigned properties known as limited, limited-but-extensible, shareable, and not-shareable having each an impact on the availability of these resources. Duties use resources at run-time according to specific time intervals. The composition approach presented in this paper consists of exposing capabilities of things as a set of duties, identifying necessary dependencies to connect things' duties together, reasoning over time intervals that depict when things' duties need to use resources, and finally, demonstrating thing composition through a system extending EdgeCloudSim.

In term of future work we would like to examine the impact of first, mixing resource properties like limited-and-extensible and shareable on the identification of the necessary dependencies and second, adopting other restrictions like privacy on thing composition in terms of what data could be collected, processed, and shared. We expect that mixing properties and adopting new restrictions would require adjusting regular and special dependencies between things. We would also like to examine the scalability of our system when a large number of things participate in composition scenarios. Resource unavailabilities could become a major concern that could delay the completion of thing composition scenarios.

References

[1] A. Ouksel, A. Sheth, Semantic interoperability in global information systems: A brief introduction to the research area and the special section, SIGMOD Rec. 28 (1) (1999).

[2] A. Bouguettaya, M. Singh, M. Huhns, Q. Sheng, H. Dong, Q. Yu, A. Ghari Neiat, S. Mistry, B. Benatallah, B. Medjahed, M. Ouzzani, F. Casati, X. Liu, H. Wang, D. Georgakopoulos, L. Chen, S. Nepal, Z. Malik, A. Erradi, Y. Wang, M. Blake, S. Dustdar, F. Leymann, M. Papazoglou, A service computing manifesto: The next 10 years, Commun. ACM 60 (4) (2017).

[3] A. Sheth, Internet of things to smart IoT through semantic, cognitive, and perceptual computing, IEEE Intell. Syst. 31 (2) (March/April 2016).

[4] A. Taivalsaari, T. Mikkonen, A roadmap to the programmable world: Software challenges in the IoT era, IEEE Softw. 34 (1) (2017).

[5] M. Weiser, The computer for the 21st century, Newslett. ACM SIGMOBILE Mob. Comput. Commun. Rev. 3 (3) (1999).

[6] Q. Wu, G. Ding, Y. Xu, S. Feg, Z. Du, J. Wang, K. Long, Cognitive internet of things: A new paradigm beyond connection, IEEE Internet Things J. 1 (2) (April 2014).

[7] H. Green, The internet of things in the cognitive era: Realizing the future and full potential of connected devices, 2015, https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WWW12366USEN.

[8] D. Androec, B. Tomas, T. Kisasondi, Interoperability and lightweight security for simple IoT devices, in: Proceedings of the Information Systems Security Conference (ISS'2017) Held in Conjunction with the 40Th Jubilee International Convention on Information and Communication Technology, Electronics, and Microelectronics, MIPRO'2017, Opatija, Croatia, 2017.

[9] DZone, The Internet of Things, Application, Protocols, and Best Practices, Tech. Rep., 2017, https://dzone.com/guides/iot-applications-protocols-and-best- practices (visited in May 2017).

[10] G. Chen, J. Huang, B. Cheng, J. Chen, A social network based approach for IoT device management and service composition, in: Proceedings of the IEEE World Congress on Services, SERVICES'2015, New York, USA, 2015.

[11] L. Li, Z. Jin, G. Li, L. Zheng, Q. Wei, Modeling and analyzing the reliability and cost of service composition in the IoT: A probabilistic approach, in: Proceedings of the IEEE 19th International Conference on Web Services, ICWS'2012, Honolulu, HI, USA, 2012.

[12] A. Parvaneh, R. Amir Masoud, J. Hamid Haj Seyyed, Service composition approaches in IoT: A systematic review, J. Netw. Comput. Appl. 120 (2018).

[13] A. Khaled, S. Helal, A framework for inter-thing relationships for programming the social IoT, in: Proceedings of WF-IoT'2018, Singapore, 2018.

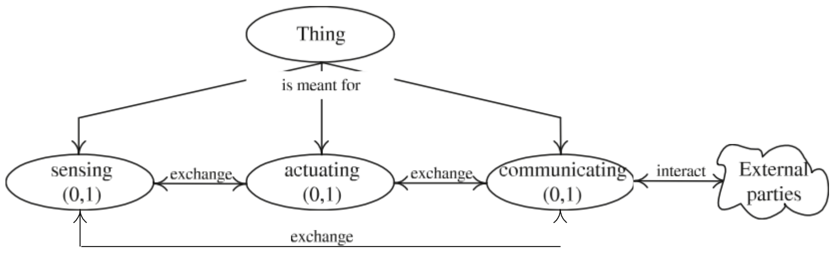
[14] A. Krishna, M. Le Pallec, R. Mateescu, L. Noirie, G. Salaün, IoT composer: Composition and deployment of IoT applications, in: Companion Proceedings of the 41st International Conference on Software Engineering, ICSE'2019, Montreal, QC, Canada, 1999.

[15] Z. Maamar, K. Boukadi, B. Koné, D. Benslimane, S. Elnaffar, Thingsourcing to enable iot collaboration, in: Proceedings of the 29th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'2020), France (online), 2020.

[16] J. Middleton, Long live the thing! temporal ubiquity in a smart vintage wardrobe, Ubiquity:J. Pervas. Media 1 (1) (September 2012).

[17] J. Phuttharak, S. Loke, A review of mobile crowdsourcing architectures and challenges: Toward crowd-empowered internet-of-things, IEEE Access 7 (2019).
[18] A. Åkesson, G. Hedin, M. Nordahl, B. Magnusson, ComPOS: Composing oblivious services, in: Proceedings of the Third International Workshop on Mobile and Pervasive Internet of Things, PerIoT’2019, Kyoto, Japan, 2019.
[19] R. Seiger, R. Kühn, M. Korzetz, U. Aßmann, HoloFlows: Modelling of processes for the internet of things in mixed reality, Softw. Syst. Model. (2021 (forthcoming)).
[20] L. Militano, M. Nitti, L. Atzori, A. Iera, Enhancing the navigability in a social network of smart objects: A Shapley-value based approach, Comput. Netw. 103 (2016).
[21] N. Hussain, H. Wang, C. Buckingham, X. Zhang, Software agent-centric semantic social network for cyber-physical interaction and collaboration, Int. J. Softw. Eng. Knowl. Eng. 30 (6) (2020).
[22] C. Perera, C. Liu, S. Jayawardena, M. Chen, A survey on internet of things from industrial market perspective, IEEE Access 2 (2014).
[23] Z. Maamar, T. Baker, M. Sellami, M. Asim, E. Ugljanin, N. Faci, Cloud *versus* edge: who serves the internet-of-things better?, Internet Technology Letters, Wiley 1 (5) (2018).
[24] A. Qamar, A. Muhammad, Z. Maamar, T. Baker, S. Saeed, A quality-of-things model for assessing the internet-of-thing’s non-functional properties, Transactions on Emerging Telecommunications Technologies (2019) (forthcoming).
[25] M. Weske, Business Process Management - Concepts, Languages, Architectures, second ed., Springer, 2012.
[26] K. Decker, V. Lesser, Generalizing the partial global planning algorithm, Int. J. Coop. Inf. Syst. 1 (2) (1992).
[27] Z. Maamar, N. Faci, S. Sakr, M. Boukhebouze, A. Barnawi, Network-based social coordination of business processes, Information Systems 58 (2016).
[28] W3C, Web of things (WoT) thing description, 2020, (Visited January 2021), www.w3.org/TR/wot-thing-description.
[29] G. Thangavel, M. Memedi, K. Hedström, A systematic review of social internet of things: Concepts and application areas, in: Proceedings of the 25th Americas Conference on Information Systems, AMCIS’2019, Cancun, Mexico, 2019.
[30] M. Aslam, S. Din, J. Rodrigues, A. Ahmad, G. Choi, Defining service-oriented trust assessment for social internet of things, IEEE Access 8 (2020).
[31] L. Atzori, A. Iera, G. Morabito, M. Nitti, The social internet of things (sIoT)-when social networks meet the internet of things: Concept, architecture and network characterization, Comput. Netw. 56 (16) (2012).
[32] K. Boukadi, N. Faci, Z. Maamar, E. Ugljanin, M. Sellami, T. Baker, M. Al-Khafajiy, Norm-based and commitment-driven agentification of the internet of things, Internet of Things 6 (2019).
[33] Z. Maamar, N. Faci, E. Kajan, S. Purkovic, E. Ugljanin, Process-of-things: weaving film industry’s practices into the internet-of-things, Internet of Things 11 (2020).
[34] M. Dumas, M. La Rosa, J. Mendling, H. Reijers, Fundamentals of Business Process Management, Springer, 978-3-642-33142-8, 2013.
[35] M. Little, Transactions and web services, Commun. ACM 46 (10) (2003).
[36] C. Sonmez, A. Ozgovde, C. Ersoy, EdgeCloudSim: An environment for performance evaluation of edge

computing systems, in: Proceedings of the International Conference on Fog and Mobile Edge Computing, FMEC'2017, Valencia, Spain, 2017.
[37] J. Gart, The Poisson distribution: The theory and application of some conditional tests, in: G.P. Patil, S. Kotz, J.K. Ord (Eds.), Statistical Distributions in Scientific Work, Vol. 2, 1975.

[38] 

[39] **Fig. 1.** Atomic duties of a thing.

[40] *Source:* Adopted from [23].

Table 1

Examples of duties for different things.

Thing	Duty type	Duty name	Description
Smart Dispenser	a	*Change*	Modify pill's intake frequency
	sa	*Configure*	Select a container per pill's type
Smart TV			
Smart Phone			

ac	*Dispense*	Release a pill in a Container and blink lights
c	*Display*	Show quantity of pills left per container
a	*Refill*	Request for more pills
a	*Play*	Enable program for viewing
a	*Record*	Tape ongoing program
a	*Trigger*	Initiate voice control
c	*Share*	Send details to the cable TV company
sc	*Remind*	Record pill’s intake and notify for next intake
c	*Post*	Present content on the screen
sa	*Relax*	Adjust luminosity and play music

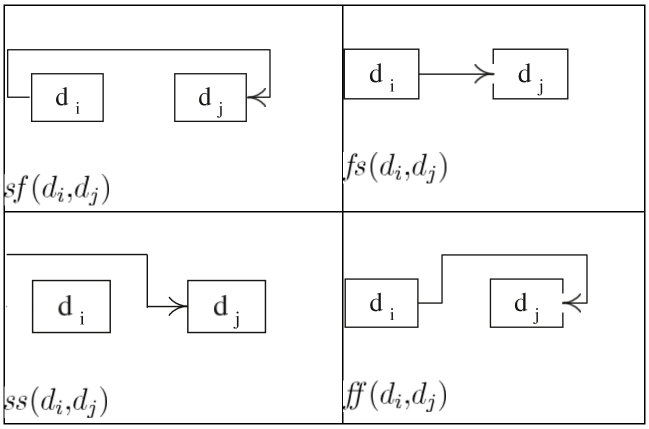
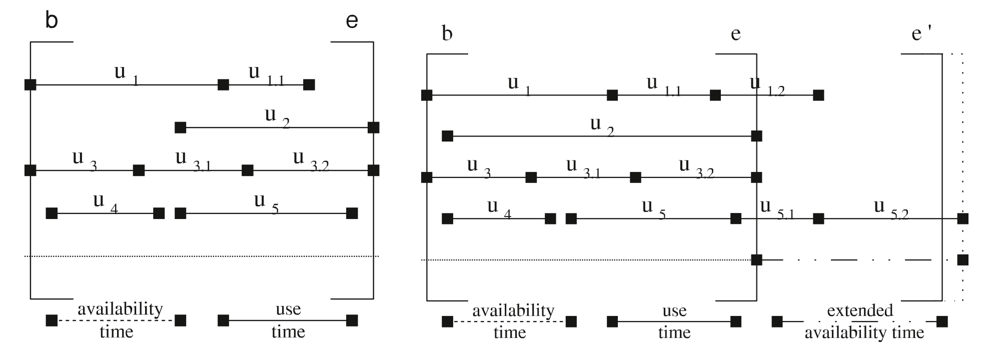


Fig. 2. Representation of regular dependencies.



(a) limited (b) limited-but-extensible

Fig. 3. Representation of a resource in terms of availability-time interval and use-time interval.

Table 2

Expressions associated with availability and use of resources.

Expression	Description
$p.r$	r's property (either l or lx)
$r[b, e]$	r's availability-time interval
$d_i(r[\alpha^{u_i}, \beta^{u_i}])$	r's use-time interval upon the request of d_i
$r(d_i \overset{left}{\longleftarrow\!\!\times} d_j)$	d_i has left r for use by d_j
$r(d_i \| d_j)$	d_i and d_j concurrently use r

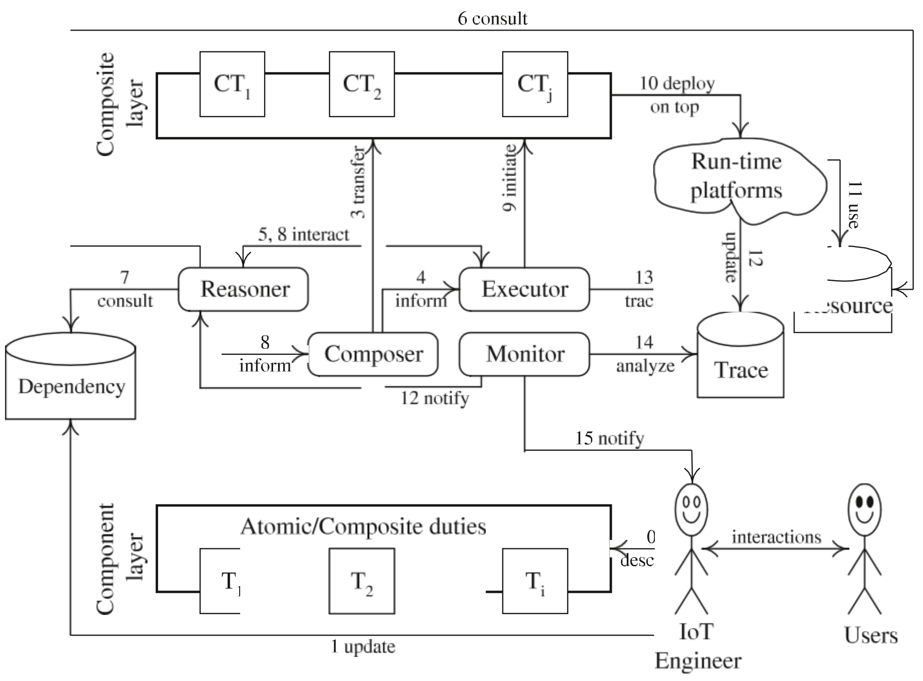


Fig. 4. Modules, repositories, and platforms to support thing composition.

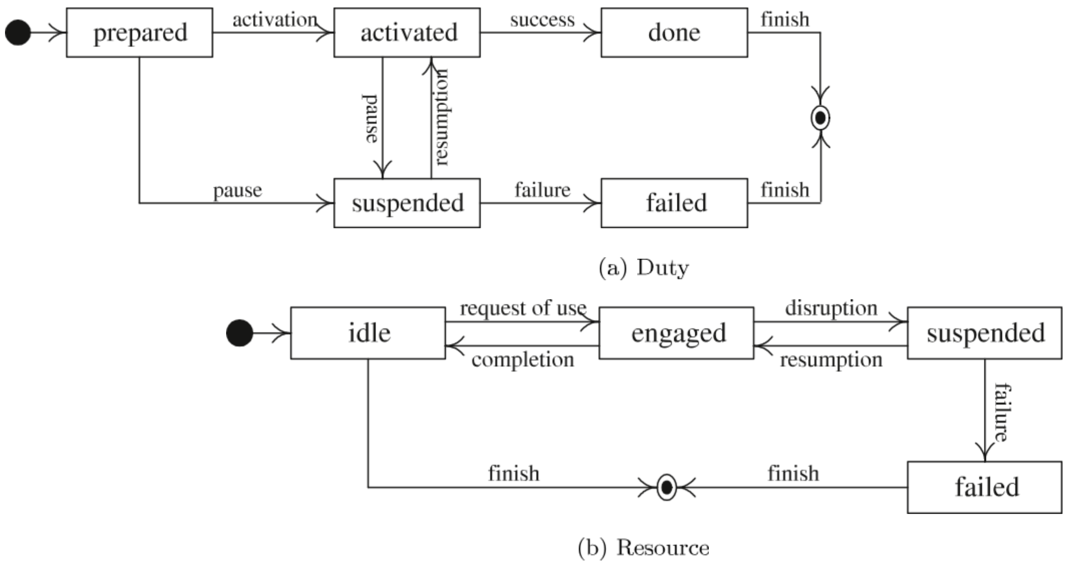


Fig. 5. Representation of duty and resource as state diagrams.

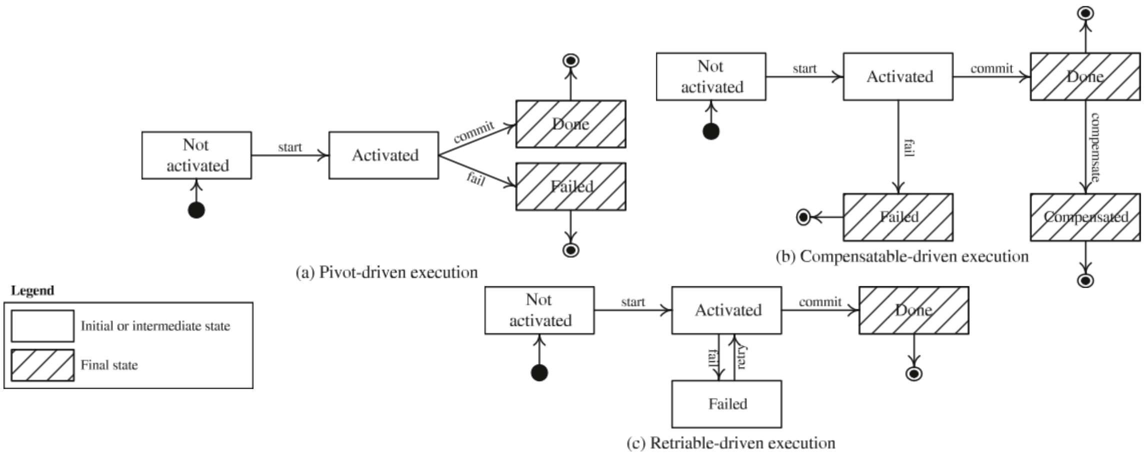


Fig. 6. Execution's life cycle per transactional property.

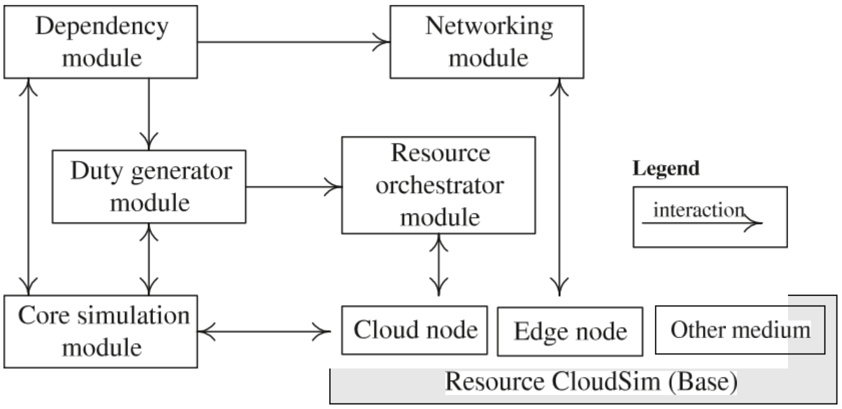


Fig. 7. EdgeCloudSim-based testbed's main modules.

Table 3

Number of regular and special dependencies during simulation.

Type of dependencies	Number of dependencies
ss	264

Regular

sf	271
fs	244

	ff	240
Facilitate		250
Constrain		232

Special

Enable	261
Cause	239
Inhibit	258

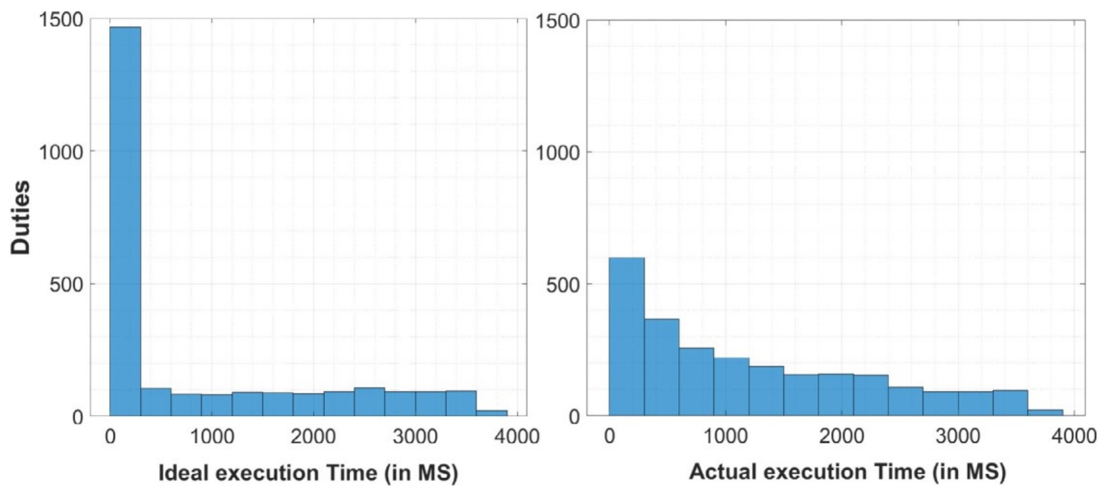


Fig. 8. Ideal-execution time *vs.* Actual-execution time in milliseconds.

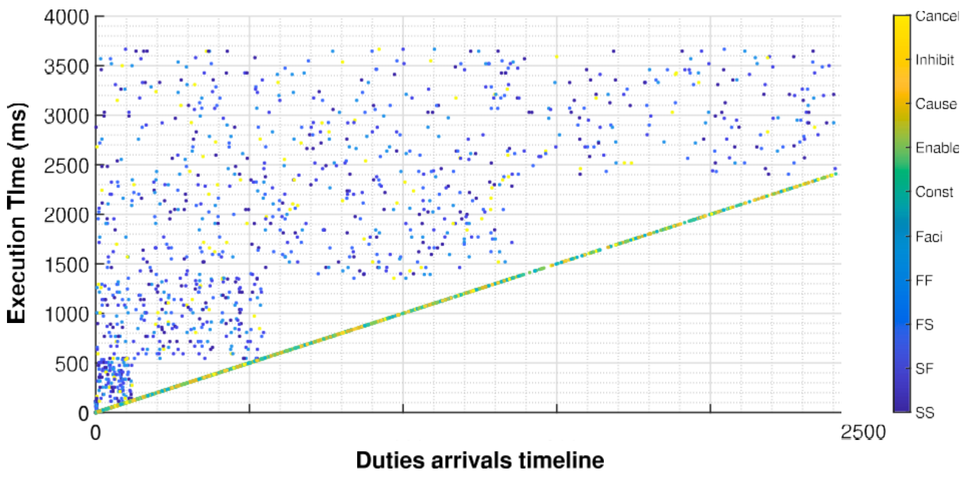


Fig. 9. Duties' arrival and execution times.

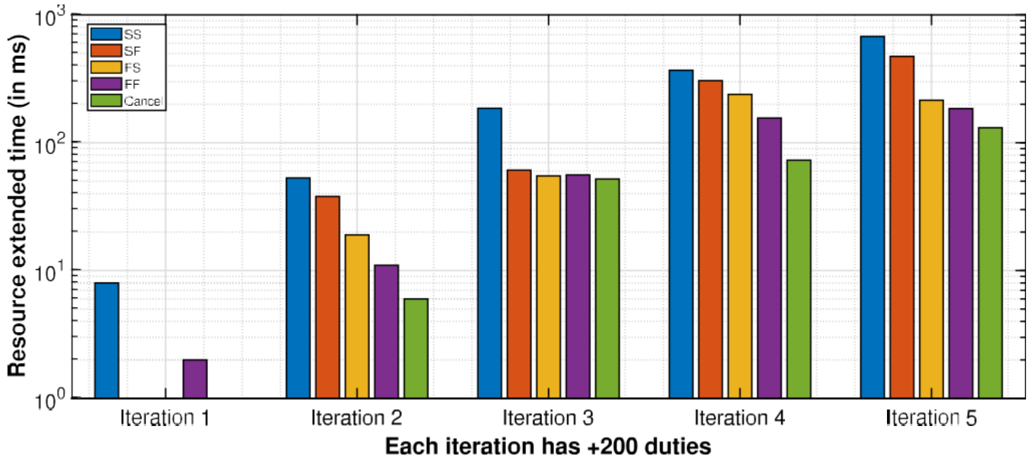


Fig. 10. Resource extended times per type of regular dependency.

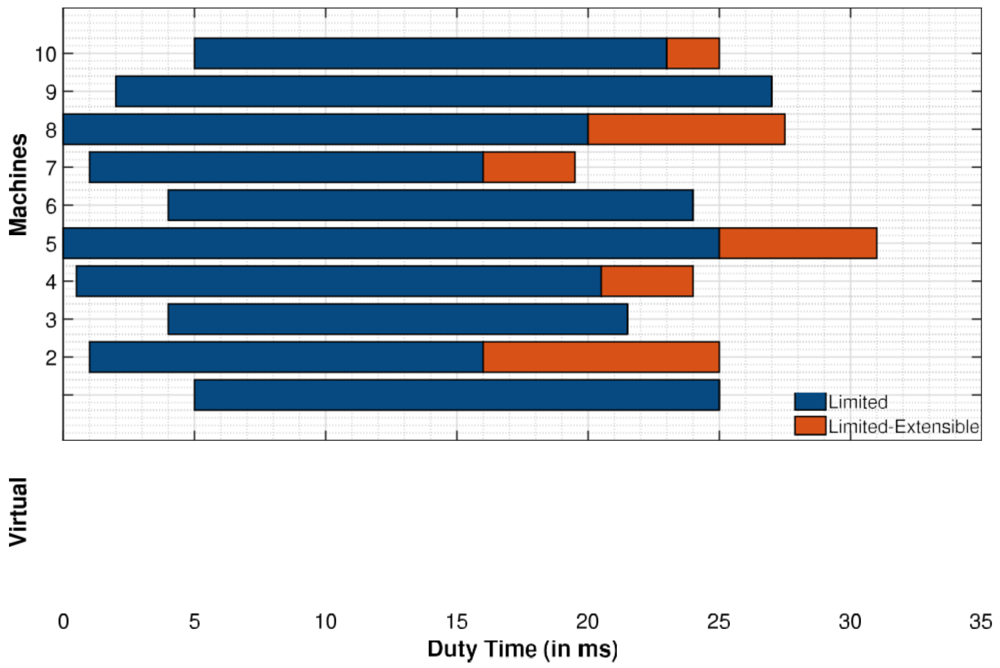


Fig. 11. Resource extended time at a certain timestamp during the execution.