(Article begins on next page)

An architecture for context-aware reactive systems based on run-time semantic models

Ester Giallonardo [1] , **Francesco Poggi** [Corresp., 2] , **Davide Rossi** [2] , **Eugenio Zimeo** [1]

[1] Department of Engineering, University of Sannio, Benevento, Italy

[2] Computer Science and Engineering (DISI), University of Bologna, Bologna, Italy

Corresponding Author: Francesco Poggi
Email address: fpoggi@cs.unibo.it

In recent years, new classes of highly dynamic, complex systems are gaining momentum. These systems are characterized by the need to express behaviors driven by external and/or internal changes, i.e. they are reactive and context-aware. These classes include, but are not limited to IoT, smart cities, cyber-physical systems and sensor networks.

An important design feature of these systems should be the ability of adapting their behavior to environment changes. This requires handling a runtime representation of the context enriched with variation points that relate different behaviors to possible changes of the representation.

In this paper, we present a reference architecture for reactive, context-aware systems able to handle contextual knowledge (that defines what the system perceives) by means of virtual sensors and able to react to environment changes by means of virtual actuators, both represented in a declarative manner through semantic web technologies. To improve the ability to react with a proper behavior to context changes (e.g. faults) that may influence the ability of the system to observe the environment, we allow the definition of logical sensors and actuators through an extension of the SSN ontology (a W3C standard). In our reference architecture a knowledge base of sensors and actuators (hosted by an RDF triple store) is bound to real world by grounding semantic elements to physical devices via REST APIs.

The proposed architecture along with the defined ontology try to address the main problems of dynamically reconfigurable systems by exploiting a declarative, queryable approach to enable runtime reconfiguration with the help of (a) semantics to support discovery in heterogeneous environment, (b) composition logic to define alternative behaviors for variation points, (c) bi-causal connection life-cycle to avoid dangling links with the external environment. The proposal is validated in a case study aimed at designing an edge node for smart buildings dedicated to cultural heritage preservation.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27702v1 | CC BY 4.0 Open Access | rec: 4 May 2019, publ: 4 May 2019

An architecture for context-aware reactive systems based on run-time semantic models

Ester Giallonardo[1], **Francesco Poggi**[2], **Davide Rossi**[2], **and Eugenio Zimeo**[1]

[1]**Department of Engineering, University of Sannio, Italy**
[2]**Department of Computer Science and Engineering (DISI), University of Bologna, Italy**

Corresponding author:
Francesco Poggi[1]

Email address: francesco.poggi5@unibo.it

ABSTRACT

In recent years, new classes of highly dynamic, complex systems are gaining momentum. These systems are characterized by the need to express behaviors driven by external and/or internal changes, i.e. they are reactive and context-aware. These classes include, but are not limited to IoT, smart cities, cyber-physical systems and sensor networks.
An important design feature of these systems should be the ability of adapting their behavior to environment changes. This requires handling a runtime representation of the context enriched with variation points that relate different behaviors to possible changes of the representation.
In this paper, we present a reference architecture for reactive, context-aware systems able to handle contextual knowledge (that defines what the system perceives) by means of virtual sensors and able to react to environment changes by means of virtual actuators, both represented in a declarative manner through semantic web technologies. To improve the ability to react with a proper behavior to context changes (e.g. faults) that may influence the ability of the system to observe the environment, we allow the definition of logical sensors and actuators through an extension of the SSN ontology (a W3C standard). In our reference architecture a knowledge base of sensors and actuators (hosted by an RDF triple store) is bound to real world by grounding semantic elements to physical devices via REST APIs.
The proposed architecture along with the defined ontology try to address the main problems of dynamically reconfigurable systems by exploiting a declarative, queryable approach to enable runtime reconfiguration with the help of (a) semantics to support discovery in heterogeneous environment, (b) composition logic to define alternative behaviors for variation points, (c) bi-causal connection life-cycle to avoid dangling links with the external environment. The proposal is validated in a case study aimed at designing an edge node for smart buildings dedicated to cultural heritage preservation.

INTRODUCTION

Reactive systems bonds actuating (what is performed by the system) and sensing (what is perceived by the system) with a reactive behavior that represents the logic driving the application. Examples of such systems can be very diverse and present a large variation in complexity. They span from simple open loop systems, such as a domotics one in which when a light sensor reports a reading below a given threshold a light switch actuator is fired, to very complex systems such as a production line support one in which when an AI-based analyzer feeded by a time series of observations produced by IoT activity sensors predicts that a machine in a line is going to need maintenance shortly, a bypass actuator is fired to activate a backup production line and allow to perform maintenance on the main line.

What these systems are able to sense (or to act on) constitutes their *context*, and since their behavior depends on it we also call them *context-aware*. In fact, according to (Furno and Zimeo, 2014), context is the state (variable and corresponding values) that a system is able to access to or modify. This state is the set of variables that are possibly shared with other systems: they can be read or modified by users, devices or applications other than the one the state is referred to. While context represents the state that influences

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27702v1 | CC BY 4.0 Open Access | rec: 4 May 2019, publ: 4 May 2019

an entity, sensing is the process needed to capture the environmental information that contributes to define the context.

Depending on the relationship between context and application, we are presented with a spectrum ranging from simple reactive systems (the application logic is immutable but is able to change the context) to self-adaptive ones (the application logic can change according to the context) (Cheng et al., 2009; De Lemos et al., 2013). All these applications share the need to reason upon context at runtime, and can benefit from a flexible, expressive and queryable representation of context. The structure of this model can be very simple (e.g. a collection of variables representing the latest observations reported by sensors) or very articulated (e.g. a megamodel, as the model of models proposed for self-adaptive systems in (Vogel and Giese, 2014)). When dealing with different representations of runtime models, we end up with systems whose behavioral elements are bound to these diverse encodings and strongly depend on them leading to unwanted brittleness that is particularly exposed when these models evolve to react to unplanned events of the context.

To avoid this problem, we propose a uniform representation of contextual models based on Semantic Web languages. This choice not only improves interoperability but also promotes the adoption of declarative approaches for context-aware behaviors definition. This approach plays nicely with the aforementioned self-adaptation property since it allows to change the system's behavior during the execution, allowing (potentially unplanned) adaptations by operating at the model level. Therefore, our first contribution is a base vocabulary to model the fundamental items constituting a reactive context aware system: sensing, actuating and reactive behavior. This vocabulary, named LSA (detailed later), is expressed in the form of an OWL ontology. Notice that we do not propose to use this ontology to represent all elements in the runtime model: different contextual domains can refer to very diverse concepts and specific ontologies should be used to represent them. LSA is designed to embody the basic reactive aspects while cooperating with other domain ontologies to fully describe contextual information.

The second contribution is a reference architecture for context-aware reactive systems that makes use of a semantic knowledge base to keep a live, queryable and updatable representation of the runtime model in which the reactive elements are encoded using the aforementioned ontology. The model represents the physical world the system interacts with and is enriched and modified with the data coming from the sensors, assuring consistency with the physical elements it represents. The knowledge base is extended with the machinery needed to interact with physical sensors and actuators and activate reactive behaviors so that not only basic reactive mechanisms can be implemented but it is also possible to ensure that model is bi-causally connected (Hölzl and Gabor, 2015). When this happens modifications of the model causes the enactment of actuators to materialize these modifications in the physical world.

The specific problems that we address can be solved with existing solutions, since self-adaptive and self-healing systems using rich runtime models already exist and the same can be said for refined systems support bi-causally connected models. However, our aim is to propose a reference architecture, able to meet the aforementioned requirements, based on standard languages and tools of the Semantic Web that supports declarative approaches to behavior definition, is well-focused, consistent and, possibly, elegant. The proposed reference architecture can be declined in different ways to better meet specific needs. For example a system dealing with a large number of IoT devices producing a continuous flow of readings needs to address problems such as the ability to efficiently operate on large streams of semantic data (e.g. by adopting languages and tools for semantic stream processing as in the autonomic approach proposed in Dautov et al. (2014)) whereas a smart domotic system could introduce elements of reasoning operating on historical semantic data sets.

To explain our approach, the overall architecture and the proposed ontology, we present a detailed scenario related to a case study in the domain of smart buildings hosting cultural heritage. In this context we propose one possible instantiation of the architecture based on Jena, OWL, and SPARQL, for the knowledge base, and RESTful services, for the interaction with the physical world. We show that by the LSA ontology, a high-level external property that enables software adaptation can be easily handled through the definition of a related logical sensor built atop other logical sensors or simple virtual representations of physical sensors.

To summarize, our proposal consists of:

- a reference architecture for context-aware reactive systems based on a semantic knowledge base extended with the machinery to support bi-causal models connection defined with a declarative behavioral notation that exploits the queryability of the runtime model. These behavioral elements

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27702v1 | CC BY 4.0 Open Access | rec: 4 May 2019, publ: 4 May 2019

are included in the runtime model themselves and can be subject to modification after the initial deployment of the system;

- a kernel ontology to represent the basic concepts at the roots of context-aware reactive systems: sensors, actuators and reactive behavior. The reference architecture makes use of this kernel ontology for the reactive elements of the runtime model, including the aforementioned behavioral aspects.

The remainder of this paper is organized as follows. Section "Related Work" presents the related work from both research and standardization points of view. Section "Semantic Context Model and Logical Entities" introduces the SSN ontology, identifies its limitations with reference to the definition of complex and runnable sensors/actuators behaviors and presents the LSA ontology. Section "Reference Architecture" describes the reference architecture proposed with this paper for implementing infrastructures for context-aware applications. Section "Case Study: A Resilient Smart Building" shows the LSA ontology in action to implement an edge node for smart buildings hosting chultural heritage. Section "Prototype" describes a possible instantiation of the reference architecture. Finally, Section "Conclusions and Future Work" concludes the paper and highlights future work.

RELATED WORK

Notable examples of context-aware systems include Internet of Things (IoT), smart cities and cyber-physical systems that propose several scenarios characterized by a high level of dynamism and heterogeneity. In these scenarios, software adaptation can be used to face dynamic changes (Abowd et al., 1999; Baresi and Sadeghi, 2018). Various recent research works take the idea of using models as central artifacts to cope with dynamic aspects of ever-changing software and its environment at runtime. For instance, ContQuest (Pötter and Sztajnberg, 2016) is an approach to dynamically integrate devices into a context-aware IoT environment, and DYNAMICO (Tamura et al., 2013) introduces an infrastructure for self-adaptive systems with context-awareness requirements. Szvetits et al. (Szvetits and Zdun, 2016) comprehensively survey these kind of approaches for adaptive context-aware systems highlighting the common idea of establishing semantic relationships between executed applications and runtime models based on monitoring events.

Some recent works propose approaches for context-aware systems based on runtime models able of supporting behavior definition. Angelopoulos et al. (2015) propose a methodology based on three variability models: goal models (to represent system requirements), behavioral models (by modeling possible sequences for goal fulfillment and task execution), and system architecture models (defined in terms of connectors and components). The behavior of the system is represented through *flow expressions* (Shaw, 1978) describing the flow of system behaviors in terms of extended regular expressions able to define sequential, alternative or optional flows, and their cardinality. Behaviors are connected to system goals, and Behavioral Control Parameters (BCP) define multiple alternative behaviors for fulfilling a goal (i.e. the possible values are all the allowed sequences).

More recently, the Tropos methodology (Bresciani et al., 2004) for requirement analysis and specification has been extended to develop context-aware reactive system, as discussed in Morandini et al. (2017). The proposed methodology, called Tropos4AS, combines goal-oriented concepts and high-variability design methods. Tropos4AS goal models formally defines the run-time behaviour for achieving a goal, but this formal definition of the behaviour has to be specified at the time of modelling. An environemental model makes explicit the dependencies between the agent's goals, which determine the agent's behaviour, and its environment. The reactive system uses these models to properly interpreting contextual information in order to decide about when to change its behaviour and which alternative behaviour to select. At run-time, a monitor-analyse-plan-execute loop realizes the adaptation by monitoring requirements satisfaction and making effective changes based on the knowledge modelled at requirements-time.

Another notable approach is RELAX (Whittle et al., 2009), a declarative requirements language for self-adaptive systems which supports the explicit expression of environmental uncertainty in requirements. The main challenge faced by this work is the difficulty to anticipate all the explicit states in which an adaptive system will be during its lifetime. The distributed nature of such systems and their changing environmental factors require the ability to tolerate a range of environmental conditions and contexts. RELAX is based on fuzzy branching temporal logic and provides modal, temporal and ordinal operators to express uncertainty imposed by changing environmental conditions, such as sensor failures, noisy

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27702v1 | CC BY 4.0 Open Access | rec: 4 May 2019, publ: 4 May 2019

networks, malicious threats, unexpected (human) inputs, etc. Example operators are SHALL to define functionality the system must always provide (invariants) and MAY/OR to define alternatives.

Most of the papers introduced before recognize the need for a run-time model of both system and context, enriched with a variability model for supporting adaptations. These two kinds of models should be semantically related since a change in the context model should be associated to a variability alternative to introduce into the current configuration of the system. According to these requirements, several efforts have tried to propose semantics to easily model and handle dynamic context-aware applications. The sensing level is considered in Frank (2001); Bettini et al. (2010) as level 0 of a possible semantic stack and contributes to create the context-awareness of an application or a computing system. At this level, context parameters are the ones directly measurable by sensors. They could regard: the physical environment, such as air temperature, humidity or pressure; the human body, such as blood pressure, heart frequency or body temperature; an entity, such as location, acceleration, direction; the execution environment of a computer system, such as number of available CPUs, available memory or disk space. Atop sensing, context models are defined by enriching the limited semantics of the measured physical parameters with additional knowledge that models the world (Pederson et al., 2008) or the specific situations that influence an application or a computing system. Therefore, context modeling requires specific languages that software engineers could use to improve the flexibility of software systems with the ability of adapting themselves to external changes.

One of the first ontologies was SOUPA (Chen et al., 2004). It is expressed in OWL and includes modular component vocabularies to represent agents and related aspects. More recently, the authors in Perera et al. (2014) have discussed the requirements that context modelling and reasoning techniques should meet, including the modelling of a variety of context information types and their relationships.

The recent diffusion of IoT also introduces the need to filter and reason about the data produced by the huge amount of deployed sensors and confirms the importance of context-awareness for many applications (Lefrançois, 2017). In this direction, the Web of Things (WoTs) is one of the major standardization effort. It aims at extending the concept of web service to devices, allowing a Web client to access the properties of local or remote devices, to request the execution of actions and to subscribe to events representing state changes (Kaebisch and Kamiya, 2017). The related ontology describes how to model physical or virtual environments, sensors and actuators, with the main objective of easing the binding among devices reachable through web protocols (REST, CoAP, etc.). In particular, each device can be modeled in terms of observable or actuable properties, interactions patterns enabling the correct communication, the type of messages exchanged (commands, observations, etc.). Therefore, WoT is more oriented to the interaction between physical and virtual environments rather than to behaviors modeling.

A different objective is pursued by the Semantic Sensor Network (SSN) ontology (Haller et al., 2017), an Open Geospatial Consortium (OGC)/World Wide Web Consortium (W3C) standard. It is mainly focused on the SOSA (Sensor, Observation, Sample, Actuator) pattern (Janowicz et al., 2018) to model reactive systems. Therefore it aims at supporting the definition of simple reactive behaviors that link observations coming from modeled sensors with the related reactions performed by actuators. These behaviors are represented by RDF sub-graphs in a knowledge base and can be activated when observation facts are asserted. In order to link observations to physical or virtual properties, the SOSA pattern is extended with some system-oriented features. However, SSN does not directly support complex processing inside the knowledge base than asserting facts due to external sensing activities.

The Semantic Smart Sensor Network (S3N) ontology (Sagar et al., 2018) is a research effort that tries to specialize SSN by introducing subclasses and restrictions in order to support the modeling of smart sensors. To this end a new class, `s3n:SmartSensor`, has been introduced as a specialization of `ssn:System`. A smart sensor is composed of embedded sensors, microcontrollers and communicating systems. It is reprogrammable, reconfigurable and supports different communication and computation profiles. The behavior is expressed by the execution of an algorithm (selected among the existing ones on context basis) by the microcontroller, which can be thought as a specialization of the `ssn:Actuator`, being able to change the state of the whole smart sensor. The main purpose of S3N is to support smart sensors modeling and not to close the logical gap between sensors and actuators with behaviors more complex than simple external reactions.

Differently from the analyzed research contributions and standards, we propose a reference architecture for developing context-aware applications whose reactive behaviors can be defined by using an extension of a standard ontology (SSN), specifically designed to model device (sensors and actuators) behaviors.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27702v1 | CC BY 4.0 Open Access | rec: 4 May 2019, publ: 4 May 2019

The proposal tries to address the main problems of dynamically reconfigurable systems by exploiting a declarative approach to enable runtime reconfiguration with the help of (a) semantics to support discovery in heterogeneous environment, (b) composition logic to define alternative behaviors for variation points, (c) bi-causal connection life cycle to avoid dangling links with the external environment.

Our proposal is fully consistent with the Models@Run.time (Morin et al., 2009; Blair et al., 2009). A knowledge base is used to provide a runtime representation of the system and its environment which is bound to real world entities by grounding (mainly via web services) semantic elements to sensors and actuators. The behavior of the system can be specified by using sensing or actuating procedures tied to logical devices provided by the semantic model. These procedures can act upon the knowledge base by generating new facts or by redefining the structural parts of the model thanks to the declarative approach adopted.

SEMANTIC CONTEXT MODEL AND LOGICAL ENTITIES

In this section, we focus on semantic models to represent the context of reactive applications. As previously discussed, sensing represents the first layer of a semantic stack to create context-awareness of a computing system. A more complex perception of the external environment can be obtained by processing and aggregating different sensors observations. We perform this processing by introducing logical sensors and actuators as an extension of sensors and actuators provided by the SSN ontology. Therefore, we first describe SSN and then we present and discuss our proposal, the LSA ontology.

Semantic Sensor Network ontology

The SSN ontology was specifically designed for supporting interoperability between WoT entities taking into account performance and composition requirements. Web developers, in fact, have their concern about semantic approaches that do not assure near real time data processing. For this reason, its core module is constituted by the lightweight SOSA ontology that defines its concepts and properties through schema.org annotations desiderata from Linked Data engineers. The SSN main perspective is the system one.

Systems of sensors and/or actuators can be deployed on platforms for particular purposes. Actuators determine changes of the state of the world through the execution of procedures triggered by observations of properties. SSN does not fix restrictions on the way to implement procedures, allowing to describe any information that is provided to a procedure for its use (`ssn:Input`), and any information that is reported from a procedure (`ssn:Output`). Finally, sensors detect stimuli that originated observations, i.e. events that assign results to observable properties. Stimuli can be proxies for observations of properties related to features of interest. For example, infrared sensors respond to thermal stimuli detected from the environment. The thermal stimulus is a proxy for a live presence in the sensor zone, which represents the observable property of interest. In turn, this property could refer to a feature of interest.

The SSN ontology is very flexible. It identifies the main concepts that characterize systems. There is no distinction among specific instances of concepts and general instances representing classes of similar concepts. Consequently, it does not include a taxonomy of types for the identified concepts (i.e. Sensor, Actuator, Observation). For example, streams of observations can be stored defining `ssn:Observation` subtypes. Simple Knowledge Organization System (SKOS) (Miles and Bechhofer, 2009) vocabularies can be mapped to the entities, allowing for re-use of available domain ontologies. SKOS, in fact, allows providing documentation notes to RDF/RDFS concepts or relationships, or to OWL Classes.

Logical Sensors and Actuators ontology

The LSA ontology introduces two main concepts: (software) *logical sensors* and *logical actuators*. A (software) logical sensor (resp. actuator) is a sensor (resp. actuator) that generates observations (resp. actuations) as result of software procedures executions that use other observations as inputs. These sensors, and in particular the properties they refer to, are more directly related to software/physical adaptation, and in many cases can be derived from this requirement.

Both logical sensors and actuators are entities that live only in the virtual space (e.g. knowledge base) and are connected to the external world only through SSN simple sensors and actuators. For example, a (physical) light sensor represented by an SSN sensor could generate an observation ($light = 90LUX$) that should trigger an actuator for switching on a lamp. However, the decision logic (e.g. switch on

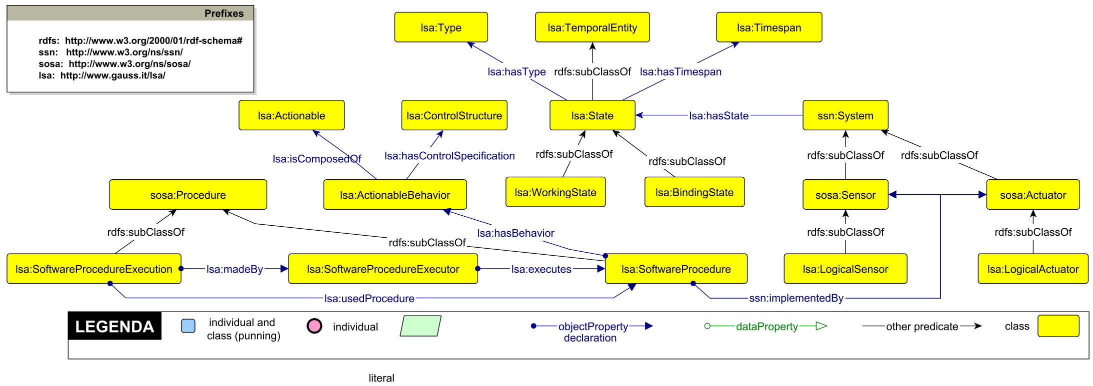

Figure 1. Core classes and properties of the Logical Sensors and Actuators (LSA) ontology.

if *light* $< 100LUX$) needed for closing the gap between the observation and the actuation can not be specified by simply using SSN, since neither the sensing procedure of the light sensor nor the actuation procedure can be programmed to decide to switch on the lamp or not. Moreover, SSN procedures are general concepts without any support for formalizing the execution steps that produce actual changes of the knowledge base. On the contrary, the composition logic of software procedures that we propose helps programmers (and reasoners) to semantically close the gap between observations and actuations, even with different implementations (useful, for example, for enabling reconfiguration).

Fig. 1 shows a Graffoo (Falco et al., 2014) diagram of the core elements of the Logical Sensors and Actuators (LSA) ontology[1]. Logical sensors and actuators are modeled with the classes `lsa:LogicalSensor` and `lsa:LogicalActuator`, which are subclasses of `sosa:Sensor` and `sosa:Actuator`, respectively. The behaviors associated to sensors/actuators are represented by the `lsa:SoftwareProcedure` class, and the property `ssn:implementedBy` is used to connect software procedures to sensors/actuators.

A `lsa:SoftwareProcedure` is a specific kind of `sosa:Procedure` with an actionable behaviour, described by executable code (via the `lsa:hasBehavior` property). A `sosa:Procedure` is defined in SSN as "a workflow, protocol, plan, algorithm, or computational method specifying how sensors make observations, or actuators make changes to the state of the world".

It is important to note that the LSA ontology does not impose constraints on how such behaviors should be represented. Another key point of the LSA ontology is that it allows to discern between:

- **procedures specifications**: the algorithm, workflow, protocol, etc. used by a sensor (actuator) to perform observations (actuations), along with a declaration of inputs and outputs. E.g. the algorithm used by a logical sensor that measures the perceived humidity (output) by aggregating a temperature and a humidity (input);
- **procedures executions**: the description of a specific execution of a procedure made by a sensor (actuator), which is carried out using a specific set of input values to produce a specific output. E.g. the perceived temperature X (output) of a room computed by using temperature Y and humidity Z as inputs.

In our pattern (which we aim at aligning with the ontology proposed in Lefrançois (2017)), a procedure execution is modeled with the `lsa:SoftwareProcedureExecution` class, and is related via the `lsa:usedProcedure` property to a `lsa:SoftwareProcedure`. The software procedure specifies the actionable behaviour (e.g. algorithm, workflow, protocol, etc.) manifested by the execution, and is perfomed (via the `lsa:madeBy` property) by a a `lsa:SoftwareProcedureExecutor` - a software agent able to execute it.

System states and life-cycle

The LSA ontology has been defined with the main objective of supporting adaptations at different levels of a context-aware application. In particular, higher-level adaptations need higher-level contextual information, that we can infer from the directly sensed ones. To support adaptation, we recognize the

[1]The Logical Sensor and Actuator ontology is available at https://sites.google.com/site/logicalsensorsactuators

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27702v1 | CC BY 4.0 Open Access | rec: 4 May 2019, publ: 4 May 2019

need of handling each device according to the working state that characterizes the ability of a sensor to correctly sense the environment and transmit the related samples, or the ability of an actuator to correctly act on the environment, changing its state as programmed.

Depending on the working state or on other applications-specific conditions, a system (sensor or actuator) can be detached from the physical counterpart to avoid the storage of altered observations in the knowledge base hosting the model. Therefore, LSA 1.1 version has been extended in order to observe the state of a `ssn:System` and to change it as a result of a meta-reaction.

`lsa:State` represents a unique defined condition of `ssn:System`, in a limited contiguous extent in time. The `lsa:hasState` property allows to associate a `lsa:State` to a `ssn:System`. A state has exactly one time-span. The `lsa:Timespan` includes temporal extents qualified by a beginning, an end or a duration. The `lsa:hasTimespan` property describes the temporal limitation of the temporal entity. The `lsa:beginningIsQualifiedBy`, `lsa:endIsQualifiedBy` and `lsa:hasDuration` datatype properties qualify respectively the beginning, the end and the duration of a time-span.

`lsa:State` is specialized in two main subclasses: `lsa:WorkingState` and `lsa:BindingState`. The former is related to the working condition of a system (e.g. it is normally working or faulty); the latter is referred to the bi-causal connection between physical sensors and their representation in the knowledge base. We claim that this state is important in order to correctly handle the life cycle of a system from the knowledge base point of view since a representation might be only descriptive or even active.

While `lsa:WorkingState` can be specialized in `lsa:NormalState` and `lsa:FaultyState`, `lsa:BindingState` can be `lsa:Inactive`, `lsa:Attached` or `lsa:Detached`. To express this specialization, we use `lsa:Type` to specify a hierarchy of terms, since we assume that each one of these specific state conditions can be described with the same properties and datatypes of `lsa:State`. A system is: *inactive* if we are interested only in its passive representation for registration purpose, *attached* when it is directly or indirectly bi-causally connected with a physical device, *detached* when it is temporally unconnected with a physical device.

According to the LSA ontology, a change of the Binding State of a System can be performed by some logical actuator, executing actuations as reactions of specific observations. To this end, `lsa:State` is a `sosa:ActuatableProperty`.

The described system life-cycle can be considered as an enabler of reconfiguration, especially when this implies to leave one or more devices. In that cases it is important to avoid dangling connections with devices that (a) could interfere with the ones used after the reconfiguration or (b) produce incorrect observations (errors) due to some fault. By combining `WorkingState` and `BindingState`, we enable self-healing, an important non-functional requirement that ensures system resilience (Delic, 2016), the capability to resist to external perturbations and internal failures, to recover and enter stable state(s), as we show in the next sections.

REFERENCE ARCHITECTURE

In this section we propose a reference architecture to implement reactive context-aware systems that make use of a semantic runtime model hosted in a knowledge base (an RDF triplestore). The sensing-behavior-activation elements of the runtime model are represented using the LSA ontology. To connect these elements of the model to the physical world, the knowledge base is continuously enriched and updated with the data coming from the sensors, assuring consistency with the physical elements it represents (that is, ensuring causal connection). A reactive mechanism is used to trigger virtual sensors and actuators, making it possible to also achieve bi-causal connections.

To exemplify these concepts just think about a simple reactive system immersed in an environment composed of a room containing a light bulb, a bulb actuator and a light sensor, and in which all these elements are represented in a virtualized form within the system. In a causally connected system the change of the state of the real-world light bulb (turned on/turned off) is reflected in the model element that represents the bulb within the system. In a bi-causally connected system, in addition to the aforementioned relationship, also the modification of the state of the model element is reflected as a change of state of its real-world counterpart. Thus if we set the state of the model element representing the light bulb to off while the real-world light bulb is turned on, this triggers an actuator to turn off the bulb.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27702v1 | CC BY 4.0 Open Access | rec: 4 May 2019, publ: 4 May 2019

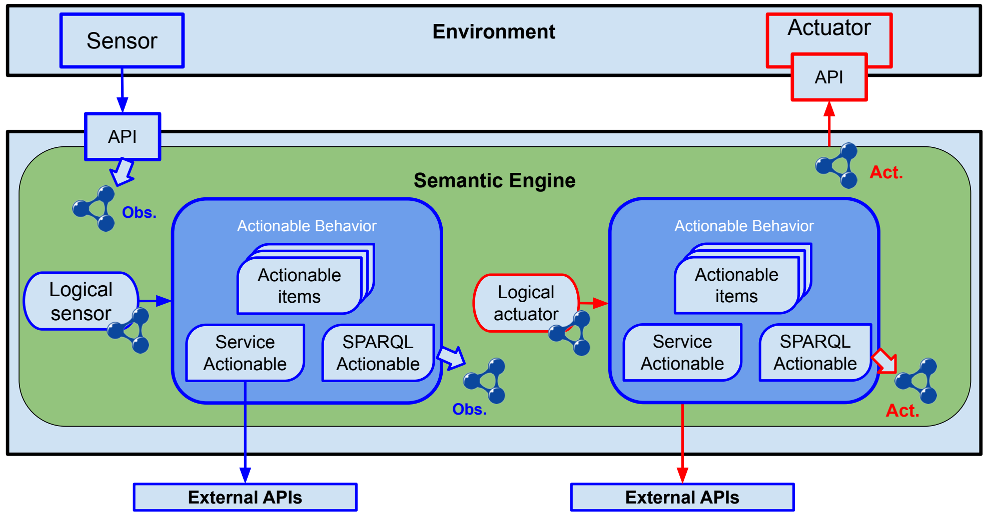

Figure 2. Architecture outline.

To achieve this behavior logical causal connections propagate updates throughout the knowledge base, and a binding mechanism mapping updates to actuators activation preserves the model alignment with real-world situations. Since this process expresses a form of application logic some kind of computational support is also needed.

In our architecture causal connections are supported by what are essentially rules in the form of logical sensors and actuators. We consistently represent these rules in the knowledge base itself: the activation part is modeled as Software Procedures (implementing the aforementioned computational support) associated to semantic sensors and actuators whereas the triggering logic is implemented by monitoring changes to the properties that are declared as inputs for these semantic sensors and actuators.

We consistently represent these rules in the knowledge base itself. The triggering logic is implemented by monitoring changes to the properties that are declared as inputs for these semantic sensors and actuators. The action part is modeled as Software Procedures (implementing the aforementioned computational support) associated to semantic sensors and actuators.

The basic component of our architecture is a semantic engine (see Fig. 2) whose elements are:

- a triplestore (and RDF database) hosting the semantic runtime model;
- a service API used to receive observations from external sensors (upper left side in the figure);
- a binding mechanism turning actuations facts/statements in the model (in the form of RDF statements) into actual invocation of remote actuators; this is realized by a component that monitors the triplestore for new actuations facts/statements and when they appear it invokes the corresponding actuators service endpoint (upper right side in the figure);
- a machinery to trigger logic sensors/actuators and execute their Actionable Behavior. This is realized by a component that monitors the triplestore for new observations pertaining to properties that are declared as inputs for Software Procedures associated to sensors/actuators. When these observations appear the sensor/actuator is activated and its related Software Procedure is executed (as defined by its Actionable Behavior), producing new facts (observations or actuations).

In this approach both the model of the external context and that of the system (in terms of logical sensors/actuators and their behaviors) is represented in a semantic format (e.g. by RDF triples). This allows to change the overall behavior of the system by manipulating the knowledge base: at runtime new logical sensors can be defined, the behavior of the existing ones can be modified, existing sensors/actuators can be deleted. A further advantage of this architecture is that self-adaptive behaviors can easily be implemented by simply allowing the software procedure of a sensor/actuator to work as described in Poggi et al. (2016); Rossi et al. (2018).

As stated above logical sensors and actuators have Software Procedures that are associated to their Actionable Behavior, that is a computation producing an observation (or actuation) that is added to the knowledge base. This computation usually operates on information coming from the contextual model, so it should be able to query the model in order to retrieve relevant data, and to insert RDF triples in the triplestore (representing the produced observation/actuation). A straightforward technology to realize these tasks is the SPARQL query language, its use is also aligned with our requirement of using standard Semantic Web languages and technologies when possible. Consider, for example, a logical sensor that produces a new apparent temperature observation whenever an update is produced by the physical sensors for temperature or humidity. The semantic engine will observe that temperature and humidity obervations (for a given place) are declared as inputs for the actionable behavior of the logical sensor. Whenever a new observation pertaining these properties will be inserted in the triplestore, the logical sensor will be activated and its actional behavior executed. In this case a simple CONSTRUCT (or INSERT) SPARQL query can be used: the query retrieves the latest observations related to temperature and humidity, combines them with a simple formula, and produces an RDF graph representing a new apparent temperature observation. Not always, however, the computation required is a simple linear combination of existing data, so we cannot assume that SPARQL can be used to implement all Actionable Behaviors. For this reason, we generally expect that this behavior is a combination of various computations (actions) performed by local or remote software components. Among these actions one or more can use SPARQL to retrieve data from the triplestore and to produce the RDF graph for the observation (or the activation). To get back to the previous example: if we have a remote service implementing a ”very sophisticated AI-based algorithm” to calculate the apparent temperature, we can use a SPARQL query action to retrieve the input data needed by the remote service from the contextual model, followed by a remote service invocation action performing the required computation, followed by a SPARQL query to create an RDF observation with the value returned by the remote service. The specific way in which this combination of actions is described is outside the scope of the reference architecture. Specific instantiations can choose a representation that better suits their needs. As previously discussed examples of existing ontologies that can be used includes OWL-S (the processes part) and BPMN ontologies. The figure contains general references to actionable items suggesting that some can invoke external services and some can interact with the knowledge base using SPARQL.

CASE STUDY: A RESILIENT SMART BUILDING

We consider a running example, extracted from a more general context of cultural heritage preservation (Giallonardo et al., 2017). In particular, we suppose that in a museum a new temporary exhibition is arranged. In a room of this exhibition a multimedia content has to be played. A solution that is often adopted is to play the content cyclically, through monitors or projectors; one of the possible drawbacks of this approach is that, in the absence of an adequate organization of groups, visitors who arrive at any moment in time have to wait for a subsequent delivery of the contents. The organizers of the exhibition express the desire for a more refined behavior in which the content starts playing when visitors enter the room, and is stopped when the room is empty.

We assume that the museum rooms are equipped with both specific physical sensors able to detect people presence, and surrogate ones based on a logical composition of other kinds of sensors. Specifically, these logical sensors can be opportunistically defined by exploiting InfraRed (IR) detectors close to the doors (used as part of the anti-theft system). The knowledge base of our edge node is populated with both specific presence sensors and the logical ones. All the equivalent sensors that are able to perceive people presence in a specific room are tied to the same observable property.

To explain the ontology, we first analyze how LSA allows a designer to model logical sensors and actuators, assuming that in a room the working presence sensor is the one based on InfraRed detectors (see Fig. 3).

Multimedia playback control based on a logical presence sensor

We initially consider the following scenario:

1. a tourist crosses the door of the museum, and the two physical infrared sensors on the two door sides produce two observations about the presence of a person in their detection areas;

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27702v1 | CC BY 4.0 Open Access | rec: 4 May 2019, publ: 4 May 2019

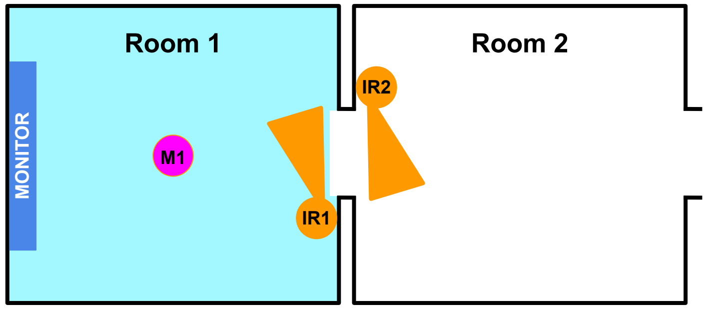

Figure 3. Museum layout.

2. a logical sensor aggregating such observations produces another observation updating the number of persons present in the rooms (i.e. n-1 in the room left by the tourist, m+1 in the room the tourist entered);

3. if the tourist enters an empty room an actuator starts to play a multimedia flow on the room monitor; if the tourist is the last person that leaves a room before the end of the playback, an actuator will stop the multimedia flow. In both cases, the information about the new actuation is inserted in the triple store.

Before describing in detail the aforementioned steps, we show how we modeled behavioral information about sensors and actuators in the context of this case study.

Fig. 4 depicts the actionable behavior (`gmus:doorRoomEntrance/behavior/1/actionable`, `gmus:doorRoomEntrance/behavior/2/actionable`) that has been defined for `gmus:DoorRoomEntrance`, the software procedure that logical infrared sensors (`gmus:infraredPresenceSensor`) implement.

The `gmus:doorRoomEntrace/behavior` is composed of executable actions (i.e. individuals of the `lsa:Actionable` class) and of an objectively recognizable control structure based on a `lsa:List` (to define a sequence of actions), using the `lsa:hasControlSpecification` property.

In the example there are two actions: `gmus:doorRoomEntrance/behavior/1/actionable`, which is a SPARQL query that is defined as executable by `gmus:sparqlQueryEngine`, a specific type of `lsa:QueryEngine`; and `gmus:doorRoomEntrance/behavior/2/actionable`, a REST action that is defined as executable by `gmus:restRequestEngine`, a specific type of `lsa:RequestEngine`. Both `gmus:sparqlQueryEngine` and `gmus:restRequestEngine` are specific types of `lsa:SoftwareProcedureExecutor`, and implement `gmus:doorRoomEntrance/behavior` (as defined by the process execution pattern defined in the LSA ontology).

1. Observations made by physical sensors: Fig. 5 shows the RDF statements that are added to the triplestore by the semantic engine when a person crosses a door. Whenever this occurs, the infrared sensors placed on the two sides of the door detects the presence of a person and invoke the engine REST API in sequence (providing their ids and the instants of time when the observations occurred as request parameters).

Two observations (i.e. `gmus:observation/ir1/1` and `gmus:observation/ir2/1`) made by sensors `gmus:ir1` and `gmus:ir2` are produced, which relate to the same feature of interest (i.e. `gmus:door1`). Each observation concerns a distinct observable property (i.e. the presence in the detection area of the each sensor - `gmus:presence/room1/ir1/zoneDoorInside` and `gmus:presence/room2/ir2/zoneDoorOutside`), and keeps track of the time in which the observations were performed.

It is important to note that in these examples we make use of punning[2], an OWL metamodeling capability that allows to treat elements of the model as classes and individual as the same time.

Elements with this double nature are represented as light blue squares in the diagram. This has been used in Fig. 5, for instance, to model the concept of infrared sensor (`gmus:IRSensor`), which is at

[2] See https://www.w3.org/TR/owl2-new-features/#F12:_Punning

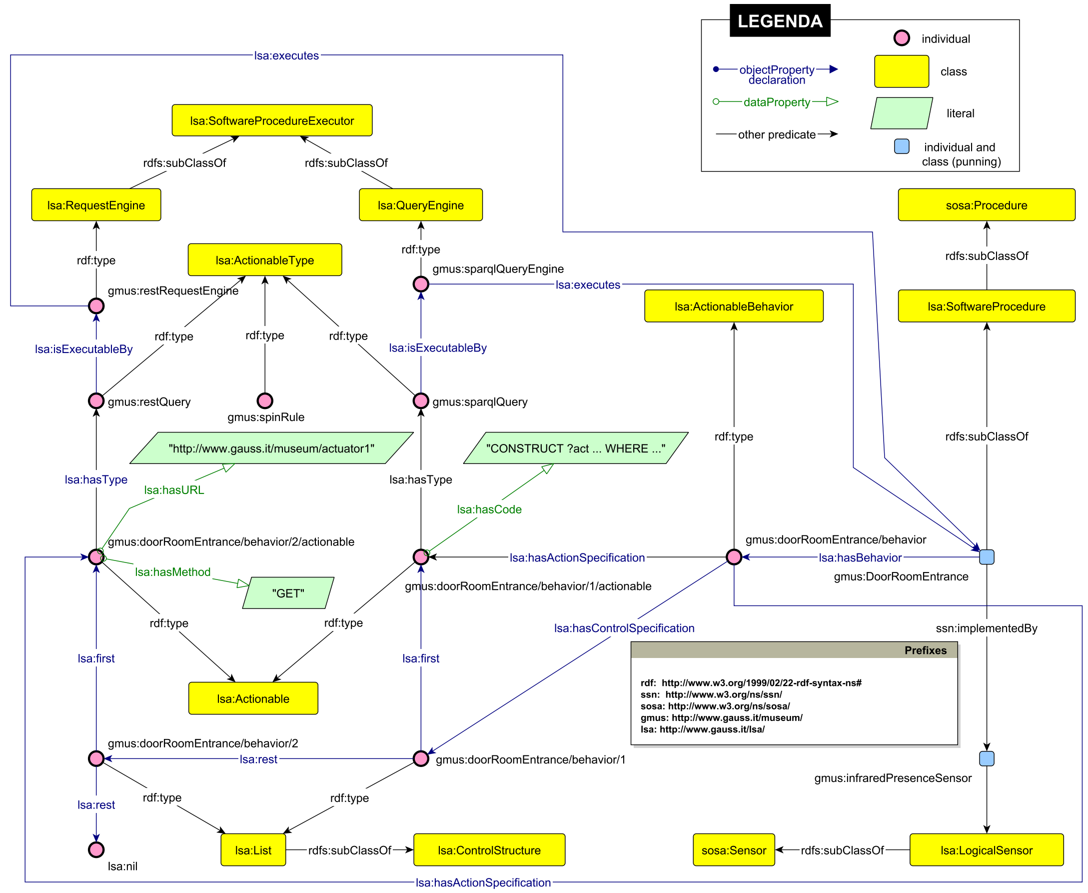

Figure 4. Logical sensor behaviour specification.

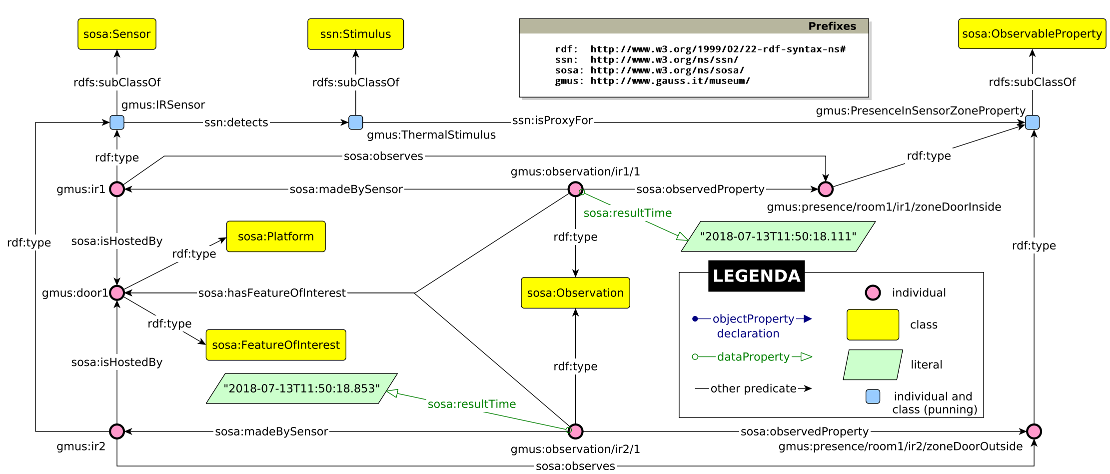

Figure 5. Observations made by two infrared sensors.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27702v1 | CC BY 4.0 Open Access | rec: 4 May 2019, publ: 4 May 2019

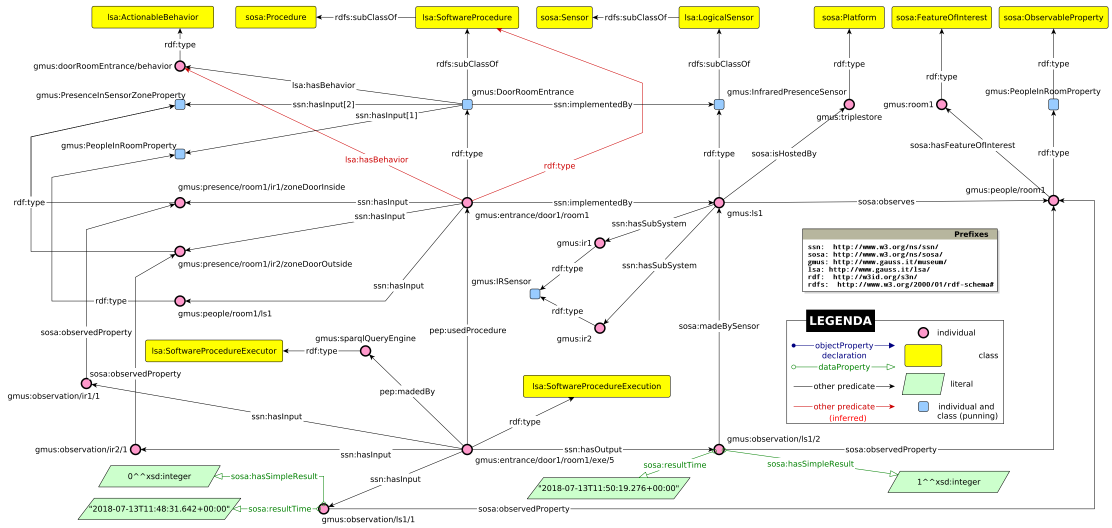

Figure 6. Observations made by the logical presence sensor. Square brackets are used to specify property cardinality restrictions.

the same time a class (i.e. a specific subclass of sensors representing infrared sensors) and an individual (since it is connected with `gmus:ThermalStimulus` by the `ssn:detects` property). In the same way, `gmus:PresenceInSensorZoneProperty` is a type of observable property (i.e. subclass of `sosa:ObservableProperty`) and an individual (connected to `gmus:ThermalStimulus` by the `ssn:isProxyFor` property). This approach is also useful to model logical sensors behaviors, as described in the rest of this section.

2. Observations made by logical sensors: Whenever a modification occurs in the triplestore (e.g. the insertion of a new observation), the semantic engine checks if one or more procedures specifying the behaviors of logical components (i.e. logical sensors and actuators) should be executed. To do so, the engine checks if the properties related to the new observations (e.g. `gmus:presence/room1/ir1/zoneDoorInside` and `gmus:presence/room2/ir2/zoneDoorOutside` in the previous example) are specified as inputs of one or more software procedures. As depicted in Fig. 6, such properties are inputs of the `gmus:entrance/door1/room1` procedure (as specified by `ssn:hasInput`), which is hence executed by the semantic engine. Such procedure is implemented by the logical sensor `gmus:ls1`, a specific instance of `gmus:infraredPresenceSensor` (the class representing logical presence sensors) hosted by the triplestore (`gmus:triplestore`) and observing the presence in a specific room (`gmus:people/room1`).

A general mechanism is adopted by the semantic engine to retrieve behavioral information (e.g. a sequence of activities to perform) pertaining logical sensors. Since behavioral information are shared by all logical sensors of a kind, the engine identifies the related software procedure (`gmus:DoorRoomEntrance` in our case) and retrieves the behavioral specification (`gmus:doorRoomEntrance/behavior`) by navigation the `lsa:hasBehavior` property. Alternatively, such behavioral specification can be inferred by a reasoner, since `gmus:DoorRoomEntrance` has been defined as a subclass of `lsa:SoftwareProcedure` having `gmus:doorRoomEntrance/behavior` as behavior (through an OWL membership restriction on the `lsa:hasBehavior` property).

Such behavioral specification in this case is composed of two actions, i.e. two SPARQL CONSTRUCT queries checking the entrance/exit in/from the room, respectively. Each of these queries retrieves the new observations made by the two infrared sensors, and if they have been performed in a short time interval - e.g. one second - produces:

1. a new software procedure execution (`gmus:entrance/door1/room1/exe/5`), connected to the software procedure (`gmus:entrance/door1/room1/`) by the `lsa:usedProcedure` property, and to the observations used as input (those made by the two infrared sensors and those

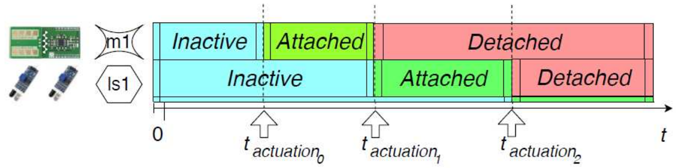

Figure 7. States produced as results of actuations during systems lifecycles.

pertaining the number of persons in the rooms connected by the door[3]) and the software procedure executor (`gmus:sparqlQueryEngine`) by the `ssn:hasInput` and `lsa:madeBy` property, respectively;

2. two new observations as output of the procedure execution[3], represented using the `ssn:hasOutput` property. For instance, the number of people in the first room has been update from zero (in `gmus:observation/ls1/1`) to one (in `gmus:observation/ls1/2`) since a person entered the room.

3. Actuations made by logical actuators: the newly added statements (i.e. those about the observations produced by the logical sensor `gmus:ls1` and the relative procedure executions) trigger another control performed by the semantic engine to check logical sensors/actuators interested to those observations.

In our example, the logical actuators controlling the video playback on the monitor in the room is activated, and the related software procedures is retrieved and executed. In this case the behavioral specifications are composed of two actions: a REST action that invokes the physical actuator API (to start the video playback on the monitor since a person has entered an empty room) and SPARQL INSERT query adding information in the triplestore about the performed actuation.

System reconfiguration

Non functional requirements are particularly important for context-aware systems because they usually impact the overall architecture of the system, whereas functional requirements can often be met with behavioral extensions of existing components (something that can be addressed at real-time, for example, with a plugin architecture). In this section, we show how the declarative approach we have presented before is very useful for (a) dynamically re-configuring our context-aware system with virtual or logical sensors / actuators that can be not known at design time; (b) extending our system with additional logic. We still make use of our case study about smart buildings for cultural heritage preservation but in this case we assume that specific microwave occupancy sensors are deployed within the exhibition rooms and are used to drive the switching of the multimedia presentations. After the deployment, the administrators of the system realize that presence can also be obtained by combining the anti-theft infrared sensors at the doors of the rooms, especially in case of malfunctioning of the microwave sensor. So they decide that this workaround can be activated as a backup.

Failure detection is not in the scope of this paper and, for simplicity, here we assume that presence sensors are battery operated and that they produce a specific observation about themselves when the battery is critically low before going offline. Implementing self-healing in this case is a two steps process: in the first step new virtual sensors are synthetized from existing physical sensors to report the presence in the rooms; in the second step a mechanism to replace failing sensors with available alternatives is put in place. As we will show this mechanism does not need to know in advance if and which replacement sensors are available, but can query the knowledge base to retrieve information about available alternatives.

To avoid interference among equivalent sensors, we assume that backup sensors are initially in the *inactive* state. System reconfiguration can be performed by a specific virtual actuator that is activated whenever an observation about a failure of an operating sensor is reported: when this happens the actuator queries the knowledge base to obtain a list of available sensors able to observe the same observable property of the failing sensor. If all sensors in this list are not active one is chosen (on the basis of some kind of policy: preference-based, round-robin, random) and activated.

Two actuations related to the activation of the new sensor and the deactivation of the failed one are then produced, as illustrated in Fig. 7. The figure shows that the `lsa:State` of the "m1" presence sensor

[3]Because of space limitations in the diagram we depicted only the observations about a room (i.e. we omitted the observations about the number of people in gmus:room2)

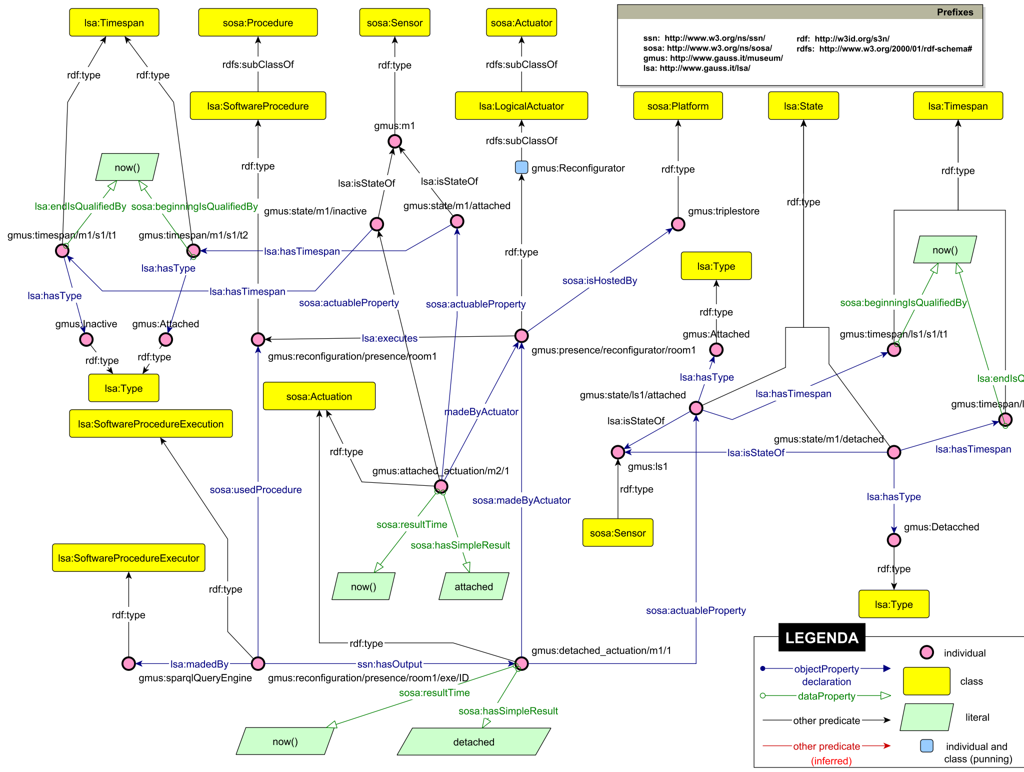

Figure 8. Core LSA classes and properties exploited during the system reconfiguration.

`gmus:m1` is "Inactive" at t=0. After the attaching/removing actuations performed by the reconfiguration system at $t = t_{actuation_1}$, the "Attached" state of the `gmus:m1` sensor terminates and a new "Detached" state instance comes into existence for it. A new "Attached" state is created for the `gmus:m2` sensor.

In our example, a virtual sensor combining the infrared ones is activated so it will begin producing observations allowing the system to be automatically kept operational also in presence of failures. The overall snapshot of the interested portion of knowledge base during reconfiguration is shown in Fig. 8, whereas the actionable code of the `gmus:presence/reconfigurator/room1` reconfiguration meta-actuator is reported in Fig. 9. It reacts to whatever observation produced by a failure detector as the one reported in Fig. 10. The reconfiguration actuator accesses to the observable properties of a system to change the `BindingState` of the system.

This example shows that within the proposed architecture the system can perform self-modifications in a reactive manner. Please notice that the strategy we propose here is not meant as a general solution to address self-healing for sensor-based systems, it is just a very specific solution aiming at showing the advantages of an architecture based on a queryable run-time model to perform non-trivial run-time modifications of relevant aspects of the system.

PROTOTYPE

For this case study we created a prototype instantiating the reference architecture described in Sect. 'Architecture'. The prototype[4] is a Java application that makes use of Apache Jena triplestore and SPARQL engine, as depicted in Fig. 11.

The API receiving sensors observations is a REST API written in JAX-RS; this API is state-aware, this means that only updates coming from sensors whose current state in the knowledge base is Attached are converted in their semantic counterparts.

The reactive machinery is implemented in the form of a transaction listener attached to the triplestore (current Jena implementation of transactions is limited so we had to adopt a couple of workarounds not worth to be detailed here). This component monitors the knowledge base for observations in order to execute the Actionable Behavior of virtual sensors and actuators when needed.

[4]https://github.com/cars-team/semanticengine

```
INSERT{

?this_software_p_exe    a lsa:SoftwareProcedureExecution;
                        ssn:hasOutput ?this_detached_act;
                        ssn:hasOutput ?this_attached_act;
                        sosa:usedProcedure ?this_software_p;
                        lsa:madeBy gmus:sparqlQueryEngine.
gmus:sparqlQueryEngine  lsa:executes <http://example.museum.gauss.it/reconfiguration/presence/room1>.
?this_detached_act      a sosa:Actuation.
?timespan               lsa:endIsQualifiedBy ?now.
?this_state             a lsa:BindingState;
                        lsa:hasType <http://example.museum.gauss.it/Detached>;
                        lsa:hasTimespan ?this_state_timespan.
?this_state_timespan    a lsa:Timespan.
?this_timespan          lsa:beginningIsQualifiedBy ?now.
?this_detached_act      sosa:hasResult ?this_state;
                        sosa:hasResult ?state.
?this_attached_act      a sosa:Actuation.
?this_state_updated     a lsa:BindingState.
?backup_sensor          lsa:hasState ?this_state_updated.
?this_state_updated     lsa:hasType <http://example.museum.gauss.it/Attached>.
?this_update_timespan   a lsa:Timespan.
?backup_sensor          lsa:hasTimespan ?this_update_timespan.
?this_update_timespan   lsa:beginningIsQualifiedBy ?now.
?backup_timespan        lsa:endIsQualifiedBy ?now.
?backup_sensor          lsa:hasState ?this_backup_state.
?this_backup_state      a lsa:State.
?backup_sensor          lsa:hasTimespan ?this_backup_sensor_timespan.
?backup_sensor_timespan lsa:beginningIsQualifiedBy ?now.
?this_backup_state      lsa:hasType <http://example.museum.gauss.it/Attached>.
?backup_sensor_timespan a lsa:Timespan.
?this_attached_act      sosa:hasResult ?this_state_updated;
                        sosa:hasResult ?backup_state.
} WHERE {

?this_software_p        ssn:implementedBy <http://example.museum.gauss.it/presence/reconfigurator/room1>.
?observation            a sosa:Observation;
                        sosa:madeBySensor <http://example.museum.gauss.it/faultDetector/room1>;
                        sosa:hasResult ?result;
                        sosa:hasFeatureOfInterest ?sensor.
?result                 a lsa:FaultyState.
?sensor                 sosa:observes ?p;
                        lsa:hasState ?state.
?state                  lsa:hasType <http://example.museum.gauss.it/Attached>;
                        lsa:hasTimespan ?timespan.
?backup_sensor          sosa:observes  ?p;
                        lsa:hasPriority "2"^^xsd:integer;
                        lsa:hasState ?backup_state;
                        lsa:hasType <http://example.museum.gauss.it/Inactive>;
                        lsa:hasTimespan ?backup_timespan.

BIND(UUID() AS ?this_software_p_exe). BIND(UUID() AS ?this_detached_act). BIND(UUID() AS ?this_state).
BIND(UUID() AS ?this_state_timespan). BIND(UUID() AS ?this_state_updated). BIND(UUID() AS ?this_update_timespan).
BIND(UUID() AS ?this_backup_sensor_timespan). BIND(UUID() AS ?this_act). BIND(UUID() AS ?this_backup_state).
BIND(UUID() AS ?this_attached_act). BIND(now() AS ?now).

}
```

Figure 9. SPARQL Actionable of the software procedure implemented by the Reconfigurator.

```
INSERT {

  <http://example.museum.gauss.it/observation/m1/state/1> rdf:type owl:NamedIndividual, sosa:Observation;
                                                          sosa:hasFeatureOfInterest gmus:ls1;
                                                          sosa:madeBySensor <http://example.museum.gauss.it/faultDetector/room1>;
                                                          sosa:observedProperty <http://example.museum.gauss.it/workingState/ls1>;
                                                          sosa:hasResult ?this_faulty_state.

  ?this_faulty_state                                      a lsa:FaultyState;
                                                          lsa:hasTimespan ?this_state_timespan.

  ?this_state_timespan                                    a lsa:Timespan;
                                                          lsa:beginningIsQualifiedBy ?now.
}

WHERE {

  BIND(UUID() AS ?this_state_timespan).
  BIND(UUID() AS ?this_faulty_state).
  BIND(now() AS ?now).
}
```

Figure 10. SPARQL Actionable of the software procedure implemented by a Fault Detector.

Physical sensors and actuators to not dialogue directly with the semantic engine but through an intermediary: this is a bridge component implemented using Freedomotic[5]. The use of this bridge provides two main advantages. First: Freedomotic includes a large set of "devices plugins" able to dialog with several IoT devices using various (sometimes proprietary) protocols and exposes a REST API to interact with all these plugins. This essentially provides a REST adaptor to all sensors and actuators, allowing the engine to use a uniform technology for all devices. The second advantage of Freedomotic is that it also supports a virtual environment in which it is possible to simulate the movement of persons in a topographic space composed by areas and rooms, populated with simulated sensors and *things* (usually actuatable items); simulated sensors and things can be implemented within this virtual environment for simulation purposes.

To describe the Actionable Behavior we adopted a simple control structure based on a sequence of two Actionable items of different type: SPARQL and REST actionable. SPARQL actions can retrieve data from the tripestore with SELECT queries and produce data with CONSTRUCT statements (whose results are transactionally added to the triplestore). REST actions simply specify the endpoint, the method and the payload for performing HTTP invocations (we assume APIs using JSON as content type). A simple mechanism based on a shared datamap is used to carry data from the output of one action to the input of the subsequent one. This map is populated with data produced by SPARQL SELECT actions and with JSON properties produced by the return messages of REST actions. The values in the map can be used by SPARQL actions in the form of pre-initialized variables and by REST actions as variable elements in JSON payload templates.

To exemplify the use of the shared datamap we refer to the "advanced" apparent temperature example exposed in Sect. "Reference Architecture": a logical sensor produces observations pertaining the apparent temperature whenever physical temperature or humidity sensors produce new observations; the algorithm used to calculate the apparent temperature is implemented by an external service. The Actionable Behavior of the logical sensor can be described as a sequence of three actions: a SPARQL SELECT action retrieving the latest values for humidity and temperature; a REST action invoking the external apparent temperature service by passing it the retrieved humidity and temperature values; a SPARQL CONSTRUCT query to create the RDF graph representing the new observation populated with the value returned by the external service. To share data using the datamap these three actions can cooperate in this way: the values produced by the initial SPARQL query (say `humidity` and `temperature`) are automatically stored in the map under their respective names. The REST action specifies a JSON request message template using `${humidity}` and `${temperature}` placeholders that are replaced with the values of the corresponding elements in the map before the actual invocation takes place. The REST return message is a JSON document with the property `AppTemp` set to the calculated value; this value is automatically stored in the map under its name. The SPARQL CONSTRUCT can create the observation referring to the `AppTemp` variable that is pre-set with the value returned by the external REST API.

While this Actionable Behavior has limited expressive power, it turned out to be sufficient for all the needs related to our case study and is probably sufficient for most real world applications. When this is not the case, as previously discussed, more advanced notations can be adopted.

In the prototype the binding mechanism to materialize semantic actuations into invocations of remote actuators endpoints has not been implemented: we assume that it is the duty of the Actionable Behavior of the logical actuator to define a REST action invoking the actuator (or the bridge) endpoint.

Testing the prototype with Freedomotic

As previously explained our prototype makes use of Freedomotic, an IoT framework that supports various standard and proprietary protocols to interact with a large array of sensors and actuators. The main role of Freedomotic is that of providing a REST bridge to several IoT protocols. But an interesting feature of this framework is that it also supports simulations. In Freedomotic, we created a virtual environment representing a museum composed by a central hall surrounded by four rooms, each of which contains a media player connected to a display. A Freedomotic plug-in has been developed simulating the roaming of a group of people across the rooms of the museum. We also developed plug-ins to simulate presence sensors for the rooms and infrared sensors to be placed on the inside and on the outside of each door: when a person enters the sensing zone, the virtual sensor invokes the engine API to produce an observation. These observations can trigger the cascading activation of logical sensors and logical actuators previously

[5] http://freedomotic.com/

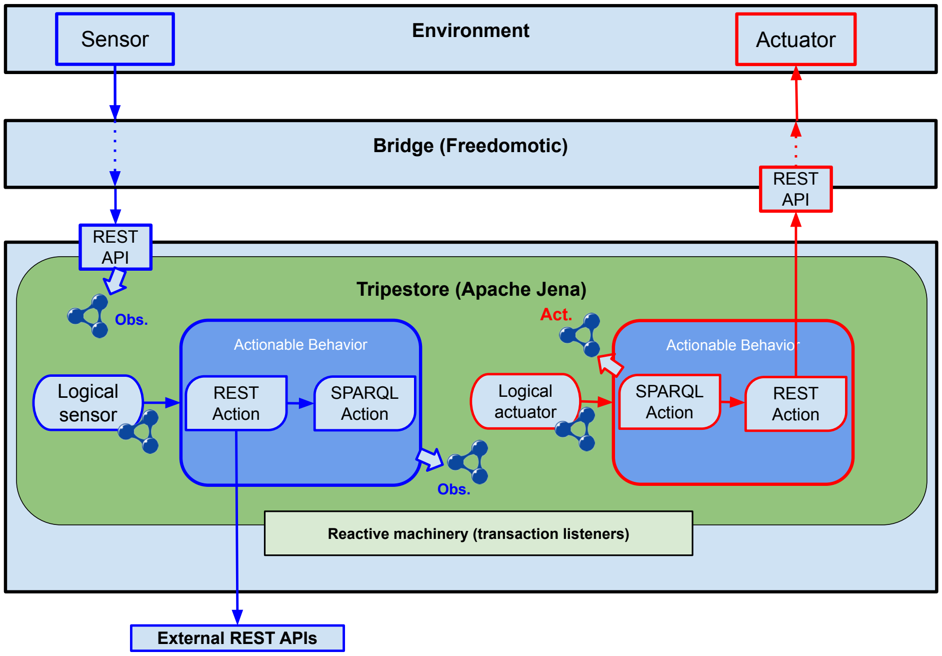

Figure 11. Prototype architecture.

described, with the actuators behavior set to invoke a REST endpoint to turn on or off a media player. This is implemented by directly invoking Freedomotic's APIs. The resulting animated simulation (see Fig. 12) shows the media players turning on when a person enters a room that was empty and turning off when the last person leaves a room.

An interesting aspect of this implementation is that is possible to bind the virtual sensors and actuators with physical ones using the various Freedomotic gateways in order to turn the simulation into a running system acting on a real environment with minimal effort.

CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a reference architecture for context-aware reactive systems aligned with a core ontology able to model logical sensors and actuators, and their behaviors. The ontology is mainly an extension of SSN. However, differently from SSN, we have introduced the concept of SoftwareProcedure to specify the actionable behavior of sensors and actuators that live only in the knowledge base (and consequently have not a direct link with physical devices). Moreover, we have enriched the ontology with the concept of `State` and in particular `BindingState` to address the double nature of device representation: *descriptive* and *executable*. Sensors or actuators descriptions that are not directly or indirectly bound to physical devices are used only for inventory purposes. Otherwise, devices are active and able to process events.

We have discussed and validated the proposed ontology and the supporting architecture with the help of a case study in the domain of smart buildings for cultural heritage. The case study was used also for illustrating the potential of the proposed approach for reconfiguring the system to react to the fault of some physical device. The case study has motivated also a first instantiation of the architecture implemented using Jena, SPARQL and RESTful APIs for the interaction with the external environment, mediated by Freedomotic that also provides simulation support.

The proposed core ontology and the related architecture represent the first step towards the definition of a more complex platform for developing context-aware applications. However, the achievement of this goal requires to address further aspects that we plan to tackle in the near future.

Performance: the current implementation of the proposed architecture has not been optimized for performance; however, triplestores have still not reached the optimization level of more consolidated data storage solutions and this could limit the adoption of the approach for time-critical applications. Nevertheless, we think that triplestores performances will improve also to take into account the diffusion

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27702v1 | CC BY 4.0 Open Access | rec: 4 May 2019, publ: 4 May 2019

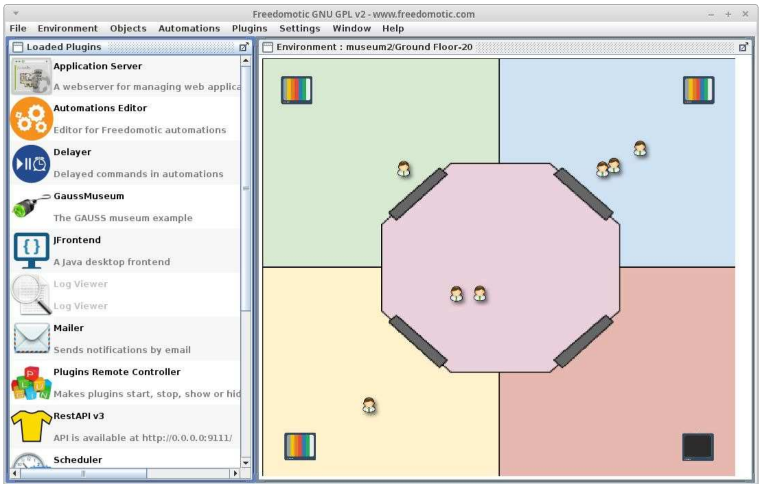

Figure 12. Simulation with Freedomotic.

of languages and approaches to deal with large flows of RDF data, as is the case for stream reasoning (Della Valle et al., 2009).

Data size: as with all storage-based architectures, care has to be taken when the amount of data increase. Most of the entities stored in the knowledge base are temporal data which means that mechanisms to clean up “old” entries can be put in place to limit the size of the “live” data. Old data can either be removed or moved to other storage solutions for offline processing.

Scale and distribution: our solution as described in the paper appears centralized and based on a monolithic data store. While this is obviously the most straightforward way to instantiate our architecture we really designed it so that it can be used to create nodes of distributed hierarchical systems: single instances acting as edge nodes (as the one proposed in the case study of this paper) and operating on local runtime models can cooperate with higher level components by passing them only the (potentially pre-processed) information they need.

Programming support: like other RDF-based approaches, we experimented with a high verbosity when implementing our prototype that can make complex and hard to follow relatively simple mechanisms. We are currently investigating options to ease these issues by adopting visual support tools and re-usable component libraries.

Adaptation policies: It may not be simple to guarantee that the modified system meets the requirements it was designed for and also guarantee then it exhibits a stable behavior, avoiding a continuous cascade of modifications trying to correct new issues introduced by previous modifications. Guaranteeing the stability of feedback-loop controlled self-modifying systems is outside the scope of this work but it should be taken into consideration for a proper design of adaptive systems.

ACKNOWLEDGMENTS

This paper was supported by MIUR PRIN 2015 GAUSS Project and MIUR PON VASARI Project.

REFERENCES

Abowd, G. D., Dey, A. K., Brown, P. J., Davies, N., Smith, M., and Steggles, P. (1999). Towards a better understanding of context and context-awareness. In *International symposium on handheld and ubiquitous computing*, pages 304–307. Springer.

Angelopoulos, K., Souza, V. E. S., and Mylopoulos, J. (2015). Capturing variability in adaptation spaces: A three-peaks approach. In *International Conference on Conceptual Modeling*, pages 384–398. Springer.

Baresi, L. and Sadeghi, M. (2018). Fine-grained context-aware access control for smart devices. In *2018 8th International Conference on Computer Science and Information Technology (CSIT)*, pages 55–61. IEEE.

Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., and Riboni, D. (2010). A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161–180.

Blair, G., Bencomo, N., and France, R. B. (2009). Models@run.time. *Computer*, 42(10):22–27.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236.

Chen, H., Perich, F., Finin, T., and Joshi, A. (2004). Soupa: Standard ontology for ubiquitous and pervasive applications. In *The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004.*, pages 258–267. IEEE.

Cheng, B., De Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., and Whittle, J. (2009). Software engineering for self-adaptive systems: A research roadmap. *Lecture Notes in Computer Science*, 5525 LNCS:1–26.

Dautov, R., Paraskakis, I., and Stannett, M. (2014). Utilising stream reasoning techniques to underpin an autonomous framework for cloud application platforms. *Journal of Cloud Computing*, 3(1):13.

De Lemos, R., Giese, H., Müller, H. A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N. M., Vogel, T., et al. (2013). Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer.

Delic, K. A. (2016). On Resilience of IoT Systems: The Internet of Things (Ubiquity symposium). *Ubiquity*, 2016(February):1:1–1:7.

Della Valle, E., Ceri, S., Van Harmelen, F., and Fensel, D. (2009). It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89.

Falco, R., Gangemi, A., Peroni, S., Shotton, D., and Vitali, F. (2014). Modelling OWL ontologies with Graffoo. In *European Semantic Web Conference*, pages 320–325. Springer.

Frank, A. U. (2001). Tiers of ontology and consistency constraints in geographical information systems. *International Journal of Geographical Information Science*, 15(7):667–678.

Furno, A. and Zimeo, E. (2014). Context-aware composition of semantic web services. *Mobile Networks and Applications*, 19(2):235–248.

Giallonardo, E., Sorrentino, C., and Zimeo, E. (2017). Querying a complex web-based KB for cultural heritage preservation. In *Knowledge Engineering and Applications (ICKEA), 2017 2nd International Conference on*, pages 183–188. IEEE.

Haller, A., Janowicz, K., Cox, S., Le Phuoc, D., Taylor, K., and Lefrançois, M. (2017). Semantic sensor network ontology. *W3C Recommendation, W3C*.

Hölzl, M. and Gabor, T. (2015). Reasoning and learning for awareness and adaptation. In *Software Engineering for Collective Autonomic Systems*, pages 249–290. Springer.

Janowicz, K., Haller, A., Cox, S. J., Le Phuoc, D., and Lefrançois, M. (2018). SOSA: A lightweight ontology for sensors, observations, samples, and actuators. *Journal of Web Semantics*.

Kaebisch, S. and Kamiya, T. (2017). Web of things (WoT) thing description. *First Public Working Draft, W3C*.

Lefrançois, M. (2017). Planned ETSI SAREF Extensions based on the W3C&OGC SOSA/SSN-compatible SEAS Ontology Paaerns. In *Workshop on Semantic Interoperability and Standardization in the IoT, SIS-IoT*, page 11p.

Miles, A. and Bechhofer, S. (2009). SKOS simple knowledge organization system reference. *W3C recommendation*, 18:W3C.

Morandini, M., Penserini, L., Perini, A., and Marchetto, A. (2017). Engineering requirements for adaptive systems. *Requirements Engineering*, 22(1):77–103.

Morin, B., Barais, O., Jezequel, J., Fleurey, F., and Solberg, A. (2009). Models@ Run.time to Support Dynamic Adaptation. *Computer*, 42(10):44–51.

Pederson, T., Ardito, C., Bottoni, P., and Costabile, M. F. (2008). A general-purpose context modeling architecture for adaptive mobile services. In *International Conference on Conceptual Modeling*, pages 208–217. Springer.

Perera, C., Zaslavsky, A., Christen, P., and Georgakopoulos, D. (2014). Context aware computing for the internet of things: A survey. *IEEE communications surveys & tutorials*, 16(1):414–454.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27702v1 | CC BY 4.0 Open Access | rec: 4 May 2019, publ: 4 May 2019

Poggi, F., Rossi, D., Ciancarini, P., and Bompani, L. (2016). Semantic run-time models for self-adaptive systems: a case study. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2016 IEEE 25th International Conference on*, pages 50–55. IEEE.

Pötter, H. B. and Sztajnberg, A. (2016). Adapting heterogeneous devices into an iot context-aware infrastructure. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2016 IEEE/ACM 11th International Symposium on*, pages 64–74. IEEE.

Rossi, D., Poggi, F., and Ciancarini, P. (2018). Dynamic high-level requirements in self-adaptive systems. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 128–137. ACM.

Sagar, S., Lefrançois, M., Rebaï, I., Khemaja, M., Garlatti, S., Feki, J., and Médini, L. (2018). Modeling Smart Sensors on top of SOSA/SSN and WoT TD with the Semantic Smart Sensor Network (S3N) modular Ontology. In *9th International Semantic Sensor Networks Workshop*.

Shaw, A. C. (1978). Software descriptions with flow expressions. *IEEE Transactions on Software Engineering*, (3):242–254.

Szvetits, M. and Zdun, U. (2016). Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Software & Systems Modeling*, 15(1):31–69.

Tamura, G., Villegas, N. M., Muller, H. A., Duchien, L., and Seinturier, L. (2013). Improving context-awareness in self-adaptation using the DYNAMICO reference model. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2013 ICSE Workshop on*, pages 153–162. IEEE.

Vogel, T. and Giese, H. (2014). Model-driven engineering of self-adaptive software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 8(4):18.

Whittle, J., Sawyer, P., Bencomo, N., Cheng, B. H., and Bruel, J.-M. (2009). Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *2009 17th IEEE International Requirements Engineering Conference*, pages 79–88. IEEE.