An Ontological Architecture for Principled and Automated System of Systems Composition

Abdessalam Elhabbash, Vatsala Nundloll,
Yehia Elkhatib, Gordon S. Blair
a.elhabbash@lancaster.ac.uk
SCC, Lancaster University, UK

Vicent Sanz Marco
FCRL, Osaka University, Japan

ABSTRACT

A distributed system's functionality must continuously evolve, especially when environmental context changes. Such required evolution imposes unbearable complexity on system development. An alternative is to make systems able to self-adapt by opportunistically composing at runtime to generate systems of systems (SoSs) that offer value-added functionality. The success of such an approach calls for abstracting the heterogeneity of systems and enabling the programmatic construction of SoSs with minimal developer intervention. We propose a general ontology-based approach to describe distributed systems, seeking to achieve abstraction and enable runtime reasoning between systems. We also propose an architecture for systems that utilize such ontologies to enable systems to discover and 'understand' each other, and potentially compose, all at runtime. We detail features of the ontology and the architecture through two contrasting case studies. We also quantitatively evaluate the scalability and validity of our approach through experiments and simulations. Our approach enables system developers to focus on high-level SoS composition without being tied down with the specific deployment-specific implementation details.

CCS CONCEPTS

• **Computer systems organization → Architectures**; • **Computer systems organization Self-organizing autonomic computing**;

KEYWORDS

self-adaptation; context awareness; ontology; runtime composition; system of systems

ACM Reference Format:
Abdessalam Elhabbash, Vatsala Nundloll, Yehia Elkhatib, Gordon S. Blair and Vicent Sanz Marco. 2020. An Ontological Architecture for Principled and Automated System of Systems Composition. In . ACM, New York, NY, USA, 11 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Computing systems have evolved from the basic picture of connected PCs and mobile devices to the complicated picture of the inter-connected collections of heterogeneous systems. These systems include Internet of Things (IoT), clouds and micro-clouds, ad-hoc networks (MANETS, VANETs, FANETs), smart grids, among others. Each of these systems contains a set of heterogeneous components that implement diverse functions communicating using different protocols. Furthermore, these various systems often need to interact and collaborate to achieve their goals. For example, isolated rescue teams need to opportunistically compose and access each others services (*e.g.,* data look-up) to exchange information; deployed environmental IoT systems process their data on a micro-cloud in their vicinity and offload certain processing to the cloud when needed; etc. In this sense, larger systems are constructed through the interaction of smaller ones, a practice known as *System of Systems (SoS) composition* or *construction* [21].

The inherent characteristics of SoSs make the development of these systems a challenging task [27]. By definition, a SoS is a complex system built from a (potentially large) number of sub-systems that in turn are made up of different components, and so on. Moreover, SoSs are typically deployed in environments where context changes. This is where the need arises to veer from a strict workflow path that has been defined at design time, and form a more complex system in order to maintain the intended abstract behavior or to be able to provide new behavior that is only possible through uniting with other systems in the new context. Such examples include disaster recovery, adaptive IoT applications, volunteer and crowd computing, military defense operations, and other forms of cyberphysical systems. Furthermore, the development of SoS is dynamic in nature. This is grounded in the knowledge that systems are persistent and long-living [29]. As such, their objectives and functionalities evolve over time as they are constantly added, modified, or removed at different time scales. This might also predicate changes in architectural and functional dependencies.

Despite the challenges posed by the above characteristics, current approaches of constructing SoSs assume that developers have in-depth knowledge of the internal structure of each system and its components [23]. Taking into account the above challenges and the characteristics of SoSs, it might be obvious that such approaches are deficient. We argue that the construction of SoS needs to be *autonomous* and *dynamic*. Systematic approaches for internal and context awareness are required to attain this goal. All systems should be able to accumulate knowledge about their own structure and behavior. Then, systems should exchange and be able to understand such knowledge at runtime so that they can opportunistically compose and form complex SoSs.

In order to achieve this objective, there is a need for semantics to comprehensively describe the system structure, capturing the information required for dynamically composing systems such as

SEAMS 2020, May 25–26, 2020, Seoul, South Korea
© 2020 Association for Computing Machinery.
https://doi.org/0000001.0000001

those relating to communication (*e.g.,* unique identifiers), service discovery, quality of service, physical properties (*e.g.,* power level and location), environmental properties, etc. We refer to this comprehensive structure description of a system as a *holon* [7]. A holon is constantly modified to reflect the system structure and to contain any new or modified information, and then published to aid discovery and reasoning about composition. When holons compose, they form a new holon that represents the SoS. This newly constructed holon will now have its own specification, which is published so that the system can further compose with other SoSs and so on.

The above holon ecosystem requires means of utilizing system (or holon) description in order to reason about SoS construction at runtime, and in a programmatic and adaptive manner. One solution is to adopt ontologies to specify the holons. Ontologies are engineering tools for formal and explicit specification of a shared conceptualization [8]. They provide vocabularies that can be used to represent knowledge that can be utilized programmatically to understand the corresponding contents, mainly system parameters, offered services, and requirements. These are expressed through standard concepts, allowing other holons to understand the described holon and, accordingly, make decisions about how to interact with it.

In [7] we proposed the vision towards the construction of distributed SoSs using holons, then in [28] we defined the basic conceptual structure with particular focus on IoT deployments. In this paper we build on these works by putting forward an architecture for realizing the holons and how they compose. The paper investigates how ontologies can be used to capture the holons' space to enable autonomous SoS construction. Once specified, the holon description can be published through broadcasting to other systems. The description is compiled by receiving systems in order to understand the functionality of the sending system, reason about it, and determine how to communicate with it.

Briefly, the main contributions of this paper are as follows:

- A framework for specification, compilation, dissemination, and modification of the holons (§4–§6)
- A qualitative evaluation of the proposed approach using two different case studies that motivate the need for dynamic SoSs construction and configuration (§7–§9).
- A quantitative evaluation of the scalability and validity of the proposed approach (§8,§10).

2 PROBLEM SPACE AND RELATED WORK

The SoS concept has featured in the literature for over two decades [21]. However, composing SoS from already existing systems at runtime is still an area that requires more research.

The significant body of work looming here is the Service Oriented Architecture (SOA) legacy. SOA allows a system to expose its functionalities as services, and then SOA composition technologies can be applied to form a SoS; *cf.* [4, 6, 39]. We argue that such SOA-based automation is not suitable for the following three main reasons. First, SOA does not readily provide concepts that capture physical system properties such as the context they are operating in. The service composition supported by SOA is mainly functionality-based composition. However, SoS composition is wider than this, as it also requires knowledge about system properties and context to reason about the composition. Second, services in SOA are assumed to be published in a repository that an orchestrator can consult to select services from. This assumption is not valid in the general SoS context which is fully distributed. Third, SOA-based SoS developers are expected to know a lot about individual systems, or invest significant time learning them. Fig. 1 illustrates the tasks performed by the vendors, developers, and the system for SOS composition. Developers are required to acquire knowledge about the APIs and properties of the elementary systems they need to compose. Then, they need to leverage that knowledge to design and implement the SoS, and finally deploy it. This requires the developer to focus on the elementary systems instead of the whole SoS.

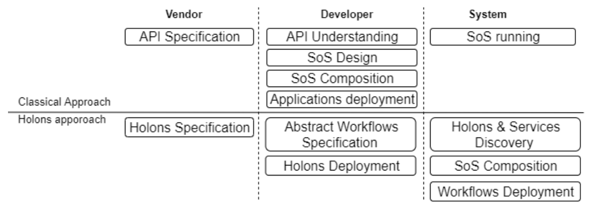


Figure 1: Responsibilities in current SoS composition approaches as compared to the proposed one.

There has also been other non-SOA efforts. These include facilitating discovery and composition using cellular infrastructure [12] and network middleboxes [9]. Such approaches, though, require specific infrastructure deployment for mediation and, more importantly, still assume too much on the part of the developer in terms of reasoning about discovered systems. This final task is crucially difficult without means of identifying these systems' modus operandi. Some works have chosen to do this at design time (*e.g.,* [22]) through analyzing qualitative mission objectives of systems, but these are hard to express in a programmatic way that enables automated reasoning at runtime.

The closest work to ours also defines the notion of a holon [19, 31], but is focused on goal-driven service composition without means of allowing holons to reason and self-compose.

Agent-based ontological approaches have been proposed, *e.g.,* [2, 3, 13, 33]. However, these works are less systematic than our holon ontology. Moreover, they also do not indicate how composition is reasoned about in a heterogeneous environment.

3 RESEARCH STRATEGY

Based on the above, our ultimate goal is to shift the developers' focus from learning the internals of elementary systems to thinking at the level of the SoS. As shown in Fig. 1, holons allow developers to focus on defining high-level workflows of the SoS. The elementary systems need to autonomously discover each other and compose to serve the requests. The achievement of this goal requires (1) a comprehensive description of the atomic systems (their services, properties, and context) that enable autonomous composition between systems; and (2) an architecture the exploits such descriptions and supports SoS composition and adaptation.

Our research seeks to answer the following three questions.

RQ1: What are the right abstractions to represent different systems within a SoS framework?

RQ2: What systems principles and techniques are then required to support SoS composition?

RQ3: What extensions are required to support SoS composition and adaptation in heterogeneous, large-scale environments?

Our work uses an ontological approach towards real-time reasoning around the composition of systems of systems. This is made possible by an architecture we have built for comprehending what an SoS is made up of, if composition is required, and how such composition would take place. Moreover, the architecture automatically actions the low-level mechanics that enable the composition. To validate the feasibility and utility of our architecture, we adopt a hybrid experimental evaluation strategy that comprises of qualitative assessment of controlled testbed experiments (specifically in the domains of IoT and infrastructure management) as well as quantitative assessment of real open-source systems.

4 THE HOLONIC LIFECYCLE

A holon starts as an atomic one then might evolve to being composite. An *atomic holon* represents a single system that provides one or more functionalities. A *composite holon* includes functionalities provided by a number of systems interacting with each other, where each pair of systems can interact directly or through a third one. In order for the holons to compose, atomic holons need to be comprehensively specified by the system developer. After that, the system will be dynamically constructed as the holons iterate in their lifecycle. Fig. 2 illustrates the four stages of the holon's lifecycle, which are outlined below.

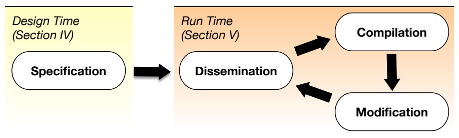


Figure 2: A holon's lifecycle

Specification. A system developer uses an ontology to create the holon from a number of elementary descriptions. This includes a holon identifier, the physical properties of the system (*e.g.,* power level), quality of service (*e.g.,* availability and reliability), environmental properties (*e.g.,* location), policies (*e.g.,* routing protocol), services (*e.g.,* sensing temperature), among others. All these are specified using our ontological model, described in §5.

Dissemination. A created holon disseminates its specification so it can be discovered. Different dissemination strategies can be adopted: push, pull, and lookup. In the *push* strategy, the holon periodically broadcasts its latest description. In contrast, a *pull* strategy uses heartbeat signals; *i.e.,* the holon periodically sends 'Hello' messages to establish interaction with other systems, which could then request the full holon description. This strategy reduces overhead, so is useful in energy-constrained environments. In the *lookup* strategy, the holon registers its description with a registry that can be consulted by other systems to obtain the holon. This strategy is to be used when infrastructure assistance is guaranteed. We adopt the push strategy for the scope of this paper.

Compilation. This aims at understanding other holons their functionalities. This is achieved by parsing received descriptions (in the form of XML representations), and identifying the functionalities contained within, and how they interact with that of the received holon. Compiling these functionalities into one holon representing a new SoS comprising of the interacting holons.

Modification. Holons can change at runtime due to a change in the physical system (*e.g.,* update of a service) or due to a structural change (*e.g.,* composition to a new holon). In either case, the holon description will be modified to reflect the change. Upon obtaining a modified version of a holon, the receiving holon will recompile its own holon and disseminate it. This will convey the changes to the whole SoS.

5 THE ONTOLOGICAL MODEL

5.1 Background

For a holon to be able to compose with another holon at runtime, it needs to advertise its own definition in a systematic way that can be easily understood by the receiving holon. For this, the definition needs to embody the different concepts surrounding the holon, triggering the need for an appropriate structure to represent it. In this respect, an ontology seems to be the best technique to capture the definition and behavior of a holon. An *ontology* is a formal and explicit specification of a shared conceptualization. It models some aspect of the world (called a *domain*), and provides a simplified view of certain phenomena in this domain. The description of the domain is based on a vocabulary that explicitly defines its concepts, properties, relations, functions, and constraints.

For our purposes, the ontology will be used by a holon to advertise its services, the types of input parameters required for said services, and the types of outputs they produce, if any. Furthermore, the ontology is used to identify the physical properties of the holon such as the power level (infinite/finite), location, operating system, mobility, etc.

Whilst an entirely new ontology can be developed from scratch, we deemed it more constructive to look at existing ontologies and extend suitable ones if and where necessary. In order to identify the most appropriate ontology to represent holons, we looked at different sensor and observation ontologies. We started with those surveyed by the W3C Semantic Sensor Network (SSN) Incubator Group [36][1], and continued with our own study of others. Table 1 shows the main ontologies that we reviewed.

Based on this investigation, the ontologies that we found suitable for extension are (in chronological order): CoDAMOS, Swamo, A3ME, and Ontonym. We concluded to use CoDAMOS [30] due to its inherent predisposition for modification, making it easily extensible for defining context-aware computing infrastructures varying from small embedded devices to high-end service platforms. Furthermore, CoDAMOS's concepts closely match the kind of definitions we want to create for a holon. For example, the *Service* concept - *i.e.,* having a service profile - can be used to define the types of services provided or required by a holon. Note, however, that this does not imply that the other shortlisted ontologies cannot be used. Indeed, A3ME [14] (to pick just one other ontology) could be merged with CoDAMOS to describe further concepts related to holonic design; for instance, the *APIPublic* CoDAMOS concept can

[1]It should be noted that we did not specifically opt for the SSN ontology – despite it being a complete ontology about sensors and their measurements – as our focus is not limited to a sensor-based system, but is rather more on capturing the abstract nature of a holon and on how to reason over whether a holon is atomic or composite.

Table 1: Survey of existing sensor-based ontologies

Ontology	Comments on Suitability	Suitable?
CoDAMOS	Easily extended to accommodate new definitions of devices and systems	✓
OntoSensor	The organization of concepts and properties is not transferable to a different context	×
MMI	Not well tested in other contexts	×
SensorML	Basic and without concept documentation	×
O&M	Expressive representations of time and space are not available	×
Swamo	Describes autonomous agents for system-wide resource sharing, distributed decision-making and autonomic operations	✓
SDO	Not available for use	×
A3ME	Can be easily extended	✓
Ontonym	A set of ontologies for representing core pervasive computing concepts	✓
Sensei	Some properties are not completely defined	×

be further described as *GlobalAddress*, *LocalAddress*, or *OtherAddress* using the *Address* concept of A3ME.

For this paper, we use CoDAMOS as the basis ontology with the aim of answering our research questions (§3), and in particular RQ1: is it possible to define something as abstract as a holon using an ontology, which is by nature prescriptive and unambiguous?

5.2 Extensions

Among its various concepts, there are four basic CoDAMOS concepts that stand out in terms of designing holons: *User*, *Environment*, *Platform*, and *Service*. The *User* concept has a profile, a role, and a task. The task can have activities and/or uses a service. The *Environment* concept has a location (relative or absolute), a time, and an environmental condition (*e.g.,* temperature, pressure, humidity, lighting, noise). The *Platform* concept has an Environment and can provide a Service (used by User). The *Service* concept has a profile, a model, and grounding. Resources such as memory, network, power, storage resources can also be modeled through the ontology, as well as different kinds of software or hardware.

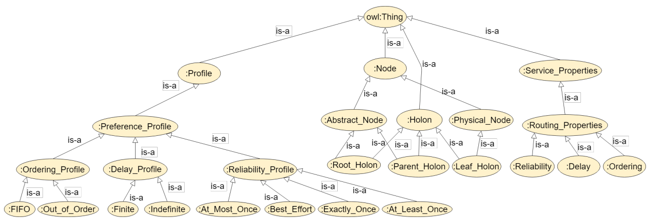


Figure 3: Our extensions to the CoDAMOS ontology, enabling the representation of holons to form SoSs

Fig. 3 summarizes how we extended CoDAMOS to accommodate the requirements of a holon. The *Profile* is extended to show profile preferences for routing messages in a system such as *Ordering*, *Reliability* and *Delay*. A new concept *Node* is added to capture the types of nodes encountered in a system: physical and abstract. This allows systems to be organized in a hierarchical way with physical systems at the very bottom of the hierarchy as leaf holons, and abstract systems as their parents, and so on till a root holon at the very top. All of these are added as concepts under the *Holon* concept. Furthermore, Service Properties has been extended to accommodate routing properties such as Delay, Reliability and Ordering. For implementing these extensions we used Protégé [24], an open-source ontology editor.

5.3 Application

For demonstration, we show how two holons can be composed using their underlying description. As such, the starting point is where each holon defines itself using the ontology: its properties, the services it provides, and the parameters being used. These descriptions are broadcasted by each respective holon.

On receiving such broadcast, holons will compose with each other if they meet the criteria for composition, for instance: if a holon H_1 is requiring a service X, and it encounters another holon H_2 that is providing such service, then H_1 can initiate the composition procedure with H_2. Once H_1 gets composed with H_2, it needs to update its definition to reflect its new state as a composed holon, *i.e.,* a SoS, that is now providing service X. This update is carried out at runtime through creating an instance of the *holon* concept (called, say, H_3) to represent the new holon that has been encountered, and it also holds the definition of H_2 in this case.

Moreover, the ontology retains the ability to infer new knowledge based on the domain information provided. For example, there is a defined concept called *composedHolon* that is used to identify whether a holon is a simple or composed one. In this context, given that a holon has been composed to another holon, the ontology reasoner can determine whether this holon is a composed one. If H_2 gets out of reach, then the ontology of the composed H_1 will be updated to accommodate this change, simply by removing the H_3 instance and clearing *composedHolon*.

6 SOS CONSTRUCTION MODEL

This section provides the mechanics of using holonic ontologies in order to reason about their composition to form a SoS at runtime. At a high level, our approach is first to transform a holon to be represented as weighted tree that reflects its interaction with other holons. Then, upon receiving a request for a service, the tree is used to find through which holon the service can be accessed. This section also presents the architecture that realizes the approach.

6.1 Composition model

Each holon needs to build a model that represents its awareness about the existence of other holons (*i.e.,* systems) and their services. This model (called the composition model) is used to interact (compose) with the other systems by accessing their services. The composition model of a holon is represented as a weighted tree T rooted by the holon and with a depth of three. The children of the root are the holons that are directly reachable by the root holon. The leaf nodes represent the detected functionalities that are provided by or accessed through the children holons. Each leaf node i is assigned a weight that represents the cost of accessing the corresponding functionality F_i. For simplicity, in this paper we define the cost as the distance between the root and the holon providing F_i in terms of the number of intermediate holons. Other cost functions such as delay, reliability or aggregated Quality of Service (QoS) can also be used. It is worth mentioning that a holon is not aware where a functionality F_i is located. However, the holon is aware of the cost of accessing F_i and through which holon can F_i be accessed. Fig. 4 shows an example of a holon connected to three other holons in its neighborhood. Functionality F_2 can be reached through both H_1 and H_2, but it is less costly to access through H_1.

The composition model is frequently updated during the lifecycle of the holon. Updates include adding/removing new holon branches, and updating the services of current ones. Adding and updating are performed upon receiving holon ontologies, while removing is carried out if the ontology is not received during a certain time period (we set this to be three times the dissemination period).

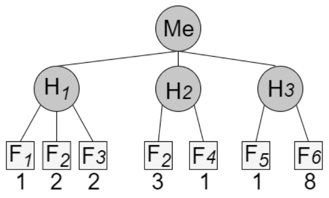


Figure 4: Composition/interaction modeling example.

6.2 Reasoning architecture

We assume that a system developer will create the atomic holon using the ontology described in §5. Once deployed, the holon lives in the described lifecycle (§4). Fig. 5 offers a high level view of the architecture that realizes SoS construction using the holonic approach. The architecture consists of five main components.

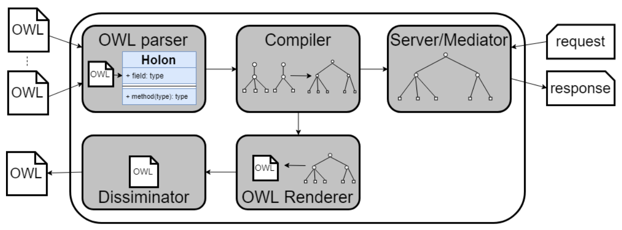


Figure 5: System architecture

OWL parser. This component receives ontologies from surrounding holons and parses them, creating objects that represent each holon and its functionalities. We adopt OWLAPI V5.1 [16] to parse ontologies and extract the knowledge therein.

Fig. 6 displays an overview of the mapping of ontology elements to a holon object. For each Class element, a Java class is created to represent it. For example, a Java class `Service` represents the ontology *Service* concept, `Profile` represents the *Profile* concept, and so on. The Instance elements of the ontology are instances of the Class elements. The instances are linked together using *DataType* or *Object* properties. Such structure is mapped as attribute objects of the `Holon` object. For example, assume the ontology contains an element $service_i$ that is an element of class *Service*. This instance can be linked to the *Holon* concept using an object property element called *hasService*. In the Java `Holon` class, this is mapped by having an attribute called `service_i` of type `Service` in the `Holon` class. Finally, in the ontology *Value* elements represent some of the parameters values that are mapped as attribute values of the `Holon` Java objects. An example is the value '25°C' of the *Temperature* element which is assigned to the `temperature` attribute in the `Holon` Java object. The OWL parser then passes the created `Holon` object to the Compiler.

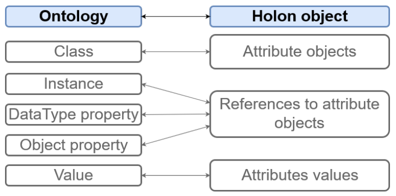


Figure 6: The relationship between main ontology elements and those of a holon.

Compiler. This component formulates a notional model of the developing SoS as seen from the perspective of the receiving holon. This model takes the form of a tree data structure with the receiving holon at the root, with each branch representing a neighboring holon and the functionalities provided either natively by it or indirectly through it. To construct or modify the tree, the Compiler creates a tree branch that represents the received holon object and its provided functionalists as children. The cost of each functionality is also updated as this stage. After that, the newly formed branch is attached to the root holon as a child. The Compiler then passes the constructed tree to the Server/Mediator and OWLRenderer.

Server/Mediator. This component receives requests for the functionalities provided by the system and serves them. It uses the constructed holon tree to specify whether the request can be satisfied by the system or needs to be directed to another holon. If the requested functionality is provided by the holon, then the system processes the request and responds to the requester. Otherwise, the system acts as a mediator and redirects the request to the holon that provides the request, waits for the response, and passes back the response to the requester.

OWLRenderer. What this component fundamentally does is represent the knowledge of a developing SoS based on interaction with new holons. The OWLRenderer reads the holon tree data structure from the Compiler and converts each node and the corresponding attributes into the appropriate XML tags that construct a valid ontology (*i.e.,* an inverse mapping of Fig. 6). To render the ontology into an OWL description, we also use OWLAPI.

Disseminator. Upon receiving an OWL description representing the holon and the compositions with other holons, the Disseminator publishes this description by broadcasting it – as is the policy in the *Push* strategy (see §4).

7 CASE STUDY I: AUTONOMIC SMART HOME

In this case study, we focus on the ontology exchange between holons to realize SoS logic.

7.1 Background and challenges

Smart Home is an application of IoT that promises the ability for users to intelligently manage their homes with minimum intervention. Devices are utilized to monitor home conditions (*e.g.,* temperature, humidity) and the state of appliances (*e.g.,* battery levels) to enable smart management of energy consumption, security, and various house keeping functions. However, it is no longer realistic to assume a defined and small scale of smart home deployments. As smart home systems are increasing in popularity, the dimensionality of IoT devices used is increasing as new devices are constantly offered in the market and with increased capabilities [37]. Thus, their deployment environments are continually evolving [35], making self-adaptation a necessity [17, 23].

Furthermore, homogeneity cannot be taken for granted [38]. The fact that these devices are developed by various vendors means that heterogeneous technologies and APIs exist. Smart home application developers need to learn these various technologies and APIs in order to develop IoT applications that manage the different aspects of a smart home in a coherent manner.

Finally, IoT applications generally limit consumers to the predetermined deployment environments they were designed for [26]. Consequently, they are brittle and susceptible to suboptimal operation when their context changes, *e.g.,* due to a backhaul network fault or a server outage. Under such conditions, centralized (mainly cloud-based) approaches fall short especially in network-constrained conditions [10]. Instead, IoT applications often need a way to be able to adapt at the edge without preparation or central coordination.

7.2 Overview of our approach

In this case study, we illustrate how the adoption of our ontology-based approach helps to significantly mitigate the mentioned complexity. Building on the approach of integrating web services with the IoT technology to enable remote access of data gathered by the IoT devices (*e.g.,* [32]), we argue that our approach enables the automatic discovery of services that are required for an adaptive smart home management application. As in the web service composition case, application developers need to specify the abstract workflow of the application execution where each task in the workflow represents an abstract service. Then, concrete implementations of the abstract services are selected for the workflow realization.

We consider each device as a holon that is described using the ontology (probably by the vendor). The management application is also considered as a holon that receives other holon ontologies, parses and compiles them, and then uses their services. Application developers specify the abstract workflow of the smart home management application. Then, the Server/Mediator component selects the concrete services that implement the abstract services of the workflow. A concrete workflow (the application) is passed to an execution unit that executes the application and passes the results to the smart home actuators. Fig. 7 illustrates the above steps.

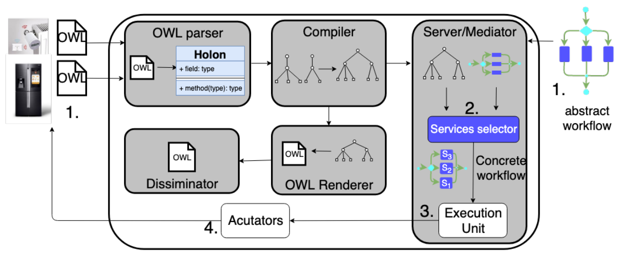
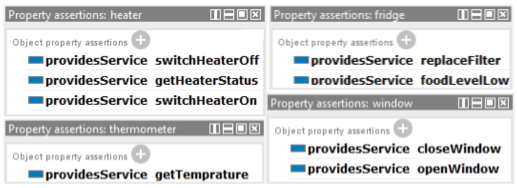


Figure 7: Smart Home Controller


Property assertions: heater
Object property assertions
providesService switchHeaterOff
providesService getHeaterStatus
providesService switchHeaterOn

Property assertions: thermometer
Object property assertions
providesService getTemprature

Property assertions: fridge
Object property assertions
providesService replaceFilter
providesService foodLevelLow

Property assertions: window
Object property assertions
providesService closeWindow
providesService openWindow

Figure 8: Object properties of the device ontologies.

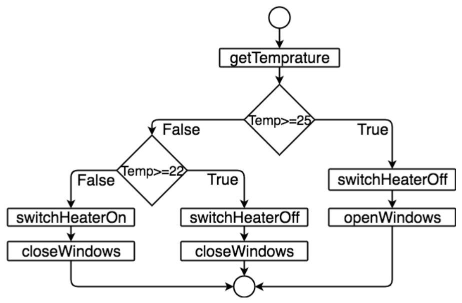


Figure 9: Prototype workflow

7.3 Experimental setup

We developed a proof of concept prototype focused on smart home temperature management. We assume a smart home that has the following smart devices: heaters, thermometers, windows, and a fridge. The devices provide the services listed in Table 2.

Table 2: Smart home devices and services.

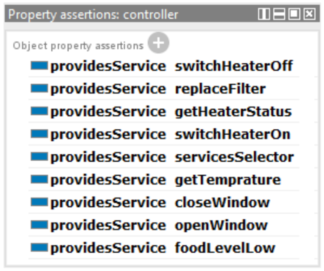
Device	Service	Description
Window	`openWindows`	Opens the windows (by actuators)
	`closeWindows`	Closes the windows (by actuators)
Thermometer	`getTemprature`	Returns the home temperature
Heater	`switchHeaterOn`	Switches on the heating
	`switchHeaterOff`	Switches off the heating
	`getStatus`	Returns true if the heating is on
Fridge	`replaceFilter`	Tells if the filter should be replaced
	`foodLevelLow`	Returns true if food level is low

We created a simple ontology for each of the devices as shown in Fig. 8. We also created an abstract workflow that aims at keeping the smart home temperature at 22°C, which is depicted in Fig. 9. The application reads the temperature from the thermometers. If the temperature is less than 22°C, it calls the heaters' `switchHeaterOn` service to switch on the heating and the windows' `closeWindows` to close the windows. If the temperature is higher than 22°C, it turns off the heater (calling `switchHeaterOff`). If the temperature is higher than 22°C it also opens the windows (by calling `openWindows`).

7.4 Behavior

Fig. 10 shows a snapshot of the object properties of the controller ontology after parsing the device ontologies and constructing the SoS tree. The ontology shows that the controller provides services that are provided by the devices themselves (see Fig. 8) in addition to `servicesSelector`, which selects concrete services for the abstract workflow.

The execution starts with reading the temperature from the thermometer by calling the `getTemeprature` service. Table 3 presents the experimental cases and the invoked cases for each case. As the


Property assertions: controller

Object property assertions

providesService switchHeaterOff
providesService replaceFilter
providesService getHeaterStatus
providesService switchHeaterOn
providesService servicesSelector
providesService getTemprature
providesService closeWindow
providesService openWindow
providesService foodLevelLow

Figure 10: Object properties of the controller ontology

Table 3: Experimental cases of the workflow in Fig. 9

Temp.	Invoked Services
10°C	`getTemperature` → `switchHeaterOn` → `closeWindows`
18°C	`getTemperature` → `switchHeaterOn` → `closeWindows`
22°C	`getTemperature` → `switchHeaterOff` → `closeWindows`
23°C	`getTemperature` → `switchHeaterOff` → `closeWindows`
26°C	`getTemperature` → `switchHeaterOff` → `openWindows`

table illustrates, the sequence of invoked services complies with the abstract workflow provided by the developer to the controller, despite the developer not hard-coding the required connections between devices. For example, when the temperature is 22°C, the execution unit invokes the `getTemperature`, `switchHeaterOff`, and `closeWindows` services respectively, which complies with the abstract workflow (Fig. 9).

This case study demonstrates the potential for easing the development of IoT applications, where application developers do not need to know the details of smart device APIs. Definition of device ontologies (by vendors) and an abstract application workflow (by smart home developer) are sufficient for runtime application synthesis and execution by the proposed architecture. Table 4 compares the tasks required from the smart home developers to implement this case study in both the classical and holonic approaches.

Table 4: Tasks required from developers to implement the smart home case study.

Classical approach	Holonic approach
1. Understand the services, APIs, and properties of the devices listed in Table 2. 2. Design the composition of services based on required functionality and context. 3. Develop code to compose the services. 4. Deploy the system.	1. Define the abstract workflow of the required functionality on the controller. 2. Deploy the devices listed in Table 2.

8 CASE STUDY II: DYNAMIC CLUSTER MANAGEMENT

In this case study we center our attention on holon interaction, demonstrating how holons containing the same service in their ontologies and in constant movement creating and modifying new SoSs, can interact with each other.

8.1 Background and challenges

Inspired by heterogeneous fog clusters, we set a scenario where device interaction is constant. We use Apache Mesos [15], an orchestration tool commonly used to manage resources that are shared between different applications and their sub-tasks. In effect, Mesos enables the viewing of data centers and other computing clusters as a single consolidated resource.

Although Mesos is a very useful utility, it was designed mainly for shared resources in relatively stable environments such as data centers. As such, the computing cluster can only change (grow or shrink) through manual modifications to the configuration by the user. Mesos was not designed to work in an environment where node status is constant flux due to movement, unreliable power, or communication outages. Mesos is also designed to work in a hierarchical fashion, whereby *Agents* (worker nodes) can only communicate directly to the *Master* but not through other Agents [15, 18].

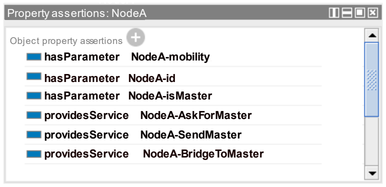
8.2 Overview of our approach

Both of the above restrictions can be overcome using the holonic ontology presented, which offers opportunistic composition (overcoming the first restriction) and horizontal composition between self-describing clusters in the form of holons (second restriction). We draw a scenario here to demonstrate this using containers running over an unreliable infrastructure such as edge PoPs [11]. In this scenario, each node can be a Master or an Agent. Following the basic design of Mesos, Masters are responsible for dispatching containers to the Agents, who in turn operate the containers.

8.3 Experimental setup

A simple example is given here to illustrate how holons could be used to facilitate the union of Mesos clusters with mobile nodes without the need for establishing direct communication. Due to the high mobility of nodes during the tests, the composition of SoSs is dynamic, creating several additions and removals of the Mesos Services. This is something that would have been prohibitive to accomplish using Mesos' manual configuration.

We tested this use case study using 100 devices that can move freely. Every device is considered a node for Mesos and is individually defined as a holon at the beginning of the experiment. Shortly afterwards (a minute later), bigger holons begin to be created, containing one or more devices. After that, each holon starts an internal process of randomized leader election mechanism [25] to elect a master from amongst the constituent devices.


Property assertions: NodeA

Object property assertions

hasParameter NodeA-mobility
hasParameter NodeA-id
hasParameter NodeA-isMaster
providesService NodeA-AskForMaster
providesService NodeA-SendMaster
providesService NodeA-BridgeToMaster

Figure 11: The ontology model of Mesos NodeA.

Each holon contains parameters to define its identifier, mobility, and whether or not it is a Mesos Master (*e.g.,* Fig. 11). Additionally, each holon contains three services: The `AskForMaster` service is performed every time two holons reach each other for the first time, where it would return the ID of the Master. After a holon receives the request of whom is its Master, it will perform the `SendMaster` service to send such ID. `BridgeToMaster` is used by the nodes to communicate with the Master through its Agents. Fig. 12 shows an abstract workflow of the experiment when a node reaches a new node from another Mesos cluster. A second abstract workflow has been created when a node receives a new ontology, which is depicted in Fig. 13. Both abstract workflows are used by the nodes to allow dynamic union of Mesos clusters with mobile nodes.

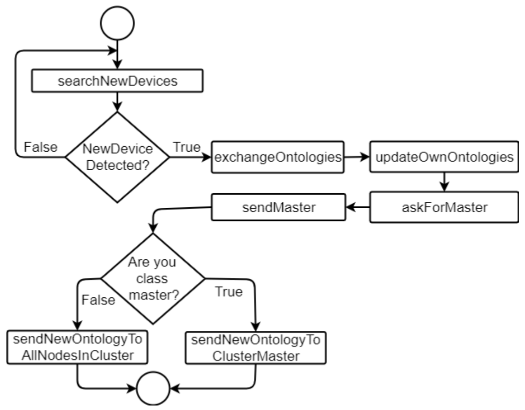


Figure 12: Prototype workflow of Dynamic Cluster Management when a node reach another node.

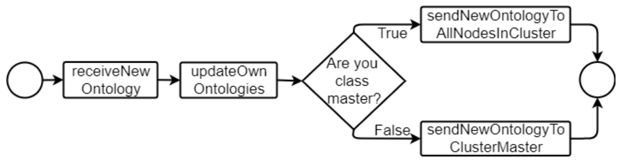
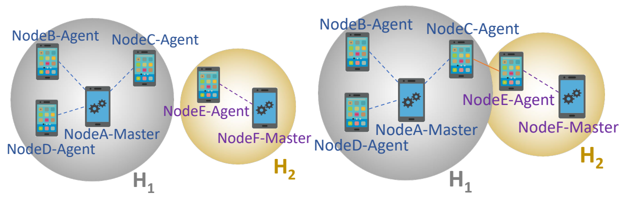
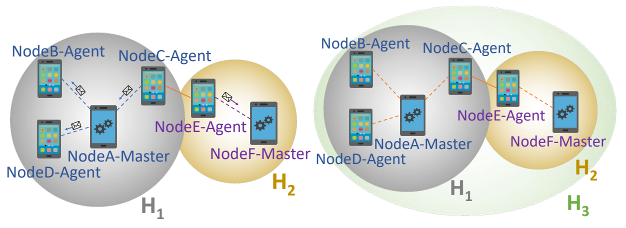


Figure 13: Prototype workflow of Dynamic Cluster Management when a node receive a new ontology.



(a) H_1 is unreachable from H_2 **(b) C and E come in range, interact**



(c) Ontologies exchanged & updated **(d) A SoS, H_3, is created**

Figure 14: How to use holons to construct a SoS of Mesos clusters running on mobile nodes.

8.4 Behavior

Fig. 14a presents the starting point with two holons: H_1 and H_2. H_1 is formed by four nodes: nodeA, working as a Master Mesos, and nodeB, nodeC and nodeD working as Agents. On the other hand, H_2 is formed by two nodes: nodeF, a Master, and nodeE, an Agent. In this case, H_1 and H_2 cannot reach each other.

In Fig. 14b, H_1 and H_2 reach each other through nodeC and nodeE that come in range of one another. This triggers them to exchange ontologies and call `AskForMaster` and subsequently `SendMaster`. The orange arrow in the figure represents this interaction.

Then, nodeC and nodeE broadcast their new ontologies to neighbors as shown in Fig. 14c. As all nodes in H_1 and H_2 receive the updated ontology, a new holon is created: H_3, which is the union of H_1 and H_2. As a result, all nodes inside H_3 have the same ontology.

All Agents are sharing the Masters nodeA (from H_1) and nodeF (from H_2). This is achieved using the service `BridgeToMaster`, performed continuously by nodeC and nodeE while they can reach each other. The resulting SoS H_3 has 2 Master nodes and 4 Agents. Therefore, each of the two Masters can communicate to all Agents in H_3 as if they were connected directly.

As such, the ontological exchange and reasoning of holons allows Apache Mesos to transcend its innate design shortcomings and enables it to form a dynamic cluster structure. Achieving such dynamic structure using manual configuration (which is the only way possible using native Mesos) significantly restricts adaptation and reduces cluster efficiency by a factor of 5 compared to using holons. Table 5 compares the tasks required from Mesos developers to implement this case study in both classical and proposed approaches.

Table 5: Tasks required from Mesos developers to implement the dynamic cluster management case study.

Classical approach	Holonic approach
Repeatedly: 1. Select a cluster Master. 2. Add nodes to cluster. 3. Add services to cluster. 4. Remove nodes from cluster. 5. Remove services from cluster.	1. Define the cluster node as a holon using the ontology. 2. Implement the cluster Master election algorithm. 3. Deploy the holons.

8.5 Simulations

For evaluating the efficacy of automated expansion of Mesos clusters through the use of holon ontologies, we used the Omnet++ discrete-event simulation framework [34] to simulate 100 nodes. We set the node transmission range to $20m$ and their speed to $1.43m/s$, an average walking speed [20]. Additionally, the nodes used the individual-level (random walks) [5] as a mobility model.

We used Mesosaurus [1] to create task loads to test the performance of the formed clusters. Specifically, we seek the length of time required by a Mesos Master to perform a specific task. The task created for this experiment is one that a Master with 5 Agents will normally perform in about 20 seconds. If more Agents are employed, execution time is expected to decrease.

Fig. 15 displays the average task execution time across all Mesos clusters in the experiment after performing it 20 times. Using

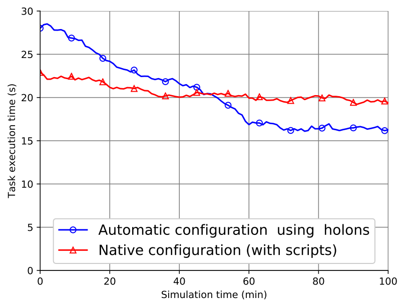


Figure 15: Computational tasks are completed faster on dynamic clusters constructed using holons as opposed to native Mesos cluster configurations.

Mesos' native method of cluster management, task execution time decreases slightly (from $\approx 23s$) as Masters expand their clusters through the use of scripts that to add nodes they encounter in their environment. This improvement in performance, however, eventually plateaus ($\approx 20s$) as churn overwhelms Masters through frequent configuration management, despite the use of automated scripts. On the other hand, using holons and ontologies introduces some overhead in terms of ontology creation and reasoning. This results in a slightly inflated initial execution time ($\approx 27s$). However, as nodes encounter others during the lifetime of the simulation, Mesos clusters identifying as holons expand dynamically according to changes in their environment. Compiling and reasoning overheads soon become relatively insignificant, enabling Mesos holons to achieve an average of $17s$ execution time, a 15% increase in performance.

9 DISCUSSION

The above experiments show how we are able to enable the automatic self-discovery and composition of systems through rich ontological description of elementary systems that are required to realize a SoS. We now reflect on our research questions (§3).

9.1 Abstractions for SoS representation

Holons offer the ability for systems to richly describe themselves and independently reason about the change in their environment and how it might affect their set up and operation. This concept enables systems to reflect on their existence and how they fit into the rest of the system around them. This in line with the ethos of reflective middleware, which enables systems to build a representation of itself that it can then adapt. In addition, holonic representation allows systems to transfer such knowledge about themselves to other systems they get in touch with. Along with this, each system is able to build a representation of the behavior of systems in its vicinity and form more complex systems without prior arrangement.

There is an assumption that each system needs to start with a representation of itself in its simplest form as an ontology. Because of this, we built our ontological architecture on the most generalize and easy-to-use ontology available in the literature. Furthermore, using such framework makes it easy to create ontologies, which is a relatively small development overhead of similar or less scale of defining a system's API. However, this enables the system to adapt after its deployment and unlock a new world of complex system creation that facilitates new forms of aware applications.

9.2 Techniques for SoS composition

The presented case studies demonstrated how a developer could define desired behavior at a high level (as summarized in Tables 4–5), and a system is subsequently composed of other sub-systems to align with this behavior. Our architecture allows systems to independently reason about their environment, and how changes might affect their set up and operation. This is a powerful concept as it maintains the separation of concerns, which is crucial for effective system development, whilst also reaping the rewards of complex system formation through autonomous composition.

Furthermore, the holonic ontology could be applied at different levels: at the atomic service level (*e.g.,* temperature sensor), at a system component level (*e.g.,* smart sensors), or at a higher system level (*e.g.,* smart home controller). This enables developers to write behavior at different levels of granularity with the same modeling effort, which is especially beneficial for dynamic environments such as the IoT where context-dependent behavior could be sought at different levels.

9.3 SoS adaptation

The presented architecture exploits the holonic ontology and maintains the holonic lifecycle to fully support the vision of autonomous SoS composition and adaptation presented in this paper. The architecture continuously updates the holonic ontology allowing up-to-date SoS state exchange and enables timely response to changes. This allows holons to detect failures and discover new functionalities a holon needs to rebuild a SoS. As demonstrated in the cluster management case study, adaptation is fully autonomous. The system can, for example, detect the arrival of new working nodes and add them to the cluster.

Furthermore, human involvement is only needed when major requirements changes are required, *i.e.,* at the SoS level. For example, a change in the smart home requirements that demands the deployment of new devices might require updating the behavior of the smart home applications. This will need developers to adapt the abstract workflows. However, such intervention is guaranteed to be minimal as developers do not need to know the low-level details of post-deployment systems such as device-specific APIs. As such, our proposed architecture provides a generic framework for supporting high-level behavior adaptation to real implementation.

10 QUANTITATIVE EVALUATION

We conducted experiments to evaluate the feasibility of our approach, specifically the time required to parse and render the ontologies at different scales. We also evaluate the validity of the parsing and rendering stages. From this, we extract conclusions about the ability of using holons to compose SoS during runtime and at scale. The used platform is Intel Core i7 with 16GB RAM, running Linux Ubuntu 16.04 and Java SE v1.8.0. Each experiment is repeated 100 times to obtain representative mean values.

10.1 Parsing

This first experiment focuses on evaluating the parsing time defined as the time of converting a received ontology into a tree (§6). Recalling that the tree contains the holon object as a root and the services provided by/via the holon as children, in this experiment we vary the number of services provided by/via the holon, which is the main dimension affecting the scalability of the parsing. Fig. 16a plots the average parsing time in milliseconds. We notice that the parsing time increases with the number of services, exhibiting a linear trend. The increase is expected and the linear complexity is acceptable as it indicates the feasibility of parsing an increasing number of holons in a SoS.

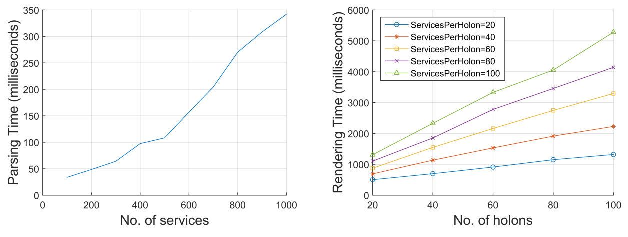
10.2 Rendering

This experiment focuses on evaluating the rendering time defined as the time of converting the holon tree into an ontology to be disseminated. Recalling that the tree contains the system holon as the root, the neighbor holons as the children, and their services as leaf nodes, we vary both the number of neighbor holons and the number of services that are provided by/via them. Fig. 16b plots the rendering time in milliseconds. The plot shows that the rendering overhead increases both with the number of holons and with the number of services per holon. In other words, the more sub-systems a holon contains, the longer it takes to create its ontology that could be used for composition. Again, the increase exhibits a linear trend which is practically acceptable.

10.3 Validation

As previously mentioned, the proposed framework includes parsing holon ontologies to build a tree that represents the SoS construction, and then subsequently rendering the tree to disseminate the modified ontologies that reflect the SoS evolution. In this subsection we validate the output of the parsing and rendering operations using two experiments.

In the first experiment we adopt the process depicted in Fig. 17a. We create ontologies for 100 holons with random values for each of their parameter and service values. We then pass the ontology files to the OWLParser, which creates the holon trees. The trees are then passed to the OWLRenderer to render them into ontologies. We then query (using OWLAPI [16]) both the created ontologies and the rendered ones to check if the results are equal. For all of the 100 ontologies, the returned values were indeed equal.



(a) Parsing time **(b) Rendering time**

Figure 16: (a) Ontology parsing time as the number of services per holon increases. (b) Rendering time for holons of increasing size (and, thus, ontology complexity) as the number of holons increases.

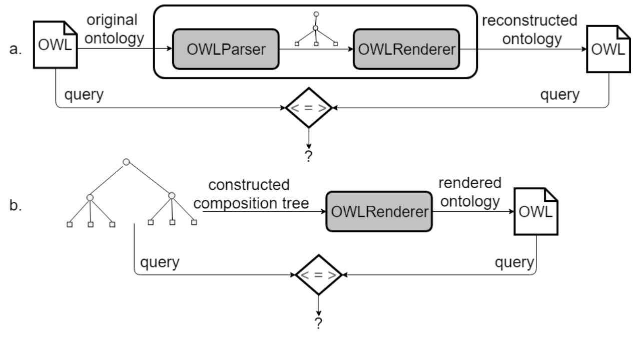


Figure 17: A diagrammatic representation of our two methods to validate ontology parsing and rendering experiments. The parsing and rendering of the ontology are valid when the information returned by the queries match.

In the second experiment, we adopt the process portrayed in Fig. 17b. Here, we synthesized 100 compiled trees, each representing a SoS. We then passed the trees to the OWLRenderer to render them into a SoS ontology for each tree. Then we queried both the rendered ontologies and the created trees and compared the results. Again, for all of the 100 cases, the returned values were equal.

11 CONCLUSION

We propose an approach for the dynamic construction of distributed systems of systems (SoSs). The approach is based on two key ideas. First, we define the concept of a *holon* as a self-describing system, which could span from atomic to complex distributed systems. Holons need to be prepared for autonomic integration with other holons. This is achieved by comprehensively describing them using an ontology that enables both self-awareness and context-awareness. Second, an architecture for SoS construction is proposed to make use of the holon description to discover, reason about their functionalities, and integrate them to form more complex SoS.

We demonstrate the feasibility of our approach through two case studies that implement contrasting SoS construction scenarios. The cases studies show that our approach reduces the development complexity of SoS by abstracting the heterogeneity of the systems using holon descriptions and their autonomic manipulation at runtime. We also evaluate scalability and validity through experimentation, concluding that our approach is realistically feasible with performance exhibiting a linear trend for manipulating and reasoning about descriptions.

This novel contribution has strong potential to be applied in various application fields beyond those covered in our case studies. Similarly, our architecture could be modified to cater for domain-specific interactions if particular situational-awareness are needed. Moreover, we are extending this work by building tools that allow very high level specification of desired SoS construction behavior and evolution.

12 ACKNOWLEDGMENTS

This work was supported by CHIST-ERA under UK EPSRC grant EP/M015734/1 (DIONASYS).

REFERENCES

[1] 2016. Mesosaurus. https://github.com/mesosphere/mesosaurus.

[2] R. Agarwal, D. G. Fernandez, T. Elsaleh, A. Gyrard, J. Lanza, L. Sanchez, N. Georgantas, and V. Issarny. 2016. Unified IoT ontology to enable interoperability and federation of testbeds. In *World Forum on Internet of Things (WF-IoT)*. 70–75. https://doi.org/10.1109/WF-IoT.2016.7845470

[3] Muhammad Intizar Ali, Pankesh Patel, Soumya Kanti Datta, and Amelie Gyrard. 2017. Multi-Layer Cross Domain Reasoning over Distributed Autonomous IoT Applications. *Open Journal of Internet Of Things (OJIOT)* 3, 1 (2017), 75–90.

[4] R. R. Aschoff and A. Zisman. 2012. Proactive adaptation of service composition. In *Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 1–10. https://doi.org/10.1109/SEAMS.2012.6224385

[5] Hugo Barbosa, Marc Barthelemy, Gourab Ghoshal, Charlotte R James, Maxime Lenormand, Thomas Louail, Ronaldo Menezes, José J Ramasco, Filippo Simini, and Marcello Tomasini. 2018. Human mobility: Models and applications. *Physics Reports* (2018).

[6] Luciano Baresi and Liliana Pasquale. 2010. Live Goals for Adaptive Service Compositions. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Association for Computing Machinery, New York, NY, USA, 114âĂŞ123. https://doi.org/10.1145/1808984.1808997

[7] Gordon S Blair, Yérom-David Bromberg, Geoff Coulson, Yehia Elkhatib, Laurent Réveillère, Heverson B. Ribeiro, Etienne Rivière, and François Taïani. 2015. Holons: Towards a Systematic Approach to Composing Systems of Systems. In *Workshop on Adaptive and Reflective Middleware*. Article 5, 6 pages.

[8] Ines Ceh, Matej Crepinšek, Tomaž Kosar, and Marjan Mernik. 2011. Ontology driven development of domain-specific languages. *Computer Science and Information Systems* 8, 2 (2011), 317–342.

[9] Abdessalam Elhabbash, Gordon S. Blair, Gareth Tyson, and Yehia Elkhatib. 2018. Adaptive Service Deployment using In-Network Mediation. In *International Conference on Network and Service Management (CNSM)*. 170–176.

[10] Yehia Elkhatib. 2015. Building Cloud Applications for Challenged Networks. In *Embracing Global Computing in Emerging Economies*, Ross Horne (Ed.). Communications in Computer and Information Science, Vol. 514. Springer International Publishing, 1–10. https://doi.org/10.1007/978-3-319-25043-4_1

[11] Yehia Elkhatib, Barry F. Porter, Heverson B. Ribeiro, Mohamed Faten Zhani, Junaid Qadir, and Etienne Rivière. 2017. On Using Micro-Clouds to Deliver the Fog. *Internet Computing* 21, 2 (2017), 8–15.

[12] Gábor Fodor, Erik Dahlman, Gunnar Mildh, Stefan Parkvall, Norbert Reider, György" Miklós, and Zoltán' Turányi. 2012. Design aspects of network assisted device-to-device communications. *IEEE Communications Magazine* 50, 3 (March 2012), 170–177.

[13] M. G. Gillespie, H. Hlomani, D. Kotowski, and D. A. Stacey. 2011. A knowledge identification framework for the engineering of ontologies in system composition processes. In *Conference on Information Reuse Integration*. 77–82. https://doi.org/10.1109/IRI.2011.6009524

[14] A. Herzog, D. Jacobi, and A. Buchmann. 2008. A3ME - An Agent-Based Middleware Approach for Mixed Mode Environments. In *UBICOMM*. 191–196.

[15] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11.

[16] Matthew Horridge and Sean Bechhofer. 2011. The OWL API: A Java API for OWL Ontologies. *Semantic Web* 2, 1 (2011), 11–21.

[17] Danny Hughes. 2018. Self Adaptive Software Systems are Essential for the Internet of Things. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 21–21.

[18] Dharmesh Kakadia. 2015. *Apache Mesos Essentials*. Packt Publishing Ltd.

[19] M. Kit, I. Gerostathopoulos, T. Bures, P. Hnetynka, and F. Plasil. 2015. An Architecture Framework for Experimentations with Self-Adaptive Cyber-physical Systems. In *Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 93–96. https://doi.org/10.1109/SEAMS.2015.28

[20] Robert V Levine and Ara Norenzayan. 1999. The pace of life in 31 countries. *Journal of cross-cultural psychology* 30, 2 (1999), 178–205.

[21] Mark W. Maier. 1998. Architecting principles for systems-of-systems. *Systems Engineering* 1, 4 (1998), 267–284.

[22] Behrokh Mokhtarpour and Jerrell Stracener. 2017. A Conceptual Methodology for Selecting the Preferred System of Systems. *IEEE Systems Journal* 11, 4 (Dec 2017), 1928–1934.

[23] Henry Muccini, Mohammad Sharaf, and Danny Weyns. 2016. Self-Adaptation for Cyber-Physical Systems: A Systematic Literature Review. In *Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. Association for Computing Machinery, New York, NY, USA, 75âĂŞ81. https://doi.org/10.1145/2897053.2897069

[24] Mark A. Musen et al. 2015. The Protégé Project: A Look Back and a Look Forward. *AI Matters* 1, 4 (Jun 2015), 4–âĂŞ12. https://doi.org/10.1145/2757001.2757003

[25] Koji Nakano and Stephan Olariu. 2000. Randomized Leader Election Protocols in Radio Networks with no Collision Detection.

[26] Anne H. Ngu, Mario Gutierrez, Vangelis Metsis, Surya Nepal, and Quan Z. Sheng. 2017. IoT Middleware: A Survey on Issues and Enabling Technologies. *Internet of Things Journal* 4, 1 (Feb 2017), 1–20.

[27] Claus Ballegaard Nielsen, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. 2015. Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions. *ACM Comput. Surv.* 48, 2, Article 18 (2015), 41 pages.

[28] Vatsala Nundloll, Yehia Elkhatib, Abdessalam Elhabbash, and Gordon S Blair. 2020. An Ontological Framework for Opportunistic Composition of IoT Systems. In *International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*. IEEE.

[29] Kai Petersen, Mahvish Khurum, and Lefteris Angelis. 2014. Reasons for bottlenecks in very large-scale system of systems development. *Information and Software Technology* 56, 10 (2014), 1403 – 1420.

[30] Davy Preuveneers et al. 2004. Towards an Extensible Context Ontology for Ambient Intelligence. In *European Symposium on Ambient Intelligence*. Springer, 148–159.

[31] Luca Sabatucci, Carmelo Lodato, Salvatore Lopes, and Massimo Cossentino. 2015. Highly Customizable Service Composition and Orchestration. In *Service Oriented and Cloud Computing*, Schahram Dustdar, Frank Leymann, and Massimo Villari (Eds.). Springer International Publishing, Cham, 156–170.

[32] M. Soliman, T. Abiodun, T. Hamouda, J. Zhou, and C. H. Lung. 2013. Smart Home: Integrating Internet of Things with Web Services and Cloud Computing. In *CloudCom*, Vol. 2. 317–320.

[33] Jean-Baptiste Soyez, Gildas Morvan, Rochdi Merzouki, and Daniel Dupont. 2017. Multilevel Agent-Based Modeling of System of Systems. *IEEE Systems Journal* 11, 4 (Dec 2017), 2084–2095.

[34] András Varga and Rudolf Hornig. 2008. An overview of the OMNeT++ simulation environment. In *Conf. on Simulation Tools and Techniques for Communications, Networks and Systems*. ICST, 60.

[35] Bahtijar Vogel and Dimitrios Gkouskos. 2017. An Open Architecture Approach: Towards Common Design Principles for an IoT Architecture. In *European Conf. on Software Architecture (ECSA)*. 85–88.

[36] W3C SSN XG. 2011. Review of Sensor and Observations Ontologies. https://www.w3.org/2005/Incubator/ssn/wiki/Review_of_Sensor_and_Observations_Ontologies.

[37] K. Xu, X. Wang, W. Wei, H. Song, and B. Mao. 2016. Toward software defined smart home. *IEEE Communications Magazine* 54, 5 (May 2016), 116–122.

[38] Ibrar Yaqoob, Ejaz Ahmed, Ibrahim A. T. Hashem, Abdelmuttlib I. A. Ahmed, Abdullah Gani, Muhammad Imran, and Mohsen Guizani. 2017. Internet of Things Architecture: Recent Advances, Taxonomy, Requirements, and Open Challenges. *IEEE Wireless Communications* 24, 3 (2017), 10–16.

[39] Xinfeng Ye. 2006. Towards a Reliable Distributed Web Service Execution Engine. In *International Conference on Web Services (ICWS âĂŹ06)*. IEEE Computer Society, USA, 595âĂŞ602. https://doi.org/10.1109/ICWS.2006.131