SPECIAL ISSUE ARTICLE

A model-based infrastructure for the specification and runtime execution of self-adaptive IoT architectures

Iván Alfonso[1,2] · **Kelly Garcés[1]** · **Harold Castro[1]** · **Jordi Cabot[2,3]**

Abstract

To meet increasingly restrictive requirements and improve quality of service (QoS), Internet of Things (IoT) systems have embraced multi-layered architectures leveraging edge and fog computing. However, the dynamic and changing IoT environment can impact QoS due to unexpected events. Therefore, proactive evolution and adaptation of the IoT system becomes a necessity and concern. In this paper, we present a model-based approach for the specification and execution of self-adaptive multi-layered IoT systems. Our proposal comprises the design of a domain-specific language (DSL) for the specification of such architectures, and a runtime framework to support the system behaviuor and its self-adaptation at runtime. The code for the deployment of the IoT system and the execution of the runtime framework is automatically produced by our prototype code generator. Moreover, we also show and validate the extensibility of such DSL by applying it to the domain of underground mining. The complete infrastructure (modeling tool, generator and runtime components) is available in a online open source repository.

Keywords Domain-specific language · Internet of things · Self-adaptive system · Edge computing · Fog computing

✉ Iván Alfonso
   id.alfonso@uniandes.edu.co

   Kelly Garcés
   kj.garces971@uniandes.edu.co

   Harold Castro
   hcastro@uniandes.edu.co

   Jordi Cabot
   jordi.cabot@icrea.cat

1   Department of Systems and Computing Engineering, Universidad de los Andes, Bogotá, Colombia

2   Universitat Oberta de Catalunya, Barcelona, Spain

3   ICREA, Barcelona, Spain

1 Introduction

Nowadays, billions of connected devices sense, communicate, and share information about their environment. To manage all these devices, traditional Internet of Things (IoT) systems rely on cloud-based architectures, which allocate all processing and storage capabilities to cloud servers. Although cloud-based IoT architectures have advantages such as reduced maintenance costs and application development efforts, they also have limitations in bandwidth and communication delays [1]. Given these limitations, edge and fog computing have emerged with the goal of distributing processing and storage closer to data sources (i.e. things). As a result, new IoT systems aim to leverage the advantages of edge, fog, and cloud computing by following a multi-layered architecture.

Nevertheless, creating such complex designs is a challenging task. Even more challenging is managing and adapting IoT systems at runtime to ensure the optimal performance of the system while facing changes in the environmental conditions. Indeed, IoT systems are commonly exposed to changing environments that induce unexpected events at runtime (such as unstable signal strength, latency growth, and software failures) that can impact its Quality of Service (QoS). To deal with such events, a number of runtime adaptation rules should be automatically applied, e.g. *architectural adaptations* such as auto-scaling and offloading tasks.

In this sense, a better support to define and execute complex IoT systems and their (self)adaptation rules to semi-automate the deployment and evolution process is necessary [2]. A usual strategy when it comes to modeling complex domains is to develop a domain-specific language (DSL) for that domain [3]. In short, a DSL offers a set of abstractions and vocabulary closer to the one already employed by domain experts. Nevertheless, current IoT modeling approaches do not typically cover multi-layered architectures [4–7] and even less include a sublanguage to ease the definition of the dynamic rules governing the IoT system. Our research aims at overcoming this situation by presenting a model-based infrastructure for the specification and runtime execution of multi-layered IoT architectures, including self-adaptation rules. Our proposal combines a DSL for the specification of static and dynamic aspects of this type of systems together with a runtime infrastructure and a code-generator able to semi-automate their deployment and runtime monitoring and adaptation.

This work is an extension of our study presented in [8], in which we proposed a first version of a DSL for IoT systems and a proof of concept of a code generator. The current work extends this previous contribution to the following aspects:

- Metamodel improvement: we have enhanced the metamodel to support modeling new DSL concepts such as sensor threshold values, publish/subscribe messaging and data persistence for containers.
- Runtime support: we have developed a framework based on the MAPE-K [9] loop to automatically monitor, execute the expected behaviour and self-adapt the IoT

system by executing the IoT execution rules modeled with the DSL, including rules for the self-adaptation of the system to ensure its QoS.

- Code generator enhancements: We now generate the code required to support the execution of the whole system at runtime (including the code for infrastructure monitoring and system management tools).
- a DSL extension for the mining industry: we propose an extension of our DSL focused on the modeling and operation of IoT systems in the underground mining industry, highlighting the ability to reuse the DSL in other domains.
- Empirical evaluations: we have designed and conducted empirical experiments to validate the expressiveness and usability of our DSL and the correctness of the generated code.

The remainder of the paper is organized as follows: Section 2 presents a running example to illustrate our approach. Section 3 introduces the DSL to specify multi-layered IoT systems and adaptation rules. Section 4 present an extension of our DSL for IoT systems deployed in underground mining. In Sect. 5, we present the DSL implementation, the framework to support IoT systems at runtime, and code generation. In Sect. 6, we validate the usability of our DSL and the generated runtime. Finally, the related work to this research is summarized in Sect. 7, and the conclusions and future work are presented in Sect. 8.

2 Running example

We will use a Smart Building scenario as a running example to illustrate our approach. Smart buildings seek to optimize different aspects such as ventilation, heating, lighting, energy efficiency, etc [10].

For the purposes of our example, let's assume that a hotel company (*Hotel Beach*) wants to reduce fire risks by automating disaster management in its hotels. A fire alarm and monitoring system are implemented in each of the company's hotels. We will assume that all buildings (hotels) have three floors with two rooms each. Fig. 1 presents an overview of the 1st floor. Based on this, the infrastructure (device, edge, and cloud layers) of the company hotel IoT system would be as follows:

- *Device layer* Each room has a temperature sensor, a carbon monoxide (CO) gas sensor, and a fire water valve. Furthermore, an alarm is deployed on the lobby. Each sensor has a threshold measurement to activate the corresponding alarm, e.g., a person should not be continuously exposed to CO gas level of 50 parts per million (ppm) for more than 8 hours.
- *Edge layer* In each room, an edge node receives the information collected by the sensors of the device layer and run a software container (C1 and C2) for analyzing sensor data in real time to check for the presence of smoke and generate an alarm state that activates the actuators. A fog node (linked to the edge nodes), located in this same floor, runs the C3 container (running App2, a machine learning model to predict fires), and C4 (running App3, in charge of receiving and distributing data, typically a Message Queuing Telemetry Transport (MQTT).
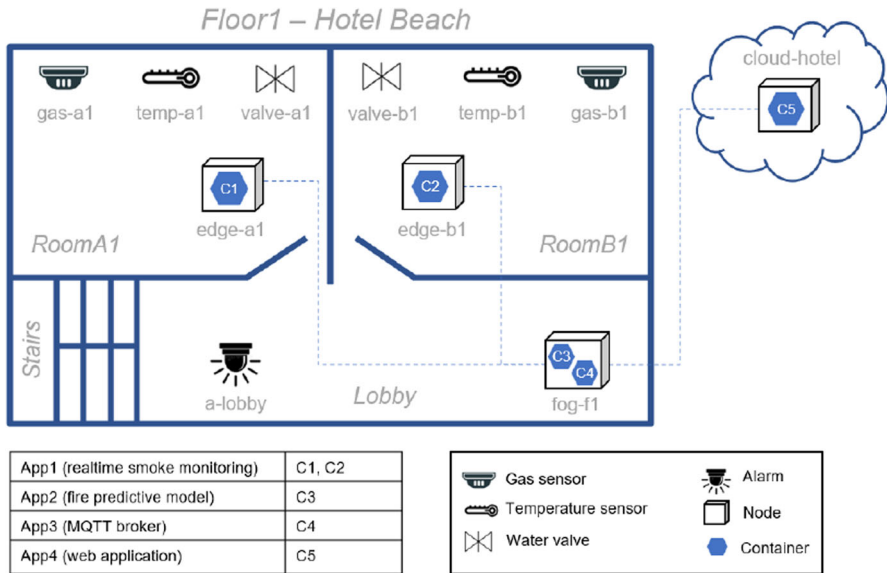
Fig. 1 Overview of the smart building IoT system, first floor

- *Cloud layer* The cloud layer has a cloud server node that runs the C5 container, a web application (App4) to display historical information of sensor data and of fire incidents in any of the hotel's property of the company.

This covers the static view of the system. But in any IoT system, there are critical applications that should be available all the time. For example, the availability of the containers running App1 (real-time smoke monitoring) should be guaranteed. However, some environmental factors can impact their availability. In these cases, the IoT system must self-adapt to guarantee its operation. For instance, a flooding could cause failures in the *edge-a1* node; then it will be necessary to migrate the *C1* container to another suitable node to ensure the continuous monitoring of the smoke presence.

Our research addresses this type of architectural adaptations by proposing a rule-based language for the runtime execution of IoT systems that can also be used to address their functional requirements. An example of this latter case would be a rule stating that when one of the room sensors detects CO gas greater than 400 ppm, an of the alarms should be triggered. The next section shows how we can model all these concepts.

3 A DSL for the specification of multi-layered IoT systems

A DSL is defined by three core ingredients [3]: the abstract syntax (i.e., the concepts of the DSL and their relationships), the concrete syntax (the notation to represent those concepts), and its semantics which are hardly ever formalized but based on the shared understanding of the domain. In this chapter, we present our DSL for modeling multi-layered IoT architectures (Sect. 3.1) and their dynamic (3.2) rules.

3.1 Modeling the IoT architecture

Traditionally, IoT system architectures consisted of two layers: device and cloud [11]. The device layer is composed by sensors (devices that sense physical properties such as temperature and humidity), actuators (devices such as valves, fans, or alarms that can influence an automated process), and network devices that send data to the cloud layer, composed of servers that run the application logic. Today, multi-layered architectures based on edge and fog computing have emerged to increase the flexibility of pure cloud deployments and help meet non-functional IoT system requirements [12].

There are several definitions for edge and fog computing. [13] states that edge computing enables computations to be performed at the edge, supporting cloud and IoT services. On the other hand, [14] defines fog computing as a bridge between the cloud and the edge of the network to facilitate the deployment of new IoT applications providing computation, storage, and network services. There are strong similarities between these two concepts, but the main difference is where the processing takes place. While fog computing take place on LAN-connected nodes usually close to end user devices, edge computing takes place on personal devices directly connected to sensors and actuators, or on gateways physically close to these [14, 15].

Edge and fog computing can leverage containerization as a virtualization technology [16]. Containers, compared to virtual machines, are lightweight, simple to deploy, support multiple architectures, have a short start-up time, and are suitable for dealing with the heterogeneity of edge and fog nodes. In terms of communications, asynchronous message-based architectures are typically chosen, especially for IoT systems that require high scalability [17]. The publisher/subscriber pattern and the MQTT protocol are becoming the standard for Machine-to-Machine communications [18], where messages are sent (by publishers) to a message broker server and routed to destination clients (subscribers).

Our DSL enables the specification of all these concepts.

3.1.1 Abstract syntax

The abstract syntax of a DSL is commonly defined through a metamodel that represents the domain concepts and their relationships. Fig. 2 shows our metamodel that abstracts the concepts required to define multi-layered IoT architectures.

The sensors and actuators of the device layer are modeled using the *Sensor* and *Actuator* concepts that inherit from the *IoTDevice* concept. All *IoTDevice*s have a connectivity type (such as Ethernet, Wi-Fi, ZigBee, or another). The location of IoTDevices can be specified through geographic coordinates (*latitude* and *longitude* attributes). Moreover, *Sensors* and *Actuators* have a type represented by the concepts *SensorType* and *ActuatorType* to organize different sensors belonging to the same category and be able to define global rules for them. For instance, following the running example (Fig. 1), there are temperature and smoke type sensors, and there are valve and alarm type actuators. We also cover the concepts for specifying MQTT communications: *IoTDevices* are publishers or subscribers to a topic specified by the relationship to the *Topic*) concept. The gateway of an *IoTDevice* can be modeled through the *gateway* relationship with the *EdgeNode* concept. Via this gateway, the
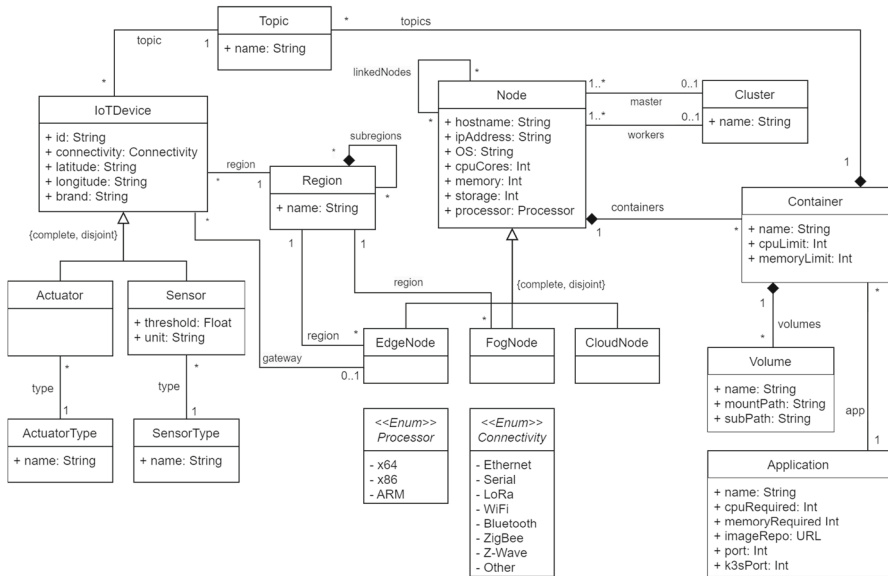
Fig. 2 Multi-layer IoT architecture metamodel

sensor can communicate with other nodes, e.g., the MQTT broker node. Additionally, the threshold value and unit of the monitored variable by a sensor can be represented through the attributes *threshold* and *unit*.

Physical (or even virtual) spaces such as rooms, stairs, buildings, or tunnels can be represented by the concept *Region*. A *Region* can contain subregions (relationship *subregion*s in the metamodel). For example, region *Floor1* (Fig. 1) contains subregions *Room1*, *Room2*, *Lobby*, and *Stairs*. *IoTDevice*s, *EdgeNode*s, and *FogNode*s are deployed and are located in a region or subregion (represented by *region* relationships in the metamodel). Back to the running example, the *edge-a1* node is located in the *RoomA1* region of *Floor1* of the *Hotel Beach*, while the *fog-f1* node is located in the *Lobby* region of *Floor1*.

Edge, fog and cloud nodes are all instances of *Node*, one of the key concepts of the metamodel. Communication between nodes can be specified via the *linkedNodes* relationship, as we may want to indicate what nodes on a certain layer could act as reference nodes in another layer. Nodes can also be grouped in clusters that work together. A *Node* can host several software containers according to its capabilities and resources (primarily *cpuCores*, *memory*, and *storage*). The CPU and memory usage of a container can be restricted through *cpuLimit* and *memoryLimit* attributes. Each software container runs an application (represented by the concept *Application*) that has a minimum of required resources specified by the attributes *cpuRequired* and *memmoryRequired*. The repository of the application image is specified through the *imageRepo* attribute. The container volumes and their paths (a mechanism for persisting data used and generated by containers) are represented by the *Volume* concept.
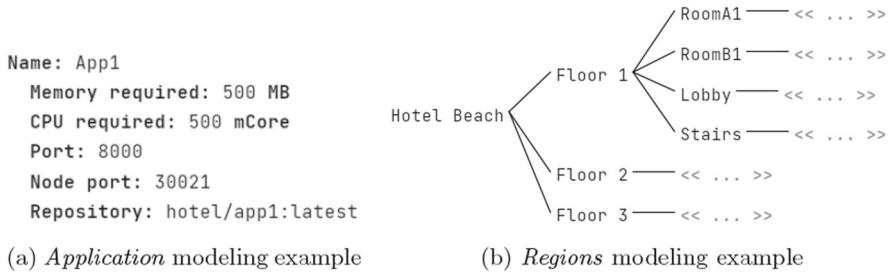
```
Name: App1
   Memory required: 500 MB
   CPU required: 500 mCore
   Port: 8000
   Node port: 30021
   Repository: hotel/app1:latest
```

(a) *Application* modeling example   (b) *Regions* modeling example

Fig. 3 *Application* and *Regions* modeling example

3.1.2 Concrete syntax

The concrete syntax refers to the type of notation (such as textual, or graphical) to represent the concepts of the metamodel. Graphical DSLs involve the development of models using graphic items such as blocks, arrows, axes, and so on. Textual DSLs involve modeling using configuration files and text notes. Though most DSLs employ a single type of notation, they could benefit from offering several alternative notations, each one targeting a different type of DSL user profile. This is the approach we follow here, leveraging the benefits of using a projectional editor.

Projectional editors enable the user's editing actions to directly change the Abstract Syntax Tree (AST) without using a parser [19]. That is, while the editing experience simulates that of classical parsing-based editors, there is a single representation of the model stored as an AST and rendered in a variety of perspectives thanks to the corresponding projectional editors that can deal with mixed-language code and support various notations such as tables, mathematical formulas, or diagrams.

Indeed, we take advantage of JetBrains Meta Programming Systems (MPS) projectional editors to define a set of complementary notations for the metamodel concepts. We blend textual, tabular, and tree view, depending on the element to be modeled. We next employ these notations to model our running example (Sect. 2). More technical details on the implementation of our concrete notations are presented in Sect. 5.

3.1.3 Example scenario

We present next how to model the IoT architecture of the running example (Sect. 2) using our DSL. First, Fig. 3a depicts the modeling of the application *App1*, including its technical requirements and repository address. Then, Fig. 3b shows the specification of the *Hotel Beach* regions, in particular those on *Floor 1* (four subregions: two *Rooms*, the *Lobby* and the *Stairs*).

Additionally, *IoT devices* can be modeled using a tabular notation. Fig. 4 shows the list of sensors and actuators located in the *RoomA1* region (we have omitted some parameters for readability purposes. To specify the nodes of the system architecture, we propose a tabular notation as well (Fig. 5 shows *edge-a1* and *fog-f1* node modeling). The node description includes the layer it belongs to, the hardware properties, the regions where it is located, and the hosted application containers. Each container

	Device	ID	Type	Unit	Threshold	Regions	Brand	Communic.	Gateway	Topic
1	Sensor	gas-a1	CO	ppm	50	RoomA1	Winsen	ZigBee	edge-a1	f1/rA1/smoke
2	Sensor	temp-a1	Temperature	°C	23	RoomA1	MLX	Z_Wave	edge-a1	f1/rA1/temp
3	Actuator	valve-a1	Valve	---	---	RoomA1	Bray	Serial	edge-a1	f1/rA1/valve

Fig. 4 IoT devices modeling example (tabular notation)

	Hostname	Layer	Properties	Regions	Linked nodes	Containers
1	edge-a1	Edge	Memory: 2000 MB Storage: 16000 MB CPU cores: 1 Core IP address: 192.168.10.1 Operating system: Raspbian Processor: ARM	RoomA1	fog-f1	* Name: C1 Application: App1 Memory limit: no limit CPU limit: no limit Volumes: << ... >>
2	fog-f1	Fog	Memory: 4000 MB Storage: 20000 MB CPU cores: 2 Cores IP address: 192.168.10.3 Operating system: Raspbian Processor: ARM	Lobby	cloud-hotel	* Name: C3 Application: App2 Memory limit: no limit CPU limit: no limit Volumes: << ... >> * Name: C4 Application: App3 Memory limit: no limit CPU limit: no limit Volumes: -> Name: mosquitto-config Mount path: /config/mqtt.conf Sub path: mosquitto.conf

Fig. 5 Nodes and containers modeling example (tabular notation)

includes the specification of the application to be executed, limits, and volumes (e.g., the volume of the *C4* container for the configuration of the MQTT broker).

3.2 Modeling dynamic rules

The dynamic environment of an IoT system requires dealing with expected an unexpected events. The former may trigger actions to comply with the standard behaviour of the system (e.g. to turn on an alarm upon detection of fire), unexpected ones may require a self-adaptation of the system itself to continue its normal operation. This section presents a rule-based language that can cover both types of events (and even mix them in a single rule). This facilitates an homogeneous of all the dynamic aspects of an IoT system.

To decide what unexpected environmental situation should we include and what the standard patterns of response are common in the self-adaptation of IoT systems, we rely on our previous systematic literature review [11]. For instance, the three architectural adaptations (offloading, scaling, and redeployment) addressed in this study were identified in the SLR. Our language covers all of them and even enables complex rules where policies involving several strategies can be attempted in a given order.

3.2.1 Abstract syntax

The metamodel representing the abstract syntax for defining the rules is depicted in Fig. 6.

Every rule is an instance of *Rule* that has a triggering condition, which is an expression. We reuse an existing *Expression* sublanguage to avoid redefining in our language

all the primitive data types and all the basic arithmetic and logic operations to manipulate them. Such Expression language could be, for instance, the Object Constraint Language (OCL), but to facilitate the implementation of the DSL later on, we directly reused the MPS *Baselanguage*. The metamodel extends the generic Expression concept by adding sensor and QoS conditions that can be combined also with all other types of expressions (e.g., *BinaryOperation*s, *Literal*s, or *BooleanConstant*s) in a complex conditional expression.

A *SensorCondition* represents the occurrence of an event resulting from the analysis of sensor data (e.g., the detection of dioxide carbon gas by the *gas-a1* sensor). Note that *SensorCondition* conditions can be linked to specific sensors or to sensor types to express conditions involving a group of sensors.

Similarly, the *QoSCondition* is a relational expression that represents a threshold of resource consumption or QoS metrics. This condition allows checking a *Metric* (such as Latency, CPU consumption, and others) on a specific node or a group of nodes belonging to a *Region* or *Cluster*. For example, the condition $cpu(HotelBeach->edge\_nodes) > 50\%$ is triggered when the CPU consumption on the edge nodes of the *Hotel* exceeds 50%.

Moreover, we can define that the condition should be true over a certain period (to avoid firing the rule in reaction to minor disturbances) before executing the rule. Once fired, all or some of the actions are executed in order, depending on the *allActions* attribute. If set to false, only the number of *Action*s specified by the attribute *actionsQuantity* must be executed, starting with the first one in order and continuing until the required number of actions have been successfully applied.

For the sake of clarity, we have grouped the rule concepts into two categories: *Architectural Adaptation Rules* and *Functional Rules* but note that they could be all combined, e.g., a sensor event could trigger at the same time a functional response such as triggering an alarm and, at the same time, an automatic self-adaptation action, such as scaling of apps related to the event to make sure the IoT system has the capacity to collect more relevant data).

Among the self-adaptation patterns, the *Offloading* action consists in migrating a container from a source node to a destination node. This migration can be among nodes of different layers. The *container* relationship represents the container that will be offloaded. The target node is specified by the *targetNode* relationship. However, if the target node does not have the resources to host the container, a cluster or a group of nodes in a *Region* can be specified (*targetRegion* and *targetCluster* relationships) to offload the container. The *Scaling* action involves deploying replicas of an application. This application is represented by the *app* relationship, and the number of replicas to be deployed is defined by the *instances* attribute. The replicas of the application are deployed in one or several nodes of the system represented by the *targetNodes*, *targetCluster*, and *targetRegion* relationships. The *Redeployment* action consists in stopping and redeploying a container running on a node. The container to redeploy is indicated by the *container* relationship. Finally, the *OperateActuator* action is to control the actuators of the system (e.g., to activate or deactivate an alarm). The message attribute represents the message that will be published in the broker and interpreted by the actuator.
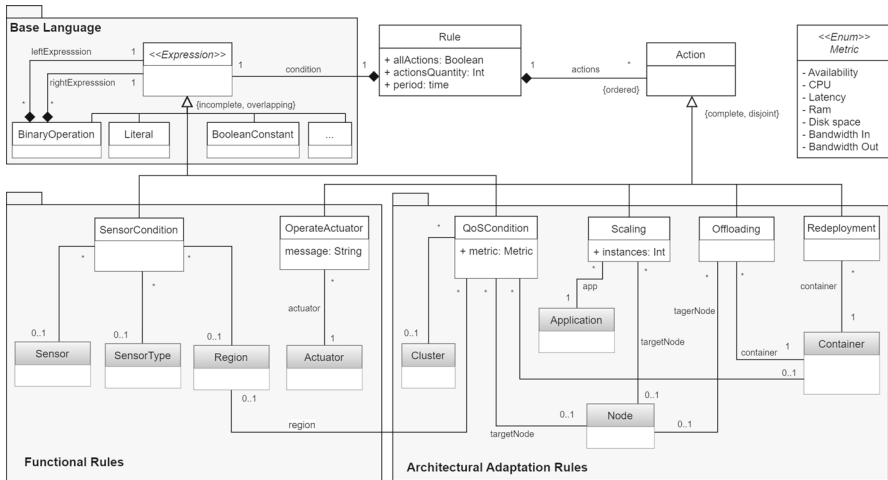
Fig. 6 IoT system dynamic rules metamodel

3.2.2 Concrete syntax

Rules are specified thanks to a textual notation using as keywords the names of the metaclasses of the abstract syntax. The conditions follow the grammar of a relational expression with the use of mathematical symbols (such as $<$, $>$, and $=$) and logical operators (such as & and ∥). The rule editor (see Sect. 5) offers a powerful autocomplete feature to guide the designer through the rule creation process.

3.2.3 Example scenario

We show how to use the rule's concrete syntax to model two example rules from the smart building example.

First, to guarantee the execution of the *C4* container deployed on the *fog-f1* node, we modeled the rule as shown in (Fig. 7a). This rule offloads the container *C4* hosted on node *fog-f1* to a nearby node (e.g., node *edge-b1*) when the CPU consumption exceeds 80% for one minute. If the *edge-b1* node does not have the necessary resources to host that new container (when the rule is activated), a Region (e.g., *Floor1*) can be specified so that a suitable node will be searched there. However, if this offloading action cannot be executed, for example, because in *Floor1* there is no node capable of hosting the container, then we must define a backup action. Therefore, we have modeled a second action (*Scaling*) to deploy a new container instance of the *App3* application on any of the nodes of the *Hotel Beach*. When a list of actions is specified, the checkbox *all actions* controls whether all or only a certain number of them should be performed. All actions in the list will be performed. For *Rule 1*, only one action (the first one, or the second one if the first one fails) will be executed.

Secondly, we model another rule (see Fig. 7b) to activate the alarm (a-lobby) when any gas sensors in the *Floor1* region (gas-a1 or gas-b1) detects a gas concentration
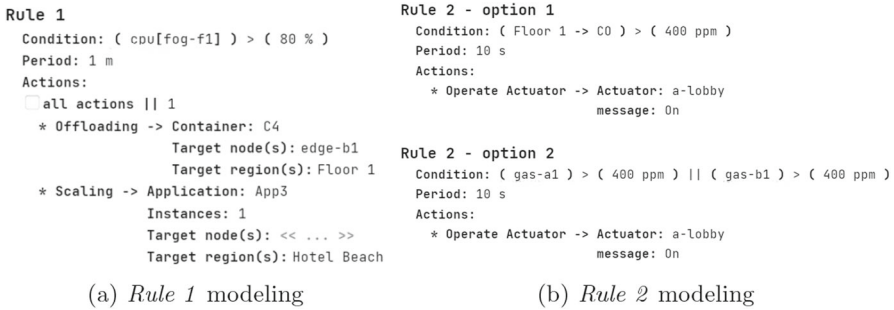
```
Rule 1
  Condition: ( cpu[fog-f1] ) > ( 80 % )
  Period: 1 m
  Actions:
  □ all actions || 1
    * Offloading -> Container: C4
                    Target node(s): edge-b1
                    Target region(s): Floor 1
    * Scaling -> Application: App3
                 Instances: 1
                 Target node(s): << ... >>
                 Target region(s): Hotel Beach
```

(a) *Rule 1* modeling

```
Rule 2 - option 1
  Condition: ( Floor 1 -> CO ) > ( 400 ppm )
  Period: 10 s
  Actions:
    * Operate Actuator -> Actuator: a-lobby
                          message: On

Rule 2 - option 2
  Condition: ( gas-a1 ) > ( 400 ppm ) || ( gas-b1 ) > ( 400 ppm )
  Period: 10 s
  Actions:
    * Operate Actuator -> Actuator: a-lobby
                          message: On
```

(b) *Rule 2* modeling

Fig. 7 Example of rule modeling

greater than 400 *ppm* for 10 seconds. The "On" message is published in the broker topic consumed by the actuator (*a-lobby* alarm). Note that there are two ways to model this rule. While Option 1 involves all CO type sensors on *Floor 1*, Option 2 directly involves both gas sensors.

4 DSL extension: coal underground mining

Our DSL can be used as is to model any type of multi-layered IoT system. However, it has also been designed to be easily extensible to further tailor it to specific types of IoT systems. As an example, we present an extension of our DSL to model underground mining systems as this is a key economic sector in the local region of one of the authors and there is a need for a better modeling of these systems, e.g., for analysis of regulatory compliance.

Indeed, the dynamic and hostile environment of the underground mining industry threatens the operation of IoT systems (e.g. by causing physical damage to the devices) implemented primarily to monitor and ensure the safety of workers. Explosive and toxic gases, risk of geotechnical failure, fire, high temperatures and humidity are some of the risks. Therefore, mining IoT systems must cope with these unexpected changes by self-adapting to guarantee a proper run-time operation.

In addition to system self-adaptations, our DSL should support expressing mining functional requirements. For example, in Colombia, the safety regulation for underground mining works (decree 1886 of 2015) determines limits for the concentration of explosive and toxic gases. If any of these limits is exceeded, a series of actions/adaptations must be performed such as turning on alarms, activating the ventilation system, etc.

To better cope with these scenarios, our extended DSL offers new modeling primitives (see Fig. 8). All concepts that inherit from *Region* represent physical spaces. A *Tunnel* can be *Internal* or *Access*. Each mine access tunnel (*Drift*, *Slope*, or *Shaft*) must have one or more entrances (represented by the *entries* relationship). Finally, checkpoints (areas of the mine where gases, temperature, oxygen, and airflow are monitored) are specified through the *CheckPoint* concept. Each *CheckPoint* could contain multiple *IoTDevices* (sensors or actuators) represented by the *devices* relationship in the metamodel.
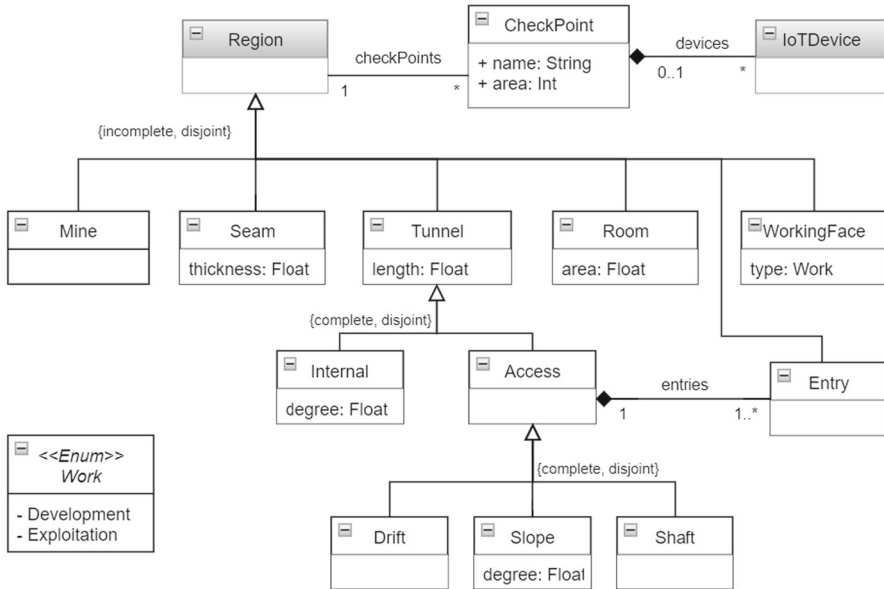
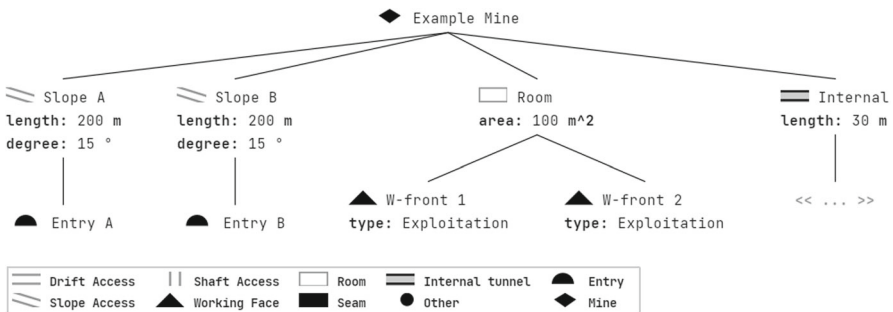**Fig. 8** Excerpt of the DSL extension metamodel

Fig. 9 Mine structure modeling

We offer a tree-based notation for modeling the relevant regions[1] that make up the mine structure. Figure 9 presents an example of the modeling of an underground mine containing two entries (*Entry A* and *Entry B*) in each of its inclined access tunnels (*Slope A* and *Slope B*), an internal tunnel (*Internal*), and a (*Room*) with two exploitation work fronts (*W-front 1* and *W-front 2*).

At each mine control point, the airflow should be controlled by the fans. While very fast air currents can produce fires, very low air currents may not be efficient in dissipating gas concentrations. To involve control points directly in the adaptation

---

[1] Note that our DSL is focused on the structure and rules governing the "behaviour" of the IoT system of the mine, it does not pretend to replace other types of 3D mine mining models.

rules, we have also modified our language. This extension of the DSL enables the modeling of conditions such as $(Control Point A \rightarrow air Flow) > (2m/s)$.

All the other concepts directly reuse the notation of the core DSL presented in Sect. 3.

5 Tool support

In this section, we describe the implementation and tool support for both our DSL and the core runtime framework responsible for monitoring and self-adapting IoT systems at runtime. Once both are described, we discuss the link between the two, i.e. the code generator that takes as input the model specified with our DSL and generates the code for runtime deployment and execution of the system.

5.1 DSL tool support

Our DSL is created using MPS, an open-source language workbench developed by JetBrains. By building the DSL on top of MPS, we automatically get a projectional editor for the DSL with facilities the implementation of the different notations highlighted in the previous sections. The DSL editor is freely available in our repository [20].

In particular, to develop the modeling environment for the DSL, we had to define in MPS three core elements: *structure*, *editors* and *constraints*. The structure is equivalent to the abstract syntax of the DSL. Projection editors define the desired Abstract Syntax Tree (AST) code editing rules. For our DSL We have defined textual, tabular, and tree view editors by implementing the mbeddr[2] extension of MPS.

For example, Fig. 10 shows the definition of the tabular editor for modeling the *Sensor* concept. We have used the *partial table* command to define the table structure (cells, content, and column headers). By defining this editor, the user is enabled to model *Sensors* using a tabular notation as shown in Fig. 4.

Finally, constraints restrict the relationships between nodes as well as the allowed values for properties. We have used this constraint mechanism to embed in the editor several well-formedness rules required in our DSL specification. For instance, we have added constraints to avoid repeated names, constraints to limit the potential values of some numerical attributes, constraints to restrict the potential relationships between nodes, and other constraints that prevent ill-formed models from being built.

5.2 Runtime tool support

Figure 11 summarizes an operational view of our architecture by distinguishing design time (left-hand side) and runtime (right-hand side).

At design time, the user creates an initial IoT system specification model using the modeling editor for the DSL described in the previous section. The code generator, presented in Sect. 5.3, transforms such a specification into a set of deployment and

---

[2] http://mbeddr.com/.

```
tabular editor for concept Sensor
  node cell layout:
    partial table {
      horizontal r<> {
        cell Sensor c<"Device"> r<>
        cell { name } c<"ID"> r<>
        cell ( % type % -> { name } ) c<"Type"> r<>
        cell { unit } c<"Unit"> r<>
        cell { threshold } c<"Threshold"> r<>
        cell
            (/ % regions %                    /) c<"Regions"> r<>
               /empty cell: <default>
        cell { brand } c<"Brand"> r<>
        cell { communication } c<"Communic."> r<>
        cell ( % gateway % -> { name } ) c<"Gateway"> r<>
        cell ( % topic % -> { name } ) c<"Topic"> r<>
        cell { latitude } c<"Latitude"> r<>
        cell { longitude } c<"Longitude"> r<>
      }
    }
```

Fig. 10 Definition of the tabular editor for the *Sensor* concept

configuration options that describe a MAPE-K loop [9] which is performed at runtime. This section covers these components so that we can then describe in the next section how they are generated from the DSL definition.

The MAPE-K loop is a reference model to implement adaptation mechanisms in auto-adaptive systems. This model includes four activities (monitor, analyze, plan, and execute) in an iterative feedback cycle that operate on a knowledge base (see Fig. 11). These four activities produce and exchange knowledge and information to apply adaptations due to changes in the IoT system.

Based on the MAPE-K loop, our runtime architecture is composed of a set of components and technologies to monitor, analyze, plan, and execute adaptations as illustrated in the right-hand side of Fig. 11). We next describe how our architecture particularizes the generic MAPE-K concepts for self-adaptive IoT systems.

5.2.1 Monitor

In the monitoring stage, information about the current state of the IoT system is collected and stored. The collected information is classified into two groups: (1) infrastructure and QoS metrics (presented in Table 1); and (2) information that is published in the system's MQTT broker topics such as temperature, humidity, gas levels, and other types of sensor data. These two kinds of information are aligned with the addressed types of events to be detected, i.e., QoS events and sensor events.
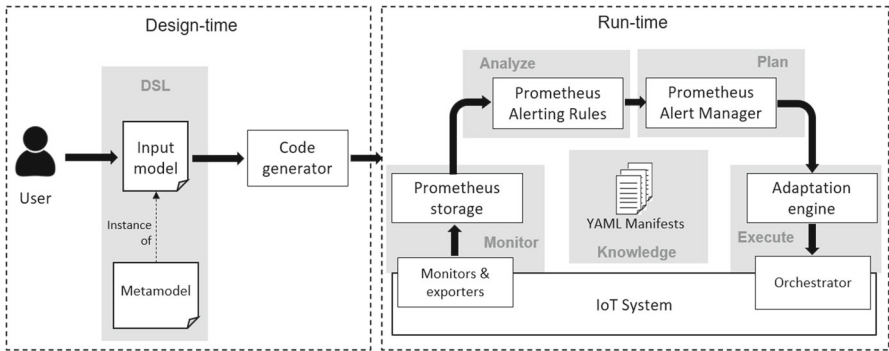
Fig. 11 Overview

Table 1 QoS and Infrastructure metrics

Metric	Exporter	Description
Availability	Kube-state-metrics	Equal to 1 if the component being monitored is available, 0 otherwise
CPU	Node exporter	Number of seconds the CPU has been running in a particular mode
RAM	Node exporter	Available and total Ram memory of the node
Disk usage	Node exporter	Available and total disk space of the node
Bandwidth in	Node exporter	Number of bytes of incoming network traffic to the node
Bandwidth out	Node exporter	Number of bytes of outgoing network traffic from the node

We have implemented Prometheus Storage[3] (a time-series database) to store the information collected by the exporters and monitors (such as *kube-state-metrics*[4] and *node-exporter*[5]). Exporters are deployed to convert existing metrics from third-party apps to Prometheus metrics format. The information collected and stored can be queried in real time through the Prometheus user interface or the Grafana dashboard.

5.2.2 Analyze

The information collected in the monitoring phase must be analyzed, and changes in the system that require adaptations must be identified. To deal with this, we have implemented Prometheus Alerting Rules to define alert conditions based on Prometheus query language expressions (PromQL) and to send notifications about firing alerts to the next MAPE-K loop phase. Each IoT system rule specified through the DSL is transformed into an alert rule of Prometheus.

---

[3] https://prometheus.io/docs/prometheus.

[4] https://github.com/kubernetes/kube-state-metrics.

[5] https://github.com/prometheus/node_exporter.

5.2.3 Plan

According to the analysis made in the previous stage, an adaptation plan is generated with the appropriate actions to adapt the system at runtime. The adaptation plan contains the list of actions (scaling, offloading, redeployment, and operate actuator) that the user has defined for each rule via the DSL. In this stage, Prometheus Alert Manager is used to handle the alerts from the previous stage (Analyze) and routing the adaptation plan to the next stage (Execute). The adaptation plan is sent as an HTTP POST request in JSON format to the configured endpoint (i.e., to the Adaptation Engine).

5.2.4 Execute

In the Execute stage, the actions are applied to the IoT system following the actions defined in the adaptation plan. To achieve this, we have built an Adaptation Engine, an application developed using Python, flask, and the python API to manage the Kubernetes orchestrator. The adaptation engine can apply two sets of actions: (1) architecture adaptations through the orchestrator (e.g., autoscaling an application or offloading a pod); and (2) regular system operations such as controling a system actuator by sending it an instruction (e.g., a message to turn on or off an alarm). This adaption is generic and can be used to run any IoT system modeled with our DSL, including its mining extension.

5.2.5 Putting it all together: example scenario

To better understand how the different elements cooperate, we will exemplify how the framework monitors and executes tge rule specified in Fig. 7b. Code for the rule management is automatically generated (Sect. 5.3), including YAML[6] manifests for deployment, configuration and execution of the monitors, exporters, Prometheus, the Adaptation Engine and other software components implemented in the MAPE-K loop.

In the Monitoring stage, the exporters gather information about CPU consumption of the *fog-f1* node. This information is stored in the Prometheus database. Then, in the Analysis stage, the condition of the rule is verified by executing query expressions in PromQL language. For example, the expression (executed by Prometheus Alerting Rules) that checks if the CPU consumption of the fog-f1 node exceeds 80% for 1 minute is presented in listing. 1 Note that we are calculating the average amount of CPU time used excluding the idle time of the node. If the condition is true, the alert signal is sent to the Alert Manager component of the next stage of the cycle (Plan). When the alert is received, the adaptation plan is built containing the two actions (offloading and scaling) and their corresponding information such as container to be offloaded, application to be scaled, number of instances, target nodes and target regions. In the Execute stage, the Adaptation Engine component first performs the Offloading action, and only if it fails, then the second action (Scaling) is performed.

```
1    - alert: ram-consumption
```

---

[6] YAML is a data serialization language typically used in the design of configuration files.

```
2       expr: 100 - (avg by(node_hostname) (rate(
    node_cpu_seconds_total{mode=
3            ``idle'',node_hostname=``fog-f1''}[15s])) * 100)
      > 80
4       for: 1m
```

Listing 1 Query expression to check CPU consumption of fog1-f1 node

5.3 Code generator

To configure and run the runtime infrastructure of an IoT system from its DSL model, we have implemented a model-to-text transformation that generates YAML files to deploy the IoT system's container-based applications and the components of each stage of our MAPE-K loop based framework, including its internal logic. In particular, the generated code includes the following components:

- The container-based IoT applications specified in the input model. Following the running example of Sect. 2, the YAML manifests for deployment of containers C1, C2, C3, C4, and C5 are generated
- YAML Manifests to deploy the monitoring tools and exporters such as kube-state-metrics, node-exporter, and mqtt-exporter
- YAML code to deploy and configure the Prometheus Storage, Prometheus Alerting Rules, and Prometheus Alert Manager components. The PromQL code to define the rules (e.g., the code shown in Listing 1) is also generated as a Prometheus configuration file
- YAML manifest to deploy the Adaptation Engine
- and the Grafana application to display the monitored data stored in the Prometheus database.

Due to space limitations, we do not show here the model-to-text transformation. You can find all this information in the project repository [20], including the generated code for the running example from Sect. 2.

6 Empirical evaluation

We have designed and conducted three experiments to validate our DSL and its accompanying infrastructure.

6.1 Language validation

We conducted two experiments to validate the expressiveness and ease of use of our DSL: *Experiment 1*, focused on specific mining concepts, and *Experiment 2*, focused on core architectural concepts, both based on the basic methodology for conducting usability studies [21]. Both also cover the modeling of adaptation rules.

We report here the results of *Experiment 1*. Full details of *Experiment 2* are available on the project repository[7]).

6.1.1 Experiment design and setup

We designed the experiment to validate the expressiveness and usability of the DSL regarding the modeling of the mine structure, the control points, sensors and actuators, and rules to manipulate such actuators (e.g., turn on the mine ventilation system when the methane gas sensor exceeds the threshold value).

Eight subjects participated in the experiment. Participants were experts from the mining domain, but had not been exposed to our DSL before. The goal was to check whether they were able to use it and get their feedback on the experience.

The experiment consisted of an asynchronous screening test (pre-questionnaire) to assess subjects' prior knowledge and suitability for participation, and a synchronous exercise (virtual meeting) with two parts (Sessions 1 and 2). The materials and exercises provided to the participants, the questionnaires and the anonymized answers can be found in the repository of our DSL extended to the mining industry domain.[7]

- Pre-questionnaire (10 min): this screening questionnaire (Q0) was conducted prior to the start of Sessions 1 and 2 to ensure that participants had a basic level of mining knowledge including the structure of underground coal mines, gas monitoring systems, and modeling tools in this domain.
- Session 1 (50 min): In the first 20 minutes, we introduced basic knowledge of IoT systems and the use of the DSL implemented in MPS to model the structure of underground mines, the control points and the IoT devices deployed (sensors and actuators). Next, the participants performed the first modeling exercise about an underground coal mine (with the structure shown in Fig. 9), two control points (one at each working face) with three gas sensors and an alarm, a fan, and a control door in the internal tunnel. Each participant was provided with a virtual machine configured with the necessary software to perform the modeling exercise. Finally, the participants filled out a questionnaire (Q1) about the usability and expressiveness of the DSL to model the concepts of the first exercise.
- Session 2 (40 min): In Session 2, we first introduced the basic concepts of self-adaptive systems and the design of rules using our DSL. Next, participants performed the second exercise: modeling three rules involving sensor data and actuator operation. For example, if any of the methane gas sensors throughout the mine exceed the threshold value for 5 seconds, then turn on the fan and activate the alarms. Finally, participants completed the questionnaire Q2 to report their experience modeling the rules. Q2 also contained open-ended questions to obtain feedback on the use of the entire tool and suggestions for improvement.

The experiment was conducted in Spanish on three different dates in 2022. The first author of the paper conducted the virtual meetings and ensured that all were equally well executed.

6.1.2 Results

Four of the participants were involved in education (either students, teachers, or researchers), while the remaining four were involved in industry. All of them are

---

[7] https://github.com/SOM-Research/IoT-Mining-DSL.

(a) DSL ease of use

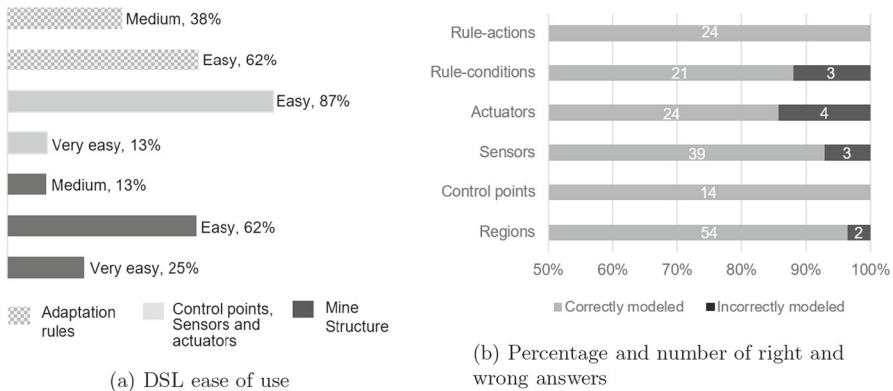(b) Percentage and number of right and wrong answers

Fig. 12 Validation results

aware of the terminology used in the design and structure of underground coal mines. Only two participants were not familiar with cyber-physical or IoT systems for mining. The modeling tools in mining context that they have used are AutoCAD[8] and Minesight[9] for the graphical design of the mine structure, and VentSim[10] for ventilation system simulations. None of them were familiar with MPS.

Figure 12a presents the responses from questionnaires Q1 and Q2 related to the ease of use of the DSL. Most of them reported that the modeling of the mine structure, the control points, the devices (sensors and actuators), and the rules were easy. The results are positive and can also be evidenced by the number of right and wrong concepts modeled by the participants (Fig. 12b). The number of errors were low (12 of 188 modeled concepts): three incorrect *Rule-conditions* by wrong selection of the unit of measure, four incorrect *Actuators* by wrong assignment of actuator type and location within the mine, three missing *Sensors* not modeled, and two incorrect *Regions* (working faces) whose type was not selected.

Through the open-ended questions in questionnaires Q1 and Q2, participants suggested the following improvements to the DSL.

- Include the specification of the coordinates for each region and control points of the mine. Additionally, it would be useful to specify the connection between internal tunnels.
- The condition of a rule has a single time period. However, it would be useful to associate two time periods for conditions composed of two expressions. For example, the condition $tempSensorA > 30C(10seconds) \&\& tempSensorB > 35C(20seconds)$.
- The mine ventilation system can be activated periodically at the same time each day. It would be useful if the DSL could model rules whose condition is associated with the time of day.

---

[8] https://www.autodesk.com/products/autocad.

[9] https://its.mines.edu/software-title/minesight/.

[10] https://ventsim.com.

6.1.3 Threats to validity

Although validation problems in empirical experiments are always possible, we have looked for methods to ensure the quality of the results, analyzing two types of threats: internal and external.

Internal validation concerns factors that could affect the results of the evaluation. To avoid defects in the planning of the experiments and the questionnaires (protocol), all authors of the paper discussed the protocol including the modeling exercises, the dependent variables, and the questions of the questionnaires. In addition, a senior researcher (not in the authors list) in empirical experiments validated the questionnaires. Another common thread is related to the low number of samples to successfully reveal a pattern in the results. Thirteen users total participated in this empirical validation. Eight participants were involved in Experiment 1, and five different ones in Experiment 2. Although the size of the participant group for this type of validation continues to be a matter of discussion, studies suggest 3 to 10 participants (depending on the level of complexity) are sufficient. For example, a popular guideline in this area is given by Nielsen and Landauer [22], who suggest that five participants are likely to discover 80% of the problems.

External validity addresses threats related to the ability to generalize results to other environments. For example, to validate the population and avoid sampling bias, we conducted a pre-questionnaire to the participants to ensure that they had the necessary basic knowledge and that there was no substantial difference between participants. In addition, at the beginning of each session, we introduced the definition of the concepts required for the experiment. It is important to emphasize that the participants of Experiment 1 were related to the topics of the mining domain, while those of Experiment 2 were computer science researchers.

6.2 Runtime framework validation

To evaluate the self-adaptation capability of our approach and the correctness of the code-generation and runtime infrastructure, we conducted experiments to test the architectural adaptations (scaling, offloading, and redeployment).

For each adaptation assessment we have designed a simple scenario in which an IoT system faces an event that forces adaptations. We have validated that starting from the IoT model including a specified rule triggered by such event, the code-generator creates the deployment infrastructure to run the IoT system and assess that such system actually adapts as expected when the event is triggered. We have collected and analyzed metrics such as CPU consumption, node availability, and time spent in each stage of the MAPE-K loop to also show the benefits of such self-adaptive architecture when the system is under pressure.

The design of these experiments and the results are in the appendices of the paper that can be found in the project repository. [11]

---

[11] https://github.com/SOM-Research/selfadaptive-IoT-DSL/blob/main/docs/paper-appe.pdf.

7 Related work

Modeling of cloud architectures has been widely studied, including provisioning of resources for cloud applications. Nevertheless, most proposals do not cover multi-layered architectures involving fog and edge nodes. This is also the case for Infrastructure-as-code (IaC) tools.

Some works such as [23–25] focus on the cloud layer. Others, like [5–7] are oriented towards the edge and fog layers. In particular, studies such as [5, 6] focus on the modeling of sensors, actuators, and software functionalities, while [7] addresses the modeling of application deployment at the fog layer. Works like [26, 27] are oriented towards lower-level communication aspects and individual IoT component behaviours.

A few proposals are closer to ours in terms of the static modeling of IoT systems, such as [28, 29] but using less expressive DSLs and, especially, limited possibilities when it comes to the definition of rules, including self-adaptive ones.

Indeed, support for (self-adaptive) rules is less covered by previous approaches. For instance, rules for cloud architectures are the topic of works such as [30–33] that propose only partial solutions, as they either restrict the parts of the system that could be adapted or offer a limited expressiveness in the definition of the rules (e.g., no trigger condition).

Among the most powerful solutions, Lee et al. [4] present a self-adaptive framework where IoT systems are modeled as finite-state machine. However adaptations are only enabled at the device layer level and self-adaptive rules are not possible. Garlan et al. [34] present Rainbow, a framework for adapting a software system when a constraint (expressed in terms of performance, cost, or reliability) is violated. Similarly, Weyns et al. [35] propose MARTA, an architecture-based adaptation approach to automate the management of IoT systems employing runtime models. However, neither Rainbow or MARTA cover the specification of multi-layer architectures, nor rules combining them, and their rules are coarse-grained (e.g. not at the container level). Finally, Petrovic et al. [29] propose SMADA-Fog, a semantic model-driven approach to deployment and adaptation of container-based applications in Fog computing scenarios. SMADA-Fog does not allow the specification of complex adaptation rules composed of various conditions and actions. Moreover, grouping nodes and IoT devices according to their location is not possible, forbidding the possibility to apply adaptations on group of nodes belonging to a cluster or a given region.

Concerning the specification of IoT systems in underground mining domain, works such as [36–38] focus on the design or deployment of the device layer (sensors and actuators) of monitoring systems, but do not address the mine structure specification, runtime adaptations, and deployment of containerized applications. To sum up, ours is the first proposal that enables the specification, deployment and execution of multi-layer IoT architectures (device, edge, fog, and cloud) and the definition of complex rules covering all layers (and combinations of) involving multiple conditions and actions that can, potentially, engage groups of nodes in the same region or cluster of the IoT system. Moreover, our proposal can be easily extended and specialized in different domains as shown with the underground mining scenario.

8 Conclusions and future work

We have presented a model-based approach for the specification and runtime execution of multi-layered architectures of IoT systems and their self-adaptation rules. Our approach comprises a new DSL to model such systems, a code generator, and a runtime infrastructure, based on the MAPE-K loop, to monitor and execute the IoT system at runtime based on a variety of rules, involving architectural adaptations and rules to address functional requirements. The full process is assisted by a set of open-source tools that have been released as part of this work. We have also validated the usability and extensibility of the DSL.

As part of our future work, we will continue to enrich the DSL based on the suggestions made by the experiment participants, including a new visual renderer of the modeled architecture to complement the current projections. We also plan to facilitate the definition of complex self-adaptive rules by predefining a set of common patterns such as *canary*, *rolling update*, and *blue-green* deployment strategies that could be directly referenced in the definition of a rule. Moreover, we are also interested to automatically discover potentially useful adaptation rules by analyzing historical log data from the IoT system (e.g., focusing on previous system crashes) with machine learning. Finally, we are creating additional extensions to the DSL. In particular, one to model Wastewater Treatment Plants as part of an ongoing European project.

References

1. Jiang Y, Huang Z, Tsang DH (2017) Challenges and solutions in fog computing orchestration. IEEE Netw 32(3):122–129
2. Rhayem A, Mhiri MBA, Gargouri F (2020) Semantic web technologies for the internet of things: systematic literature review. Internet Things 11:100206
3. Brambilla M, Cabot J, Wimmer M (2017) Model-driven software engineering in practice. synthesis lectures on software engineering, 2nd edn. Morgan & Claypool Publishers, USA
4. Lee E, Seo Y-D, Kim Y-G (2019) Self-adaptive framework based on mape loop for internet of things. Sensors 19(13):2996
5. Patel P, Cassou D (2015) Enabling high-level application development for the internet of things. J Syst Softw 103:62–84

6. Ciccozzi F, Spalazzese R (2016) Mde4iot: supporting the internet of things with model-driven engineering. In: Int Symposium on Intelligent and Distributed Computing, pp. 67–76

7. Yigitoglu E, Mohamed M, Liu L, Ludwig H (2017) Foggy: a framework for continuous automated IoT application deployment in fog computing. In: IEEE Int. Conf. on AI & Mobile Services, pp. 38–45

8. Alfonso I, Garcés K, Castro H, Cabot J (2021) Modeling self-adaptative IoT architectures. In: 2021 ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems Companion, pp. 761–766

9. Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36(1):41–50

10. Latifah A, Supangkat SH, Ramelan A (2020) Smart building: A literature review. In: Int Conf on ICT for Smart Society (ICISS), pp. 1–6

11. Alfonso I, Garcés K, Castro H, Cabot J (2021) Self-adaptive architectures in IoT systems: a systematic literature review. J Internet Serv Appl 12(1):1–28

12. Al-Qamash A, Soliman I, Abulibdeh R, Saleh M (2018) Cloud, fog, and edge computing: A software engineering perspective. In: 2018 Int Conf on Computer and Applications (ICCA), pp. 276–284 . IEEE

13. Shi W, Cao J, Zhang Q, Li Y, Xu L (2016) Edge computing: Vision and challenges. IEEE Internet Things J 3(5):637–646

14. Dustdar S, Avasalcai C, Murturi I (2019) Edge and fog computing: Vision and research challenges. In: 2019 IEEE Int. Conf. on Service-Oriented System Engineering (SOSE), pp. 96–9609 . IEEE

15. Yousefpour A, Fung C, Nguyen T, Kadiyala K, Jalali F, Niakanlahiji A, Kong J, Jue JP (2019) All one needs to know about fog computing and related edge computing paradigms: a complete survey. J Syst Architect 98:289–330

16. Mansouri Y, Babar MA (2021) A review of edge computing: features and resource virtualization. J Parallel Distrib Comput 150:155–183

17. Gómez A, Iglesias-Urkia M, Belategi L, Mendialdua X, Cabot J (2021) Model-driven development of asynchronous message-driven architectures with asyncapi. Softw Syst Model 21(4):1583–1611

18. Mishra B, Kertesz A (2020) The use of mqtt in m2m and iot systems: a survey. IEEE Access 8:201071–201086

19. Berger T, Völter M, Jensen HP, Dangprasert T, Siegmund J (2016) Efficiency of projectional editing: A controlled experiment. In: Proc. of the 24th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering, pp. 763–774

20. Alfonso I, Garcés K, Castro H, Cabot J (2022) Self-adaptive IoT DSL. https://github.com/SOM-Research/selfadaptive-IoT-DSL

21. Rubin J, Chisnell D (2008) Handbook of usability testing: how to plan design and conduct effective tests. John Wiley & Sons, New Jersey

22. Nielsen J, Landauer TK (1993) A mathematical model of the finding of usability problems. In: Proc. of the INTERACT'93 and CHI'93 Conf. on Human Factors in Computing Systems, pp. 206–213

23. Sandobalin J, Insfran E, Abrahão S (2019) ARGON: A model-driven infrastructure provisioning tool. In: ACM/IEEE 22nd Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 738–742

24. Sledziewski K, Bordbar B, Anane R (2010) A DSL-based approach to software development and deployment on cloud. In: 24th IEEE Int. Conf. on Advanced Information Networking and Applications, pp. 414–421

25. Bergmayr A, Breitenbücher U, Kopp O, Wimmer M, Kappel G, Leymann F (2016) From architecture modeling to application provisioning for the cloud by combining uml and tosca. In: 6th Int. Conf. on Cloud Computing and Services Science, pp. 97–108

26. Gomes T, Lopes P, Alves J, Mestre P, Cabral J, Monteiro JL, Tavares A (2017) A modeling domain-specific language for IoT-enabled operating systems. In: IECON 2017-43rd Annual Conf. of the IEEE Industrial Electronics Society, pp. 3945–3950

27. Eterovic T, Kaljic E, Donko D, Salihbegovic A, Ribic S (2015) An internet of things visual domain specific modeling language based on UML. In: 2015 XXV Int. Conf. on Information, Communication and Automation Technologies (ICAT), pp. 1–5

28. Barriga JA, Clemente PJ, Sosa-Sánchez E, Prieto ÁE (2021) Simulateiot: domain specific language to design, code generation and execute iot simulation environments. IEEE Access 9:92531–92552

29. Petrovic N, Tosic M (2020) Smada-fog: semantic model driven approach to deployment and adaptivity in fog computing. Simul Model Practice Theor 101:102033

30. Ferry N, Chauvel F, Song H, Rossini A, Lushpenko M, Solberg A (2018) Cloudmf: model-driven management of multi-cloud applications. ACM Trans Internet Technol (TOIT) 18(2):1–24

31. Chen W, Liang C, Wan Y, Gao C, Wu G, Wei J, Huang T (2016) MORE: A model-driven operation service for cloud-based it systems. In: IEEE Int. Conf. on Services Computing, pp. 633–640
32. Erbel J, Korte F, Grabowski J (2018) Comparison and runtime adaptation of cloud application topologies based on OCCI. In: The 8th Int. Conf. on Cloud Computing and Services Science, pp. 517–525
33. Cámara J, Muccini H, Vaidhyanathan K (2020) Quantitative verification-aided machine learning: A tandem approach for architecting self-adaptive IoT systems. In: 2020 IEEE Int. Conf. on Software Architecture (ICSA), pp. 11–22
34. Garlan D, Schmerl B, Cheng S-W (2009) Software architecture-based self-adaptation. In: Autonomic Computing and Networking, pp. 31–55
35. Weyns D, Iftikhar MU, Hughes D, Matthys N (2018) Applying architecture-based adaptation to automate the management of internet-of-things. In: European Conf. on Software Architecture, pp. 49–67
36. Porselvi, T., Ganesh, S., Janaki, B., Priyadarshini, K., et al.: IoT based coal mine safety and health monitoring system using lorawan. In: 2021 3rd Int. Conf. on Signal Processing and Communication, pp. 49–53 (2021)
37. Mishra P, Kumar S, Kumar M, Kumar J et al (2019) IoT based multimode sensing platform for underground coal mines. Wirel Personal Commun 108(2):1227–1242
38. Alfonso, I., Goméz, C., Garcés, K., Chavarriaga, J.: Lifetime optimization of wireless sensor networks for gas monitoring in underground coal mining. In: 7th Int. Conf. on Computers Comms. and Control, pp. 224–230 (2018)