Semantics-Driven Programming of Self-Adaptive Reactive Systems

Semantics-Driven Programming of Self-Adaptive Reactive Systems

Ester Giallonardo[*,§], Francesco Poggi[†,¶], Davide Rossi[‡,‖] and Eugenio Zimeo[*,**]

[*]*Department of Engineering, University of Sannio*

[†]*Department of Communication and Economics*
University of Modena and Reggio Emilia

[‡]*Department of Computer Science and Engineering*
University of Bologna
[§]*egiallonardo@unisannio.it*
[¶]*francesco.poggi@unimore.it*
[‖]*daviderossi@unibo.it*
[**]*eugenio.zimeo@unisannio.it*

In recent years, new classes of highly dynamic, complex systems are gaining momentum. These classes include, but are not limited to IoT, smart cities, cyber-physical systems and sensor networks. 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. A desirable design feature of these systems is the ability of adapting their behavior to environment changes. In this paper, we propose an approach to support adaptive, reactive systems based on semantic runtime representations of their context, enabling the selection of equivalent behaviors, i.e. behaviors that have the same effect on the environment. The context representation and the related knowledge are managed by an engine designed according to a reference architecture and programmable through a declarative definition of sensors and actuators. The knowledge base of sensors and actuators (hosted by an RDF triplestore) is bound to the real world by grounding semantic elements to physical devices via REST APIs. The proposed architecture along with the defined ontology tries 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.

Keywords: Context modeling; context-awareness; semantic modeling; semantic sensor networks; ontologies; models@runtime; reactive systems; self-adaptive systems.

1. Introduction

Internet of Things (IoT), smart cities and cyber-physical systems propose several scenarios characterized by a high level of dynamism and heterogeneity. These systems need to make reactive control decisions responding to changes in their environment and in themselves. Systems that perform actions based on input changes are commonly referred to as reactive. As stated in [1], reactive systems "repeatedly prompted by the outside world and ... continuously respond to external inputs". We assume that reactive systems are able to continuously react to their environment at a speed determined by the environment [2], which entails the ability to express high levels of responsiveness but also maintainability and extensibility. For their nature, these applications must be context-aware, where context is the state that a system is able to access to or modify, possibly shared with other systems and exposed to devices or applications other than the one the state is referred to [3].

In spite of the proliferation of frameworks and tools for programming IoT systems, design and implementation of reactive, context-aware systems pose additional challenges: (a) to infer high-level context properties from directly measurable ones to properly handle context changes, (b) to ensure system resilience to better support reactiveness, especially when the number of connected devices increases. Moreover, as for IoT systems, it is very important to manage the heterogeneity of the physical devices they consist of.

The following example illustrates a possible scenario that could benefit of the adoption of novel paradigms and runtime supports for addressing the problems introduced before. Let us suppose that in a museum a new temporary exhibition is arranged. The museum is monitored by several anti-theft, infrared sensors previously installed and managed by a dedicated software engine. In a room of this exhibition, a multimedia content has to be played. The organizers of the exhibition express the desire that the content starts playing when visitors enter the room, and stops when the room is empty.

This behavior can be implemented by using an actuator that turns on the playback when a person is detected in a formerly empty room, and turns it off then the last person leaves the room. A sensor reporting observations pertaining the presence of people in the room is used to trigger the multimedia actuator. The museum administrators equip each room with thermal camera sensors to track and count people moving in the room, and to update the managing software engine. Moreover, they realize that presence can also be detected by combining the anti-theft infrared sensors at the doors of the rooms, even if the related precision is lower.

The engine should support administrators in easily modeling the environment and the related detecting devices and should be able to reconfigure the model for resilience when, for example, a thermal camera sensor is no longer working, by replacing it with another "equivalent" logical sensor. The logical sensor should perform the same task (i.e. observe the presence in a room) by composing the observations

produced by other sensors, for example, the ones produced by a couple of anti-theft infrared sensors at the sides of the room door.

To satisfy these requirements, we propose an ontology and an engine for supporting semantics-driven programming of self-adaptive reactive systems, enabling on the fly declarations to satisfy evolving needs. The ontology is used to describe systems and their environments, while the engine provides the runtime support for the application logic that drives state changes. The ontology is based on the Semantic Sensor Network (SSN) ontology, a recent W3C recommendation [4] that has been designed to describe systems composed of a densely interconnected graph of sensors and actuators along with the observations and the actuations they produce.

The semantic model is supported by a software architecture centered on a knowledge base which is bound to real world entities by grounding (mainly via web services) semantic elements to physical 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 structure of the model.

The overall architecture and the proposed ontology have been tested by implementing a prototype based on Apache Jena, OWL, and SPARQL, for the knowledge base, and RESTful services, for the interaction with the physical world, currently virtualized through an emulator. We show that the proposed approach enhances systems adaptability and maintainability. The system is able to discover alternative configurations when needed (e.g. in case of faults); moreover, its maintenance is simplified by a declarative, query-based approach that allows developers to easily discover and manage devices through SPARQL queries.

The remainder of this paper is organized as follows. Section 2 presents the related work from both research and standardization points of view. Section 3 sketches our proposed architecture for context-aware reactive applications. Section 4 introduces the SSN ontology, with the aim of identifying its limitations with reference to modeling runnable sensors/actuators behaviors, and proposes an extension called LSA. Section 5 explains the proposed approach and the ontology extension by modeling and developing a system for smart buildings in eCulture domain. Section 6 shows the results of a performance evaluation that exhibits reasonable response times for a wide class of reactive systems. Section 7 discusses the pros and cons of the proposed approach. Finally, Section 8 concludes the paper and highlights future work.

2. Related Work

Reactive systems engage in stimulus-response behavior in order to produce desirable effects in their environment [5]. They should meet the demands for responsiveness, maintenance and extensibility [6], supporting the continuous changing of systems designs. In particular, for a system to be responsive, we assume that its responses to

external stimuli must take place in negligible time with respect to the response delays of its environment.

Data-flow composition is a common paradigm adopted to program reactive systems. An example of this approach is Node-RED,[a] a framework that enables static visual wiring of services, smart objects and custom nodes. However, this framework does not support the definition of resilient models able to adapt themselves in order to satisfy desirable properties.

A more flexible approach based on a queryable model introduces significant advantages in terms of resilience due to the ability of selecting alternative execution paths after each micro-operation. This flexibility is paid in terms of higher response times that in many application classes does not violate the responsiveness requirement. Moreover, if queryable models are enhanced with semantics, the recall of each query improves, by providing additional alternatives to satisfy system properties. Semantics helps also reducing the effort needed to maintain the system.

2.1. *Models@runtime in IoT architectures*

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. Szvetits *et al.* [7] 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. We restrict our analysis to models@runtime for IoT architectures.

ContQuest [8] is an approach to dynamically integrate devices into a context-aware IoT environment. The proposed IoT architecture is not self-adaptive, since the integration of new devices must be programmed through specific resource drivers; moreover only sensors can be programmed to respond to events, while actuators need requests to act. A reflexive model is exploited to discover available resources.

DYNAMICO [9] introduces an infrastructure for self-adaptive systems with context-awareness requirements. Monitoring strategies can be specified at runtime through the COb (Control Objectives) model, determining the planning of the adaptation to perform. However, the overall behavior is not purely reactive, i.e. it does not only exploit sensed knowledge available in memory but it also generates strategies proactively.

In [10], the authors use a graph model for defining a smart home context and capabilities. They apply the MAPE-K feedback loop for increasing workflow resilience according to various context factors, e.g. battery levels, local engine state and sub-processes states. The resilience is ensured by defining compensations actions for the required goals. Compensations refer to faulty executions that require new plannings. The approach does not work on open component systems, but only on business processes executable by a workflow engine.

[a] https://nodered.org/docs/.

The works discussed above do not provide access to the maintenance of IoT systems behaviors at model level. ContQuest [8] exploits models@runtime only for discovering, while Dynamico [9, 10] for discovering and monitoring. In ContQuest [8], system behavior changes require programming of resource drivers, while Dynamico [9, 10] assume highly available resources. Related efforts based on semantics and models@runtime for MAPE-K architectures can be found in [11–14].

In this paper, we propose an approach to context-based self-composition, self-healing and meta-adaptation according to behaviors defined at model level of open component systems. Some recent work proposes approaches for context-aware systems based on runtime models able of supporting behavior definition. Angelopoulos *et al.* in [15] 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* [16] 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).

Another notable approach is RELAX [17], a declarative requirements language for self-adaptive systems supporting the explicit expression of 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.

2.2. *Semantic models*

Most of the papers introduced before recognize the need for a runtime 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 variability alternatives 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 [18, 19] 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 [20] 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.

Several papers have tried to propose approaches and technologies to easily model and handle dynamic context-aware applications especially for ubiquitous and pervasive computing. One of the first ontology-based approaches is SOUPA [21]. It is expressed in OWL and includes modular component vocabularies to represent intelligent agents, time, space, events, user profiles, actions, and policies for security and privacy. However, it does not focus on sensors/actuators and reactive systems but on smart meeting places. In [19], the authors discuss the requirements that context modeling and reasoning should meet, including the modeling of a variety of context information types and their relationships, of high-level context abstractions describing real world situations, and of uncertainty of context information, without defining an ontology.

Paper [22] surveys context awareness from an IoT perspective. IoT researchers are taking into consideration Web technologies (WoTs) to support context-driven system engineering. The goal of the WoT is to extend Web services to devices, allowing a Web client to access devices properties, to request the execution of actions or to subscribe to events representing state changes [23]. The related ontology describes how to model sensors and actuators with the main objective of easing the binding with devices reachable through web protocols (REST, CoAP, etc.).

A different objective is pursued by the SSN ontology [4], an Open Geospatial Consortium (OGC)/World Wide Web Consortium (W3C) standard. It is mainly focused on the Sensor, Observation, Sample, Actuator (SOSA) pattern [24] to model reactive systems. It aims at supporting the definition of simple reactive behaviors that link observations, coming from modeled sensors, with the related reactions, performed by actuators. 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 [25] is an effort that tries to specialize SSN for supporting 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. 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 select algorithms from the current context and to change the state of the whole smart sensor. Therefore, the main purpose of S3N is to support smart sensors modeling and not to close the logical gap between sensors and actuators for fully programming reactive systems.

3. An Architecture for Semantic Context-Aware Reactive Systems

Based on recent W3C standards, we propose an architecture and a modeling approach for defining context-aware reactive systems as graph-based resilient compositions of semantically equivalent logical devices.

3.1. *Architecture*

The main component of the architecture is the Semantic Engine (see Fig. 1). It extends a knowledge base with the machinery needed to interact with sensors and actuators and execute their software procedures. The knowledge base contains a model of the physical world it interacts with, which is enriched and modified with the data coming from the sensors, assuring consistency with the physical elements it represents. This alignment is usually referred to as *causal connection*. When a modification of the model causes the enactment of an actuator to materialize this modification in the physical world we say that the model is *bi-causally connected* [26], a feature that is supported by our architecture.

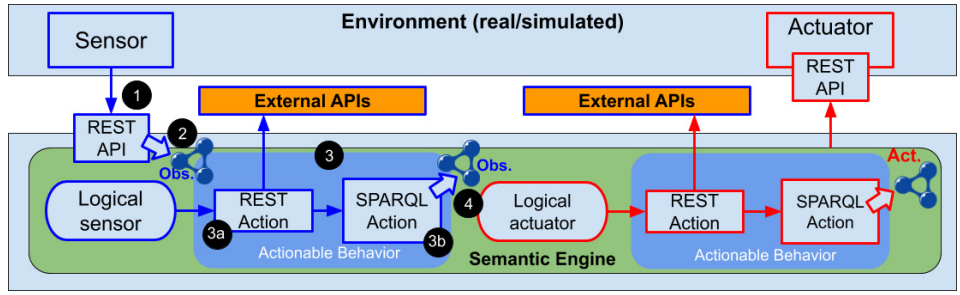

Fig. 1. Architecture outline.

To exemplify these concepts just think about a simple reactive system immersed in an environment composed by a room with a light bulb, a bulb actuator and a light sensor. 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, the modification of the state of a 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.

The key ingredients to actualize a system of this type are: one or more models that describe real-world conceptual classes, a binding mechanism that maps sensor observations to knowledge base updates, logical causal connections that propagate updates throughout the knowledge base, and a binding mechanism that maps updates to actuators activation for preserving the model alignment with real-world situations. It is worth noting that causal connections need some kind of

computational support. According to the organization above, our architecture presents the following:

(i) a semantic model built using the ontology presented in Sec. 4.2, hosted by a knowledge base platform (i.e. a triplestore in our case);
(ii) a linking mechanism to report sensor readings to the system, implemented using web services exposed by the system, which is responsible for converting readings into appropriate system modifications (i.e. new statements to insert into the knowledge base);
(iii) a programmed logic that changes the semantic model generating new facts (i.e. observations and actuations statements) triggered by observations;
(iv) an actuation mechanism exploiting Web services exposed by the system actuators, in a way that is consistent with the WoT approach.

Causal connections are supported by rules that correlate real-world changes observed by sensors with knowledge base updates. We consistently represent these rules in the knowledge base itself: the *activation part* is modeled as *software procedures* 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*.

The engine connects to the physical world by exposing a service API used to receive observations from external sensors (see Fig. 1 on the left side) that can be real or simulated ones, and by invoking web service endpoints for activating external actuators (see Fig. 1 on the right side).

Whenever an external sensor notifies an observation invoking the engine's API (represented by the black circle with a "1" inside in Fig. 1), that observation is transformed in a semantic format and added to the knowledge base (Fig. 2). If a logical sensor/actuator is interested in that observation (which means that it is modeled in such a way that its software procedure uses as one of its inputs the property reported by the observation), then its related software procedure is executed (Fig. 3) by running the actions (Figs. 3(a) and 3(b)) that define the specific Actionable Behavior, producing new facts (observations or actuations, Fig. 4). The actionable behavior can be composed of various *Action* elements, such as SPARQL queries, Java code and external services (depicted in orange) to increment the capabilities of the logical sensors and actuators (LSA). The two actions depicted in the figure are just a possible example. If new actuations are produced, they trigger external actuators through REST API invocations.

3.2. *Prototype*

We realized a working prototype,[b] that we call *semantic engine*, on the basis of the architecture presented above. The triplestore hosting the knowledge base is

[b] The prototype is available at https://github.com/cars-team/semanticengine.

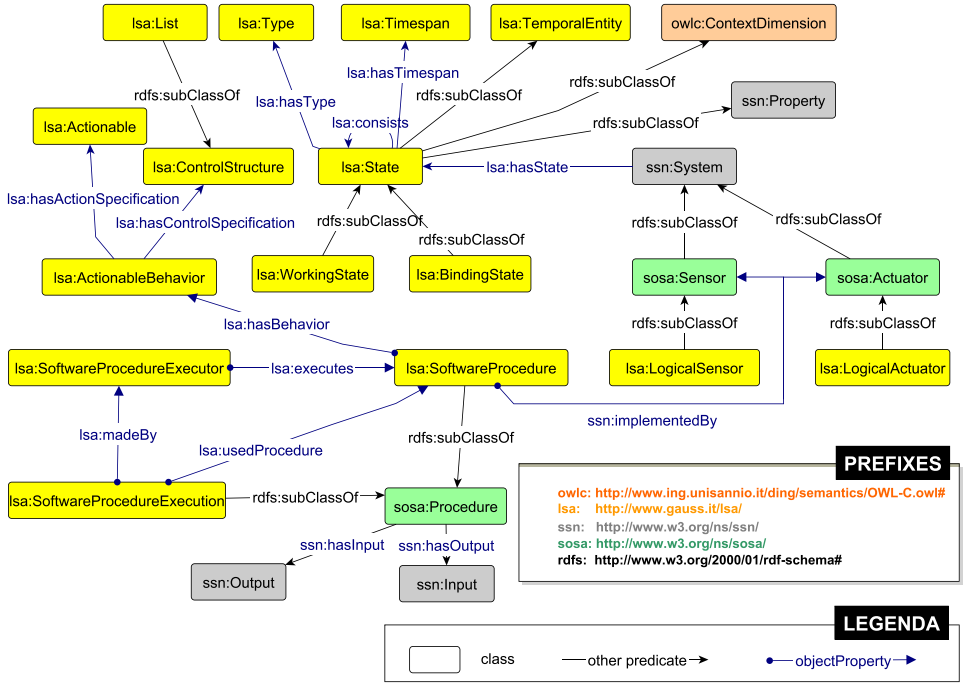

Fig. 2. (Color online) Core classes of the LSA ontology. Classes belonging to different ontologies are identified by different background colors: yellow for LSA, gray for SSN and green for SOSA (i.e. the SSN core module).

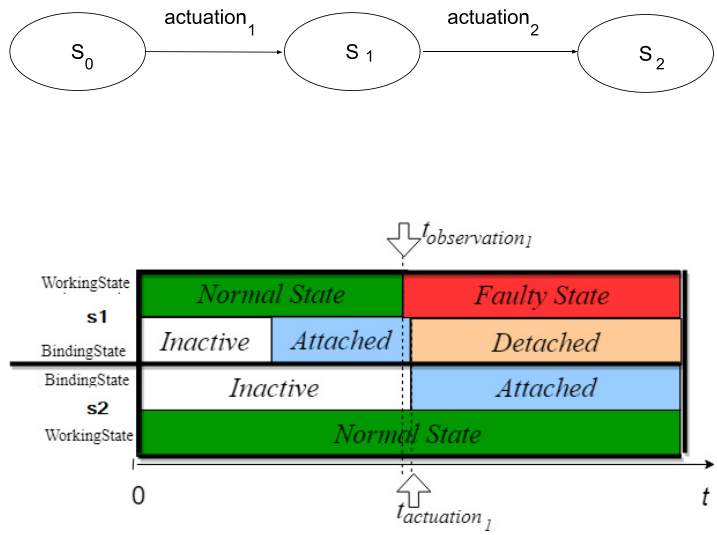

Fig. 4. State transitions of two sensors ($s1$ and $s2$) during the reactive reconfiguration.

Apache Jena.[c] The external contact points of the engine are a REST API that is mainly used to submit observations and a SPARQL endpoint implemented using Apache Jena Fuseki. We assume that all sensors connect via the REST API so, if they cannot directly do that, wrappers (or polling components) have to be provided. We successfully used Freedomotic[d] as a gateway and emulator for a large number of existing IoT devices supporting various network protocols. To implement context detection, we augmented Jena with a transaction monitor that intercepts all write transactions performed on the triplestore.

The activation mechanism works as follows: whenever a new observation (or actuation) related to an observable property, declared as an input for a procedure associated with a logical sensor (or logical actuator), appears in the knowledge base, this fires the execution of the related logical element. This execution can be carried out by different executors. Current implemented executors are: (1) local Java code (a) JAR file containing the code to be executed can be posted via the REST API); (2) SPARQL queries (this includes SPARQL update queries so creation/modification of elements is possible); (3) invocation of external REST API (that can access the knowledge base via the SPARQL endpoint whose URL can be passed to the REST API). At the present time only REST services encoding messages with JSON are supported.

4. Semantic Modeling of Context and Behavior

In this section, we illustrate the SSN ontology and describe our extension of SSN to support modeling and execution of reactive systems.

4.1. *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 concerns 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 the main concepts and properties of WoT systems through schema.org annotations.

The SSN main perspective is the system one. "System" is indeed the main ontological concept of the SSN ontology: it is a unit of abstraction for pieces of infrastructure that implement procedures for performing observations on the state of the world, or actuations to make a change to the state of the world. Systems can be decomposed into their constituent subsystems and deployed on specific platforms (as vehicles, aircrafts, etc.) for particular purposes. Sensors are triggered by stimuli that originate observations, i.e. events that assign results to observable properties.

[c] https://jena.apache.org/.
[d] https://www.freedomotic-iot.com.

Stimuli can be proxies for observations of properties (i.e. observable qualities, e.g. the temperature) related to features of interest (e.g. a room). 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 related to a feature of interest, for example a room. Actuators determine changes to the state of the world through the execution of procedures triggered by the observations of properties.

SSN *only allows to describe* the information that is provided to a procedure for its use (`ssn:Input`), and the information that is reported by a procedure (`ssn:Output`), but it *does not provide a mechanism to support the execution* of such procedures. In order to implement self-adaptive reactive systems (which is the topic of this paper), it is necessary to provide a mechanism to describe the logic governing these procedures, that is how procedure inputs (e.g. observations by sensors) are combined to produce procedure outputs (e.g. a change in the state of a system element performed by an actuator). In other words, *SSN does not provide a way to define machine actionable system behaviors.*

To overcome this limitation, we extended the SSN concept of procedure by introducing the concept of software procedure, which we mainly exploit with LSA, as described in Sec. 4.2.

4.2. *Logical sensors and actuators ontology*

Figure 2 shows a Graffoo [27] diagram of the core elements of the LSA ontology,[e] our extension of the SSN ontology that allows describing LSA by specifying their interactions, behaviors, and state evolution.

The LSA ontology introduces two main concepts: (software) *logical sensors* and *logical actuators*. A logical sensor is a sensor that generates observations by executing a software procedure that uses as inputs the observations produced by other (one or more) sensors. A logical actuator is an actuator that generates actuations by executing a software procedure that uses as inputs actuations or observations produced by (one or more) other actuators or sensors. LSA live only in the virtual space (e.g. a knowledge base), and may be connected to the external world through SSN simple sensors and actuators, as shown in Sec. 5.

LSA are modeled through the `lsa:LogicalSensor` and `lsa:LogicalActuator` classes, which are subclasses of `sosa:Sensor` and `sosa:Actuator`, respectively. The behaviors associated to logical sensors/actuators are represented by the `lsa:SoftwareProcedure` class, and the property `ssn:implementedBy` is used to connect software procedures to sensors/actuators and consequently to the logical ones.

A `sosa:Procedure` is defined in SSN as "a workflow, protocol, plan, algorithm, or computational method specifying how sensors make observations, or actuators make

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

changes to the state of the world". A `lsa:SoftwareProcedure` is a specific kind of `sosa:Procedure` with an actionable behavior (i.e. a behavior that can be operationalized — i.e. executed — by a software agent executed by the semantic engine or by external engines). Software Procedures are mainly implemented by LSA, that produce new observations or actuations as result of the software procedure execution. The behavior associated to a software procedures can be implemented, for instance, by executable code, which is connected to the procedure by the `lsa:hasBehavior` property.

It is important to note that the LSA ontology does not impose constraints on how such behaviors should be represented. For example, they can be modeled as OWL-S [28] processes, BPMN processes described using the BPMN Ontology [29], etc. The LSA ontology differentiates procedures as follows:

- **Procedures specifications:** The algorithm, workflow, protocol, etc. used by a sensor (actuator) to perform observations (actuations), along with a declaration of its inputs and outputs. e.g. the algorithm used by a logical sensor that measures the perceived humidity in a room (output) by aggregating a temperature observation and a humidity observation (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 [30]), a procedure execution is modeled with the `lsa:SoftwareProcedureExecution` class. It is related (via the `lsa:usedProcedure` property) to a procedure specification (represented by the `lsa:SoftwareProcedure` class) and is performed (`lsa:madeBy` property) by a `lsa:SoftwareProcedureExecutor`. The executor is a software agent able to execute (`lsa:executes`) the `lsa:SoftwareProcedure`. The specifications of the procedure (i.e. the algorithm, workflow, protocol, etc. to be used) are represented by the `lsa:ActionableBehavior` class, connected to the procedure by the `lsa:hasBehavior` property. The behavior consists of a sequence of tasks (`lsa:Actionable`) organized in a `lsa:ControlStructure`, as described in detail in the following of this section.

Another key point of the LSA ontology is that it allows to represent systems continuously through their behaviors, interactions and states, enabling reactive control and coordination features. Data flows can be specified through the `ssn:hasInput` and `ssn:hasOutput` properties, enabling continuous systems redesign. The model supports data flows by allowing software procedures execution as a reaction to facts asserted in the knowledge base. In this way, the LSA ontology provides a mechanism for creating dynamic chains of procedures referred to different kinds of devices (sensors, actuators, logical sensors, logical actuators). According to this model, a serie of procedures can be executed progressively with a runtime

semantics-based planning. For instance, a logical sensor LS1 may be activated by two new observations, which are produced by physical sensors, and are used as inputs for LS1's software procedure. This procedure is executed by the engine, and uses these observations to produce a new observation as output. The new observation may be in turn one of the inputs of another software procedure defined for logical actuator LA1. The engine will then executed LA1's software procedure, and possibly produce an actuation as output.

Software procedures allow to specify the behaviors of LSA. Such behaviors can be represented from both a structural and an operational perspective. The `lsa:hasControlSpecification` property allows to associate the control flow of the behavior, while the `lsa:hasActionSpecification` allows to define the tasks that have to be executed in order to implement the behavior of logical elements.

According to SSN, each single element (e.g. a sensor or an actuator) of the whole system is itself a system (`ssn:System`). Since each reactive system should be handled according to its state, in the LSA ontology we introduce the concept of `lsa:State` to represent an observable/actuable condition of a system, in a limited contiguous extent in time. The `lsa:hasState` property allows to associate states to systems, while the `lsa:consistsOf` property allows to associate the constituent states. State values are the result of actuations (see Fig. 3).

As shown in Fig. 2, `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 or faulty working); the latter refers to whether physical systems and their representations in the knowledge base are (directly or indirectly) bi-causally connected [26]. In particular, `lsa:WorkingState` can be specialized in `lsa:normalState` and `lsa:faultyState` whereas `lsa:BindingState` can be `lsa:inactive`, `lsa:attached` or `lsa:detached`. In fact, depending on the working state, the virtual representation of a system (sensor or actuator) in the KB can be detached from the external (e.g. physical) counterpart to avoid storing altered observations in the knowledge base. On the other hand, a system is attached when it is directly or indirectly bi-causally connected with a physical device and inactive if only its passive representation is used.

4.3. *Enabling resilience*

The LSA model supports specification of dynamic architectures for a single coherent system, structured as a set of subsystems characterized by objective functions described through `ssn:Property`, and by behavioral aspects defined through `lsa:Behavior`. This runtime model can be exploited for managing the system evolution, and the `lsa:Behavior` is used for implementing the system adaptive logic.

A software architect can use it for defining logical elements (i.e. sensors or actuators) able to ensure resilience. These elements live in a virtual space (i.e. the knowledge base) and govern other system elements by inserting state transitions (e.g. detaching from the system a faulty sensor, and attaching to the system an

equivalent sensor) according to the dynamic evolution of the system state. This way, faulty sensors (or actuators) can be substituted by other sensors (or actuators) that are in normal working conditions that are able to observe (or act on) the same properties.

The element that performs this activity is a logical actuator. It implements a self-healing strategy for coordinating the entry and exit of equivalent system elements, based on the observation of their states and capabilities. In particular, the execution of its procedure performs actuations that produce state transitions able to resolve faults. Figure 4 shows the attaching actuation of sensor s_2 and the detaching actuation of sensor s_1 at $t = t_{\text{actuation}_1}$, as an example of run-time substitution for resilience.

5. Modeling a System

To clarify the overall approach, the ontology and how the engine works, in this section, we model and implement the example (derived from a larger system for Cultural Heritage preservation [31]) discussed in the introduction of this paper.

The proposed approach is model-driven. The first artifact to produce is the model of the system. In our context this means using SSN/LSA entities and relationships to define a graph of the system, which can be initially designed by using a graphical tool and then transformed in RDF. To resolve the heterogeneity of the actionable types, developers should define and share the relative types vocabularies.

During the modeling phase, developers describe the system in terms of properties, sensors, actuators and identify the software procedures and the related actionable items that allow to define devices' behaviors, states and executors.

For each actionable item, the developer must provide specific artifacts that depend on the type of the actionable item: SPARQL operations, Java code, REST interactions, and so on.

The overall system model (composed of the main RDF graph and actionable items artifacts) is then deployed and loaded in the knowledge base. Possible changes to the initial model can be introduced into the system by issuing specific queries at any time.

According to the presented methodology, the initial phase is the one related to the modeling of the system and its environment. Therefore, after a brief description of a dynamic scenario in the context example, in the following, a detailed model is reported and discussed.

5.1. *Multimedia playback scenario and graphical modeling*

In this section, we discuss how a system composed of physical and logical sensors/ actuators can be modeled with the LSA ontology. Following the example

presented in the introduction, we focus on the definition of a model for the following scenario:

(1) a tourist crosses the door of the museum, and the two physical infrared sensors on the door sides produce two observations about the presence of a person in their detection areas;
(2) a logical sensor aggregating such observations produces another observation updating the number of persons present in the rooms;
(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 into the triplestore.

In order to cover the main features introduced by the LSA ontology, we focus on modeling a logical sensor (functionally equivalent to the physical thermal camera sensor used for observing people presence in a room) and a logical actuator. A graphical notation named Graffoo [27] is used in this section to help the reader. A complete example with code excerpts of the model is discussed in the next subsection, which covers both physical and logical elements.

1. Observations made by physical sensors: Figure 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 invokes the engine REST API in sequence (providing their ids and the instants of time when the observations occurred as request parameters).

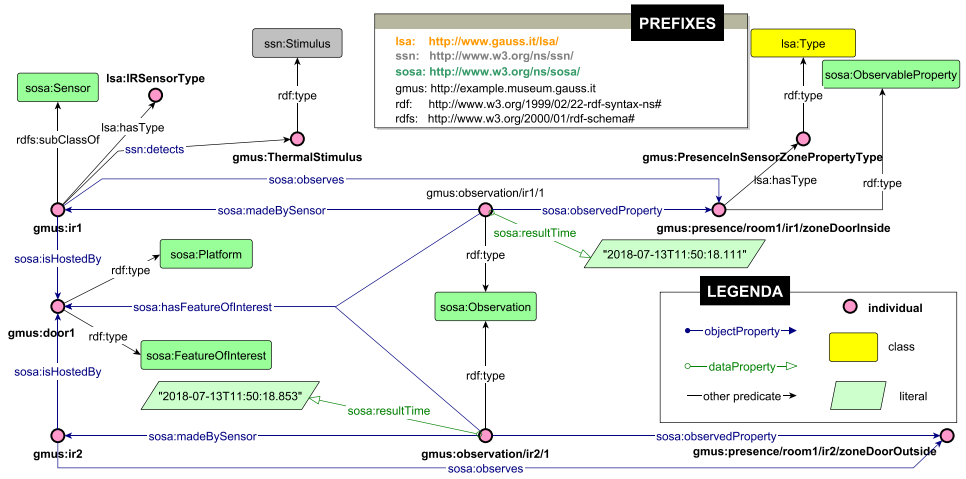

Fig. 5. (Color online) Observations made by two infrared sensors. Classes belonging to different ontologies are identified by different background colors: yellow for LSA, gray for SSN and green for SOSA (i.e. the SSN core module).

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 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.

In these examples, we make use of punning,[f] an OWL metamodeling capability that allows to treat model elements as classes and individual at 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 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. LSA) 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. Since these properties (see Fig. 6) are inputs of the `gmus:entrance/door1/room1` procedure (as specified by `ssn:hasInput`), the semantic engine identifies the procedure, which is tied to the logical sensor `gmus:ls1`, a specific instance of `gmus:infraredPresenceSensor` (the class representing logical presence sensors) hosted by the triplestore (`gmus:triplestore`), and executes it by observing the presence of people in the specific room (`gmus:people/room1`).

A 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 is shared by all logical sensors of a type, the engine identifies the related software procedure (`gmus:DoorRoomEntrance` in our case) and retrieves the behavioral specification (`gmus:doorRoomEntrance/behavior`) by navigating the `lsa:hasBehavior` property.

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

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

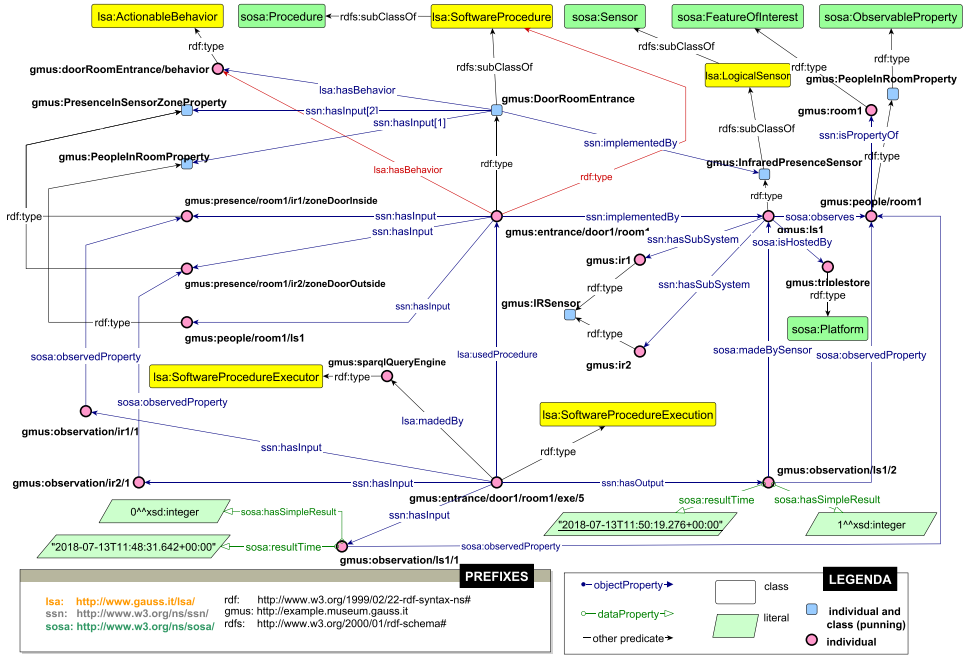

Fig. 6. (Color online) Observations made by the logical presence sensor. Square brackets are used to specify property cardinality restrictions. Classes belonging to different ontologies are identified by different background colors: yellow for LSA, gray for SSN and green for SOSA (i.e. the SSN core module).

(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 pertaining the number of persons in the rooms connected by the door[g]) and the software procedure executor (`gmus:sparqlQueryEngine`) by the `ssn:hasInput` and `lsa:madeBy` property, respectively;
(2) two observations as output of the procedure execution represented using the `ssn:hasOutput` property. For instance, the number of people in the first room has been updated 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

[g]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).

room is activated, and the related software procedures is retrieved and executed, triggering the final actuation (REST invocation) of the physical device that starts the video playback on the monitor.

This example shows a way to dynamically reconfigure the system in an unplanned way with the creation of new logical sensors, that is by only changing the content of the knowledge base where the definitions of the logical sensors are stored.

Resilience can easily be addressed in a system of this kind by implementing a monitoring infrastructure producing reports about the state of physical elements. An health monitor for physical sensors can be implemented as a sensor producing observations describing the working state of these sensors. These observations can trigger a logical actuator playing the role of Reconfigurator. The Reconfigurator can query the knowledge base, looking for alternative inactive sensors (physical or logical ones) producing the same kind of observations produced by the failing one. In our example when the health monitor produces an observation reporting that a thermal camera sensor is failing, the Reconfigurator detaches the thermal camera sensors and queries the knowledge base, looking for alternative sensors reporting the number of people in the same room where the failing sensors is deployed. This query will return the IR-based logical sensors that the Reconfigurator can activate, enabling a basic resilient behavior.

5.2. *Multimedia playback RDF modeling*

In this section, we extend the example of the previous subsection presenting both a physical presence sensor and a logical (and equivalent) one by using RDF. First of all, we model a museum room where multimedia content has to be played. The following Turtle [32] code[h] is an excerpt of the model describing the room and the thermal camera sensor that is used for tracking people and indirectly counting the number of persons that are in the room.

Listing 1. A room equipped with a thermal camera sensor tracking and counting people in a room.

```
# A room
gmus:room1 a sosa:FeatureOfInterest ;
  ssn:hasProperty gmus:people/room1 .

# The thermal camera sensor counting people in room1
gmus:thermalcamera1 a sosa:Sensor ;
  sosa:observes gmus:people/room1 ;
  sosa:isHostedBy gmus:room1 .

# The observable property about the number of people in room1
gmus:people/room1 a sosa:ObservableProperty ;
  ssn:isPropertyOf gmus:room1 .
```

[h] A list of the prefixes used in the following examples can be found at https://sites.google.com/site/logicalsensorsactuators/.

When the physical thermal camera sensor detects a person in the room, it reports its observation to the engine through the dedicated REST API, producing a new observation that is added to the system model in the knowledge base. The following excerpt is an example of observation generated by the thermal camera sensor reporting that the number of people in the room is five.

Listing 2. An example of observation created by the thermal camera sensor.

```
gmus:observation/thermalcamera1#2020-02-13T11:45:27.747
  a sosa:Observation ;
  sosa:hasSimpleResult 5 ;
  sosa:madeBySensor gmus:thermalcamera1 ;
  sosa:observedProperty gmus:people-room1 ;
  sosa:resultTime "2020-02-13T11:45:27.747+0:00"^^xsd:dateTime .
```

We can use the following SPARQL query to get the actual number of people in the room:

Listing 3. The SPARQL query to get the actual number of people in the room.

```
SELECT ?counter
  WHERE {
      ?obs a sosa:Observation;
      sosa:observedProperty gmus:people/room1 ;
      sosa:resultTime ?restime ;
      sosa:hasSimpleResult ?counter
  } ORDER BY DESC(?restime) LIMIT 1
```

By running the query, we get as result that there are five persons in the room, as we expected.

Listing 4. The result of the SPARQL query in Listing 3 reporting that the current number of people in the room is five.

```
  | ?counter |
  +----------+
  |     5    |
```

The museum is also monitored by several anti-theft, infrared sensors previously installed at the doors of the rooms and managed by dedicated software. The museum administrators realize that presence can also be detected by combining these infrared sensors, albeit with lower accuracy.

The following code is an excerpt of the model describing the door of the room and the two infrared sensors that observe user presence at the two door sides. The sensors `gmus:ir1` and `gmus:ir2` are hosted by the room door (`gmus:door1`). User presence in the detection area of the sensors is modeled as observable properties (`sosa:ObservableProperty`). These properties are observed (`sosa:observes`) by the sensors. The door is modeled both as a `sosa:FeatureOfInterest` (since properties pertaining it — i.e. the user presence near its side — are observed by sensors) and a `sosa:Platform` (since it hosts other entities — i.e. the infrared sensors).

Listing 5. An excerpt of the model defining a door in the room equipped with two infrared sensors observing people presence near the door sides.

```
# A door in room1
gmus:door1 a sosa:FeatureOfInterest ;
  a sosa:Platform ;
  sosa:isHostedBy gmus:room1 ;
  ssn:hasProperty gmus:presence/room1/ir1/zoneDoorInside ;
  ssn:hasProperty gmus:presence/room1/ir2/zoneDoorOutside .

# The observable property about presence in the inside of door1
gmus:presence/room1/ir1/zoneDoorInside a sosa:ObservableProperty ;
  ssn:isPropertyOf gmus:door1 .

# The observable property about presence in the outside of door1
gmus:presence/room1/ir2/zoneDoorOutside a sosa:ObservableProperty ;
  ssn:isPropertyOf gmus:door1 .

# The sensor on the inside side of door1
gmus:ir1 a sosa:Sensor ;
  sosa:observes gmus:presence/room1/ir1/zoneDoorInside ;
  sosa:isHostedBy gmus:door1 .

# The sensor on the outside side of door1
gmus:ir2 a sosa:Sensor ;
  sosa:observes gmus:presence/room1/ir2/zoneDoorOutside ;
  sosa:isHostedBy gmus:door1 .
```

When a person crosses the room door, each of the infrared sensors produces observations reporting the user presence near the door sides. The following code is an example of two observations produced by the infrared sensors. The instant of time when the observations were completed is reported using the `sosa:resultTime` property. The interval between the first observation and the next in the example is less than one second (0.742 s).

Listing 6. Two observations produced by the infrared sensors.

```
# An observation by the first infrared sensor
gmus:observation/ir1/1 a sosa:Observation ;
  sosa:madeBySensor gmus:ir1 ;
  sosa:hasFeatureOfInterest gmus:door1 ;
  sosa:observedProperty gmus:presence/room1/ir1/zoneDoorInside ;
  sosa:resultTime "2020-02-13T11:50:18.853+00:00"^^xsd:dateTime .

# An observation by the second infrared sensor
gmus:observation/ir2/1 a sosa:Observation ;
  sosa:madeBySensor gmus:ir2 ;
  sosa:hasFeatureOfInterest gmus:door1 ;
  sosa:observedProperty gmus:presence/room1/ir2/zoneDoorOutside ;
  sosa:resultTime "2020-02-13T11:50:18.111+00:00"^^xsd:dateTime .
```

When the physical presence sensor fails, it can be replaced by an "equivalent" logical sensor. The following code is an excerpt of the model describing the logical presence sensor. This sensor is associated to a software procedure that combines the two infrared sensors to update the number of people in the room. The procedure is in turn related to an actionable behavior specifying the action to perform.

Listing 7. An excerpt of the model with the definition of the logical presence sensor that combines the two infrared sensors, and the related software procedure.

```
# The logical sensor that observes people presence combining the two
    infrared sensors
gmus:ls1 a lsa:LogicalSensor ;
 a sosa:Sensor ;
 sosa:observes gmus:people/room1 ;
 sosa:isHostedBy gmus:triplestore ;
 sosa:hasSubSystem gmus:ir1 ;
 sosa:hasSubSystem gmus:ir2 .

gmus:entrance/door1/room1 a lsa:SoftwareProcedure ;
 a sosa:Procedure ;
 ssn:implementedBy gmus:ls1 ;
 ssn:hasInput gmus:presence/room1/ir1/zoneDoorInside ;
 ssn:hasInput gmus:presence/room1/ir2/zoneDoorOutside ;
 ssn:hasInput gmus:people/room1 ;
 lsa:hasBehavior gmus:doorRoomEntrance/behavior .

gmus:doorRoomEntrance/behavior a lsa:ActionableBehavior ;
 lsa:hasActionSpecification gmus:doorRoomEntrance/behavior/1/
  actionable .
```

The following SPARQL query is the action for the logical sensor and it is executed by the engine. It is triggered when observations pertaining the inputs of the related procedures are added to the knowledge base (e.g. when an infrared sensor detects the presence of a user). The query is responsible of updating the actual number of people

in the room, and is contained in the model with other details (e.g. the agent responsible of executing it) we omit for space limitations.

Listing 8. The SPARQL query used by the logical sensor to combine the last observations generated by the two infrared sensors. If a person crosses the door, then a new observation updating the number of people in the room is created.

```
CONSTRUCT {
  ?newObservation a sosa:Observation;
   sosa:madeBySensor gmus:ls1;
   sosa:observedProperty gmus:people/room1;
   sosa:resultTime ?now;
   sosa:hasSimpleResult ?newResult.
} WHERE {
  {
    SELECT ?timeObs1
    WHERE {
      ?observationIr1 a sosa:Observation;
      sosa:observedProperty gmus:presence/room1/ir1/zoneDoorInside;
      sosa:madeBySensor gmus:ir1;
      sosa:resultTime ?timeObs1.
    } ORDER BY DESC(?timeObs1) LIMIT 1
  }
  {
    SELECT ?timeObs2
    WHERE {
      ?observationIr2 a sosa:Observation ;
      sosa:observedProperty gmus:presence/room1/ir2/zoneDoorOutside;
      sosa:madeBySensor gmus:ir2;
      sosa:resultTime ?timeObs2.
   } ORDER BY DESC(?timeObs2) LIMIT 1
  }
  {
    SELECT ?numPeople_last
    WHERE {
      ?otherObservation a sosa:Observation;
        sosa:madeBySensor gmus:ls1;
        sosa:observedProperty gmus:people/room1;
        sosa:resultTime ?time;
        sosa:hasSimpleResult ?numPeople_last.
    } ORDER BY DESC(?time) LIMIT 1
  }
  BIND(?timeObs1 - ?timeObs2 as ?timeDifference)
  FILTER(?timeDifference < "PT1.500S"^^xsd:duration && ?timeDifference
     > "-PT1.500S"^^xsd:duration)
  BIND(IF((?timeObs1 > ?timeObs2), ?numPeople_last+1, IF(?
    numPeople_last > 0, ?numPeople_last-1, 0)) as ?newResult)
  BIND(now() as ?now)
  BIND(IF(BOUND(?timeDifference), IRI(CONCAT("gmus:observation/ls1/1#"
    , STR(?now))), ?timeDifference) as ?newObservation)
}
```

In the outer WHERE, the query collects the timestamps of the last two observations generated by the infrared sensors and the actual number of people in the room (see the three nested SELECT queries). The logical sensor works as follows: if the two observations about people presence at the door sides (i.e. inside and outside the room) occur in a short timespan (i.e. in 1.5 s), then they can be interpreted as people entering/exiting the room; otherwise they are ignored. This is implemented by the first FILTER clause. A new variable (`?newResult`) is introduced to compute the updated number of people in the room by increasing the counter if the observation on the outside precedes the one on the inside, and decreasing it otherwise. Another variable (`?newObservation`) is used to generate the URI of the new observation, obtained by concatenating a fixed prefix and a timestamp. If the result set generated by the outer WHERE clause is not empty (i.e. a people movement through the door has been identified), a new observation is created by the CONSTRUCT clause. The observation contains information about the updated number of people in the room (see `sosa:hasSimpleResult)`, a timestamp (see `sosa:resultTime`), the sensor that performed the observation (see `sosa:madeBySensor`) and the property the observation refers to (see `sosa:observedProperty`).

As described above, the query is executed when new observations by the infrared sensors are added into the knowledge base. For instance, if a person crosses the door and the two infrared sensors generate the two observations reported in Listing 6, the engine activates the logical sensor by executing the SPARQL query, that produces the following result.

Listing 9. The new observation created by the logical sensor.

```
gmus:observation/ls1#2020-02-13T11:50:19.053
    a sosa:Observation ;
    sosa:hasSimpleResult 6 ;
    sosa:madeBySensor gmus:ls1 ;
    sosa:observedProperty gmus:people/room1 ;
    sosa:resultTime "2020-02-13T11:50:19.053+0:00"^^xsd:dateTime .
```

Since the observation about people movement outside the room precedes the observation about the inside, a person entering the room is identified, and an observation incrementing the number of people in the room (from 5 to 6) is generated and added into the knowledge base. If we run again the query in Listing 3, we get as result that there are six persons in the room.

Listing 10. The result of the SPARQL query in Listing 3 reporting that the current number of people in the room is six.

```
| ?counter |
+---------+
|    6     |
```

The last observation about the current number of people in the room created by the logical sensor will in turn trigger the logical actuator that controls the multimedia playback. In this case, no action is performed (since the playback was already active).

5.3. *Modeling logical sensors for resilience*

The infrared sensors are combined in a logical sensor (`gmus:ls1`) that observes the presence in the room (`gmus:people/room1`) like the thermal camera sensor. It is composed of two subsystems (i.e. the two infrared sensors). The software procedure related to the logical sensor (`ssn:implementedBy`) is `gmus:entrance/door1/room1`. `gmus:doorRoomEntrance/behavior` is the behavior of the sensor (i.e. a detailed description of the actions to perform to execute the procedure), and is related to it via the `lsa:hasBehavior` property.

The sensors modeled above can also be used to implement a resilience mechanism: e.g. when the thermal camera sensor fails it can be replaced by the logical sensor built atop the anti-theft ones. The task of monitoring the state of the thermal camera sensor, and replacing it with the logical sensor combining infrared sensors (by changing their binding state, detaching the faulty one and attaching the equivalent logical one) is performed by a logical actuator that acts as a reconfigurator. Due to space limitation we do not describe here the details of the reconfiguration mechanism. The interested reader can find more information in [33, 34].

6. Performance Evaluation

This section describes a set of experiments we performed in order to evaluate the ability of the prototype implementation to handle different levels of workloads. It should be noticed that the RDF triplestore (Apache Jena) and its SPARQL processor (ARQ) are designed as reference implementations for most major RDF-related standards: they focus on features rather than speed. No indexing is used to speed up the retrieval of triples and the concurrency management operates at a full database level. This is reflected in our prototype that, while fully functional, has not been designed for high performance. However, even in its current implementation, the prototype can be adopted in several real-world scenarios.

In our experiments, we ran an instance of the semantic engine in a workstation with an i5-8400 hexa-core CPU and 16 GB of RAM. Clients simulating the workload interact with the semantic engine using its REST API to report (synthetic) observations in the form of HTTP POST requests. These requests are parsed and transformed into RDF triples representing the observations and are added to the knowledge base. Depending on the observed properties of these observations and on the presence of logical elements (sensors and/or actuators) represented in the knowledge base, the appearance of the observations, triples can trigger the activation of these logical elements and the execution of the related behaviors.

The load tests have been performed using Locust[i] with a swarm of containerized clients spread across three physical machines. Simulated clients send a request each 10 s; the load is varied by changing the overall number of clients. To better respect a typical IoT scenario where the rate of the requests does not depend on the response time of the system, our experiments have been realized by imposing a constant rate of requests to the systems; we then analyzed its ability to cope with this rate and the response time perceived by the clients.

In real-world scenarios, the overall request rate is the result of two main factors: the number of sensors composing the system and the rate with which they produce observations. Different kinds of sensors, in fact, can be characterized by large differences with respect to this latter parameter: it can be expected that temperature sensors for domotic applications report observations every few minutes whereas gyroscopic sensors for flying drones can produce observations every few milliseconds. The results of the tests presented below should then be analyzed with respect to the overall expected rate of observation produced for the specific application scenario: the ability to handle, for example, 100 requests per second with a delay of 5 ms could be applied to a scenario in which 100 sensors report an observation each second as well as a scenario in which five sensors produce 20 observations per second.

In our test, in those cases in which logical elements are activated, we also calculated the *activation time*, i.e. the time from the moment the client reports an observation until the execution of the logical elements triggered by this observation completes. The behavior of the prototype, with respect to when the control is returned to the client, can be configured in the following ways: clients can simply report an observation that is put in a queue for later insertion in the triplestore or can wait for the actual insertion. In these experiments the latter strategy is applied.

The ability of the system to deal with a number of clients is largely dependent on whether they trigger the activation of logical elements, since this implies the execution of the related behavior (which often also includes the retrieval of data from the knowledge base). For this reason, we created three different test scenarios.

Scenario S1 sees the clients reporting observations that do not trigger any logical element. In this scenario, we basically evaluate the baseline performance of the system including the management of network requests and the adding of triples to the triplestore.

In scenario S2, the clients report observations that trigger a logical element with an elementary behavior. In our tests, we reported temperature observations in Celsius degrees. They triggered the execution of logical sensors that produce corresponding temperature observations in Fahrenheit degrees.

Scenario S3 uses logical elements with a more complex behavior: we set up an environment containing the IR-based logical presence sensors described in Sec. 5.2; in these experiments all clients simulate couples of IR observations leading to the activation of the logical presence sensors and to the creation of the corresponding

[i] https://www.locust.io/.

observations (notice that this is different from a real-world scenario in which it can be expected that only a small subset of the passages recorded by the IR sensors are about persons getting through the doors). Given the low level of optimization in Jena and ARQ, the execution of the query describing the behavior of these logical sensors (see Listing 9) results in a write transaction that executes in total isolation with no concurrency. The inner select in Listing 9 analyzes all the triples of the triplestore creating lists that are then sorted, which is a sort of worst-case scenario for the current implementation.

Here is a brief report of our measurements for S1.

Request rate (req/s)	Response time (avg, ms)
10	12
100	12
1000	13
2000	330

These results show that up to 1000 req/s the prototype responds promptly with minimal delay. When surpassing this rate, the response time grows rapidly.

This is the report for scenario S2. Here, we also include the activation time since (as previously explained) the observations trigger a logical element.

Request rate (req/s)	Response time (avg, ms)	Activation time (avg, ms)
10	12	58
100	12	177
1000	63	709
1200	9138	958

With a higher number of clients, the response time surpasses the delay between the requests limiting the overall rate. That means that the clients are not able to increase the rate of their requests because they wait too long for each request they post. In other words, the system has reached its upper performance limit.

The measurements for S3 are as follows:

Request rate (req/s)	Response time (avg, ms)	Activation time (avg, ms)
10	12	9
50	12	24
80	641	536

With a higher number of clients, the response time surpasses the delay between the requests limiting the overall rate. Notice that, in this specific scenario, the relatively limited number of concurrent requests that the system is able to handle with reasonable performances is mostly due to transactional contention over the triplestore, as witnessed by an average CPU load of a mere 32%.

7. Discussion

The proposed approach allows for declarative definitions of reactive behaviors in a bi-causally connected system. In fact, both the model of the context and that of the system (in terms of logical sensors/actuators and their behaviors) are represented in a semantic format (e.g. by RDF triples). This allows to *change the overall behavior of the system by only manipulating the knowledge base*: *at runtime* new logical sensors/ actuators can be *defined*, the behavior of the existing ones can be *modified*, existing sensors/actuators can be *deleted*.

A further advantage of the approach is that self-adaptive behaviors can be easily implemented by defining specific sensors and actuators, such as failure detectors and reconfigurators, as shown in Secs. 4.3 and 5.3.

As Sec. 6 shows, the benefits above are obtained by slightly sacrificing performance that could limit the applicability of the solution to applications with strong real-time requirements. However, the observed response times under significant traffic intensity are reasonable and acceptable for a wide class of applications where reactions to external stimuli take place in negligible time with respect to the response delays of the environment (e.g. brightness drops below a threshold and a lamp is turned on, a person enters a room and a counter is incremented, a number of people in a room reach a specified value and a media player is turned on, the perceived temperature is high and a conditioning system is turned on, etc.).

Note also that there are scenarios where a large stream of data is produced in which our prototype can still be applied; this includes the contexts in which a hierarchical approach using instances of the prototypes as leaf nodes is possible (which is usually the case for smart cities and similar circumstances).

It is also worth noting that the proposed approach could appear too much verbose for modelers and developers who must define some complex artifacts with description logic for modeling the system to manage. However, this problem can be easily addressed in the future by providing supporting tools for graphical programming, like recent tools for IoT programming, such as NodeRED.

8. Conclusions and Future Work

In this paper, we presented a proposal for an extension of the SSN ontology to support modeling of LSA, and their behaviors. The extension enables reactive behaviors of context-aware applications by defining the decision logic that exploits sensor observations to trigger actions. The ontology is accompanied by a an architecture that supports behaviors definition and the interaction with the real devices in the physical world. A prototype of the architecture has been implemented by using Jena, SPARQL and RESTful APIs for the interaction with the external environment.

We discussed and validated the proposed ontology extension and the supporting architecture with the help of an application in the domain of smart buildings for

cultural heritage. The ontology extension and the related architecture represent the first step towards the definition of a more complex platform for context-awareness able to take into account failures and adaptation policies. We plan to extend our semantic model to include the specification of system requirements and goals [35], in order to use these elements to guide the choice of the adaptation policy for the reactive system. We also intend to use an approach similar to the one discussed in [36], so as to allow designers to include in the model other kinds of information (e.g. documentation, technical comments, versioning and change tracking, etc.).

Acknowledgments

This paper has been supported by the MIUR PRIN 2015 GAUSS Project and MIUR PON VASARI Project.

References

1. D. Harel and A. Pnueli, On the development of reactive systems, in *Logics and Models of Concurrent Systems* (Springer, 1985), pp. 477–498.
2. N. Halbwachs, Synchronous programming of reactive systems, in *Computer Aided Verification*, eds. A. J. Hu and M. Y. Vardi (Springer, Berlin, Heidelberg, 1998), pp. 1–16.
3. A. Furno and E. Zimeo, Context-aware composition of semantic web services, *Mobile Netw. Appl.* **19**(2) (2014) 235–248.
4. A. Haller, K. Janowicz, S. Cox, D. Le Phuoc, K. Taylor and M. Lefrançois, Semantic sensor network ontology, *W3C Recommendation W3C* (2017).
5. R. J. Wieringa, *Design Methods for Reactive Systems: Yourdon, Statemate, and The UML* (Elsevier, 2003).
6. R. Kuhn, B. Hanafee and J. Allen, *Reactive Design Patterns* (Manning Publications Co., 2017).
7. M. Szvetits and U. Zdun, Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime, *Softw. Syst. Model.* **15**(1) (2016) 31–69.
8. H. B. Pötter and A. Sztajnberg, Adapting heterogeneous devices into an iot context-aware infrastructure, in *2016 IEEE/ACM 11th Int. Symp. Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, IEEE, 2016, pp. 64–74.
9. G. Tamura, N. M. Villegas, H. A. Muller, L. Duchien and L. Seinturier, 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*, IEEE, 2013, pp. 153–162.
10. R. Seiger, S. Herrmann and U. Aßmann, Self-healing for distributed workflows in the internet of things, *2017 IEEE Int. Conf. Software Architecture Workshops (ICSAW)*, 2017, pp. 72–79.
11. F. Poggi, D. Rossi and P. Ciancarini, Integrating semantic run-time models for adaptive software systems, *J. Web Eng.* **18**(1–3) (2019) 1–42, doi: 10.13052/jwe1540-9589.18131.
12. D. Rossi, F. Poggi and P. Ciancarini, Dynamic high-level in self-adaptive systems, *2017 6th Int. Conf. Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, IEEE, 2017, pp. 49–60, doi:10.1109/ICRITO.2017.8342398.

13. F. Poggi, D. Rossi, P. Ciancarini and L. Bompani, Semantic run-time models for self-adaptive systems: a case study, in *2016 IEEE 25th Int. Conf. Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, IEEE, 2016, pp. 50–55, doi: 10.1109/WETICE.2016.20.
14. F. Poggi, D. Rossi, P. Ciancarini and L. Bompani, An application of semantic technologies to self adaptations, in *2016 IEEE 2nd Int. Forum on Research and Technologies for Society and Industry Leveraging a Better Tomorrow (RTSI)*, IEEE, 2016, pp. 1–6, doi:10.1109/RTSI.2016.7740548.
15. K. Angelopoulos, V. E. S. Souza and J. Mylopoulos, Capturing variability in adaptation spaces: A three-peaks approach, *Int. Conf. Conceptual Modeling*, Springer, 2015, pp.384–398.
16. A. C. Shaw, Software descriptions with flow expressions, *IEEE Trans. Softw. Eng.* **SE-4**(3) (1978) 242–254.
17. J. Whittle, P. Sawyer, N. Bencomo, B. H. Cheng and J.-M. Bruel, Relax: Incorporating uncertainty into the specification of self-adaptive systems, *2009 17th IEEE Int. Requirements Engineering Conf.*, IEEE, 2009, pp. 79–88.
18. A. U. Frank, Tiers of ontology and consistency constraints in geographical information systems, *Int. J. Geograph. Inf. Sci.* **15**(7) (2001) 667–678.
19. C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan and D. Riboni, A survey of context modelling and reasoning techniques, *Pervasive Mobile Comput.* **6**(2) (2010) 161–180.
20. T. Pederson, C. Ardito, P. Bottoni and M. F. Costabile, A general-purpose context modeling architecture for adaptive mobile services, *Int. Conf. Conceptual Modeling*, Springer, 2008, pp. 208–217.
21. H. Chen, F. Perich, T. Finin and A. Joshi, Soupa: Standard ontology for ubiquitous and pervasive applications, *The First Annual Int. Conf. Mobile and Ubiquitous Systems: Networking and Services*, IEEE, 2004, pp. 258–267.
22. C. Perera, A. Zaslavsky, P. Christen and D. Georgakopoulos, Context aware computing for the internet of things: A survey, *IEEE Commun. Surveys Tutorials* **16**(1) (2014) 414–454.
23. S. Kaebisch, T. Kamiya, M. McCool, V. Charpenay and M. Kovatsch, Web of things (WoT) thing description, W3C Recommendation 9 April 2020 (Link errors corrected 23 June 2020). W3C, 2020.
24. K. Janowicz, A. Haller, S. J. Cox, D. Le Phuoc and M. Lefrançois, SOSA: A lightweight ontology for sensors, observations, samples, and actuators, *J. Web Semantics* **56** (2019) 1–10.
25. S. Sagar, M. Lefrançois, I. Rebai, M. Khemaja, S. Garlatti, J. Feki and L. Médini, Modeling smart sensors on top of sosa/ssn and WOT TD with the semantic smart sensor network (S3N) modular ontology, *9th Int. Semantic Sensor Network Workshop*, Monterey, United States, 2018.
26. M. Hölzl and T. Gabor, Reasoning and learning for awareness and adaptation, in *Software Engineering for Collective Autonomic Systems*, Springer, 2015, pp. 249–290.
27. R. Falco, A. Gangemi, S. Peroni, D. Shotton and F. Vitali, Modelling owl ontologies with graffoo, in *European Semantic Web Conference*, Springer, 2014, pp. 320–325.
28. D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne *et al.*, OWL-S: Semantic markup for web services, *W3C Member Submission* 2004, https://www.w3.org/Submission/OWL-S/.
29. M. Rospocher, C. Ghidini and L. Serafini, An ontology for the business process modelling notation. in *Formal Ontology in Information Systems*, 2014, pp. 133–146.

30. M. Lefrançois, Planned etsi saref extensions based on the w3c&ogc sosa/ssn-compatible seas ontology paaerns, *Proc. Workshop on Semantic Interoperability and Standardization in the IoT, SIS-IoT*, 2017, p. 11.
31. E. Giallonardo, C. Sorrentino and E. Zimeo, Querying a complex web-based kb for cultural heritage preservation, *2017 2nd Int. Conf. Knowledge Engineering and Applications (ICKEA)*, IEEE, 2017, pp. 183–188.
32. E. Prud'hommeaux, G. Carothers, D. Beckett and T. Berners-Lee, Turtle–terse rdf triple language, 2013, https://www.w3.org/TR/2013/CR-turtle-20130219/.
33. E. Giallonardo, F. Poggi, D. Rossi and E. Zimeo, Resilient reactive systems based on runtime semantic models, in *2019 IEEE Int. Symp. Software Reliability Engineering Workshops*, 2019, pp. 177–184, doi: 10.1109/ISSREW.2019.00069.
34. E. Giallonardo, F. Poggi, D. Rossi and E. Zimeo, Context-aware reactive systems based on runtime semantic models, *Proc. 31st Int. Conf. Software Engineering and Knowledge Engineering*, 2019, pp. 301–306, doi: 10.18293/SEKE2019-169.
35. D. Rossi, F. Poggi and P. Ciancarini, Dynamic high-level requirements in self-adaptive systems, *Proc. 33rd Annual ACM Symp. Applied Computing*, ACM, 2018, pp. 128–137, doi: 10.1145/3167132.3167143.
36. G. Barabucci, A. Di Iorio, S. Peroni, F. Poggi and F. Vitali, Annotations with EARMARK in practice: A fairy tale, *Proc. 1st Int. Workshop on Collaborative Annotations in Shared Environment: Metadata, Vocabularies and Techniques in the Digital Humanities*, ACM, 2013, pp. 11–18, doi: 10.1145/2517978.2517990.