Modeling and Provisioning IoT Cloud Systems for Testing Uncertainties

Hong-Linh Truong, Luca Berardinelli, Ivan Pavkovic, Georgiana Copil
Distributed Systems Group, TU Wien
{truong,berardinelli,ivan.pavkovic,e.copil}@dsg.tuwien.ac.at

ABSTRACT

Modern Cyber-Physical Systems (CPS) and Internet of Things (IoT) systems consist of both loosely and tightly interactions among various resources in IoT networks, edge servers and cloud data centers. These elements are being built atop virtualization layers and deployed in both edge and cloud infrastructures. They also deal with a lot of data through the interconnection of different types of networks and services. Therefore, several new types of uncertainties are emerging, such as data, actuation, and elasticity uncertainties. This triggers several challenges for testing uncertainty in such systems. However, there is a lack of novel ways to model and prepare the right infrastructural elements covering requirements for testing emerging uncertainties. In this paper, first we present techniques for modeling CPS/IoT Systems and their uncertainties to be tested. Second, we introduce techniques for determining and generating deployment configuration for testing in different IoT and cloud infrastructures. We illustrate our work with a real-world use case for monitoring and analysis of Base Transceiver Stations.

CCS CONCEPTS

• **Computing methodologies → Model development and analysis**; • **Computer systems organization → Embedded and cyber-physical systems**; • **Software and its engineering → Software verification and validation**; *Distributed systems organizing principles*; *Software development techniques*;

KEYWORDS

Testing, IoT, Cloud computing, Cyber-Physical Systems, Uncertainty, Deployment

1 INTRODUCTION

Recent advances in the integration between Internet of Things (IoT), edge infrastructures (including so-called fog computing infrastructures) [13], and cloud services have fostered the development of several types of Cyber-Physical Systems (CPS) and IoT systems. The design, development and operation of such systems are extremely challenging [29, 32] due to the complexity of IoT elements and software services and their connectivity. Our research focuses on a particular challenge in supporting the design and operation of IoT, edge and cloud infrastructures that will be used by different applications, e.g., Geosports, smart cities, and predictive maintenance of equipment. In such applications, software components are deployed to exploit existing software and hardware elements of CPS/IoT systems; these elements can be available in public and private Cloud and IoT infrastructures, such as Google, Amazon, Azure, and FIWARE (https://www.fiware.org/). During the execution, applications would need to add new elements (e.g., due to the elasticity of workload) as well as to be reconfigured (e.g., due to the adaptation). Since our CPS/IoT systems consist of mainly IoT devices, edge systems, and cloud-based services in data centers, in this paper we denote them as *IoT Cloud Systems*. The term of *CPS* and *IoT Cloud Systems* are interchangeable in our work.

1.1 Motivation

Our motivation is from the need to deploy and test infrastructures for CPS/IoT systems that include IoT elements at the edge and cloud services in data centers. Techniques for these tasks are on high demand by researchers and developers of modern CPS/IoT systems. Furthermore, since such systems span various IoT, edge and cloud infrastructures operated by different providers, there are many types of uncertainties, ranging from known sources of device characteristics [8] to emerging IoT data and elasticity ones [2], that must be tested [3]. In our work, *uncertainty* is considered as the lack of certainty (i.e., knowledge) about the timing and nature of input data and request, the state of a system, a future outcome, as well as other relevant factors [2, 3].

In testing uncertainties of IoT Cloud Systems, however, there is a lack of tools to enable the developers to easily design such IoT Cloud Systems under test (SUT). Although it is challenging to deploy IoT and Cloud resources, various tools have been developed, such as SALSA [20]. However, with these tools, the main problem is that the developers are required to have (fixed) existing (virtual) infrastructures and they have to specify precisely the description of infrastructures, such as using TOSCA and HOT [12], for deployment. It is not flexible, and changing the infrastructure for SUT is a difficult and error-prone task. Furthermore, such tools are separated from the test purpose, as they focus on deployment of systems in general. Software engineering tools allow to model various IoT and Cloud elements [7, 14] but they lack the integration with deployment tools. Most of them lack the incorporation of uncertainty objectives (e.g., for testing). Other tools for testing are able to generate test cases but they lack features for determining test configuration based on uncertainty models.

Our work is to streamline the way how the developers would test their IoT Cloud Systems through conventional software design and development: specify systems and test configuration, uncertainty at

MobiQuitous 2017, November 7–10, 2017, Melbourne, VIC, Australia

ACM ISBN 978-1-4503-5368-7/17/11…$15.00
https://doi.org/10.1145/3144457.3144490

the high-level, e.g., in terms of UML models, and then automatically generate the infrastructure configurations and deploy the infrastructure. This requires not only an integration between various tools from different works but also requires new features to glue design time modeling with runtime deployment and configuration.

1.2 Contributions

We aim at simplifying the preparation and deployment of IoT Cloud Systems for uncertainty testing. Our approach is to enable the developer to specify SUT at a high-level view, together with potential uncertainties that the developer wants to test and test configuration (such as, preferred infrastructure providers and cost). From the specification, we search existing artifacts for SUT and establish the SUT by generating SUT configurations and deploying suitable artifacts to create an IoT Cloud Systems as a SUT. We contribute:

- A novel profile for specifying IoT Cloud Systems elements together with test configuration and uncertainties: this goes beyond separate work on, e.g., just modeling IoT or Cloud, or just focusing on uncertainties.
- Configuration generation and deployment of IoT Cloud Systems: this goes beyond existing tools by creating suitable deployments for different underlying systems.

Our contributions are built atop known technologies that allow us to experiment uncertainty testing in different IoT/Cloud settings. In this paper, we run a real-world example with an IoT Cloud System for monitoring and predictive maintenance of Base Transceiver Stations (BTSs) that is based on IoT and Cloud resources and services. In our prototype, our profiles are integrated with and supported by standard UML tools, while our SUT generation and deployment are for well-established Cloud and IoT services models.

Paper Structure: The rest of this paper is organized as follows: Section 2 describes the overall approach and tool architecture. Section 3 describes our Infrastructure, Uncertainty, and Testing modeling. We present techniques for creating SUT configuration in Section 4. Prototype and experiments are shown in Section 5. We discuss the related work in Section 6, before summarizing the paper and presenting the future work in Section 7.

2 TOOLING PIPELINE FOR UNCERTAINTY TESTING

In developing IoT Cloud systems and applications, various software artifacts are developed by different stakeholders and made available as service instances or deployable images in software repository (e.g., Docker images, virtual machine images, or executable software package). The development of such software might or might not be part of the software (uncertainty) testing. In our work we distinguish the following types of artifacts:

- *public artifacts about elements of IoT Cloud Systems from existing providers* – denoted as P_a: include, e.g., virtual machine (VM) images, Docker container images, VM/container images with middleware and application components, and executable software programs. These artifacts are generally developed and stored in public and private IoT and Cloud systems and their images are available for downloading and deploying.
- *customized artifacts of elements in system under test (SUT)* – denoted as SUT_a: are part of SUT. They may or may not be based on P_a but they are specific for SUT. For example, one can design a software sensor within a Docker container as an SUT_a, which will be executed atop a virtual machine as a P_a.
- *test utilities* – denoted as TU_a: they are utilities designed specially for testing purposes. They are part of the test infrastructure (not necessary part of the system under test). Given the abovementioned example, a log collector inside the container of SUT_a can be considered as a test utility as it helps to collect logs within the container for testing.

Figure 1(a) describes the relationships among artifacts and their deployment for testing. Generally, these artifacts have APIs for deploying and executing them; such APIs are understood by the developer and they can be modeled or specified by the developer. For example, most P_a can be instantiated or used through tools provided by corresponding providers, while SUT_a are usually associated with specific operations/API so that one can configure and invoke them for testing. Consider the complexity from IoT Cloud systems development and provisioning, we cannot simply make the assumption that a single tool would be able to deal with all steps in modeling, provisioning and testing. We therefore leverage the pipeline concept to create and employ a set of tools for provisioning SUT. First, the developer can follow certain engineering workflows to provide different views of the SUT via textual and/or graphical models [9] by using some tools. Structural and behavioral information are further enriched with uncertainties to suitably tailor the provisioning of the SUT. Based on that, we can extract various information from design models and test configuration. From such extracted information, one can decide how to select suitable elements and test utilities to perform the test. This requires a complex set of tools as there are several steps spanning from model engineering to runtime provisioning and configuration. In particular, depending on the test configuration and the availability of resource providers, we might need to have different provisioning strategies. Figure 1 (b) presents the testing process and tooling pipeline for provisioning infrastructures:

- **Modeling**: describes IoT Cloud Systems under test, potential uncertainty and other behavior information. We rely on UML profiles for modeling.
- **Extracting model information**: extracts various types of information specified in the model and makes the information available for other tools.
- **Generating Test Configuration:** creates different test configurations based on uncertainties, the availability of resources and other parameters. Each configuration will be associated with appropriate deployment configuration for SUT.
- **Deploying SUT**: performs the deployment of SUT and test utilities.

One of the key points of our framework is the extensibility. In the modeling phase, the developer can specify various information and we can extract them. However, when generating test configurations, we need to consider various parameters, related to the cost and other underlying IoT and cloud infrastructures. Therefore, we expect to have several ways, e.g., implemented through plug-ins, to generate suitable configurations and deployment. Our design enables this,

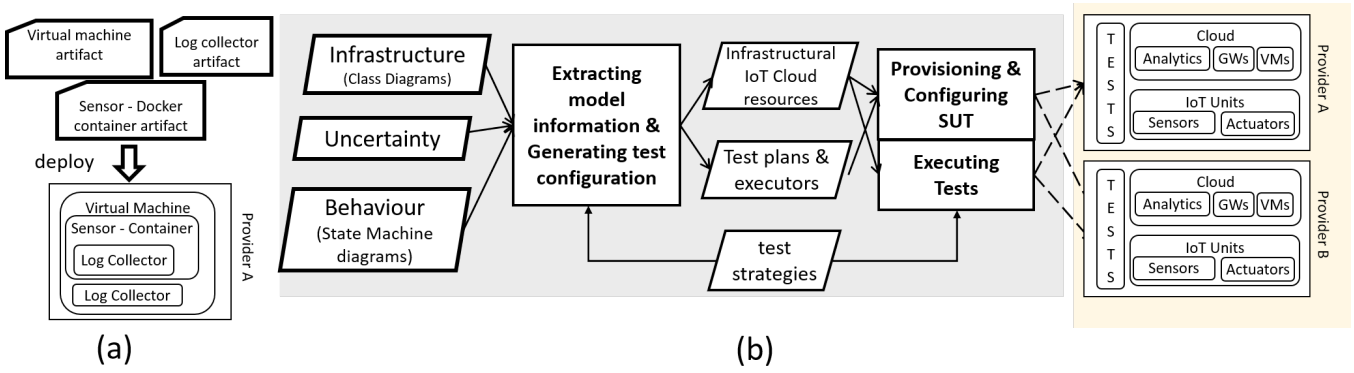

Figure 1: (a) Artifacts and their deployment, (ii) Tool pipeline for dynamic SUT provisioning

but in this paper we will focus on a specific implementation of our configuration and deployment generation.

3 MODELING IOT CLOUD INFRASTRUCTURES AND UNCERTAINTY FOR TESTING

One of the first step is to model IoT cloud systems under test (SUT), their configuration and uncertainties. This requires us to deal with a great challenge in terms of conceptualizing uncertainties and modeling various integrated aspects. Although models for IoT and clouds are quite popular, putting uncertainties, IoT, cloud, etc., into a same SUT has not been addressed. Although there are works that separately deal with IoT modeling [10] and cloud based systems [1] as part of a model-driven engineering (MDE) process, a combination of modeling and testing activities with uncertainty as an explicit, cross-cutting concern throughout the process, is missing. This is the key, multifaceted issue we are tackling in this paper.

We leverage model-driven engineering (MDE) approach to tackle the complexity of SUT to guarantee the correct design, implementation and operation of IoT Cloud Systems under uncertainty. For this purpose, we capture infrastructures as SUT in Unified Modeling Language (UML) [26] models. UML is a standard, general purpose modeling language that is currently the most used by practitioners in companies [15, 22].

3.1 Modeling IoT Cloud Systems under Test

Our goal is to extend UML with a minimal set of concepts to represent the underlying *Infrastructure* and the constituting *InfrastructureElement*s of the cloud-based cyber-physical systems. Shown in Figure 2, our an envisaged IoT Cloud System, represented by *Infrastructure*, includes both software and hardware *Unit*s. We aim at representing both kinds of units in a specular manner to provide the same modeling expressiveness, with the only exception of communication devices and protocols, as detailed later. Therefore, we apply the following design rules in the definition of the Infrastructure profile:

(1) We introduce a generic, abstract stereotype for concept cross-cutting software and platform representations.
(2) We add two specialized, concrete stereotypes adding the Virtual- and Physical- prefixes to the name of the generic, abstract stereotype if it helps disambiguating terms applicable to both software and hardware domains.
(3) We add common properties within to generic abstract stereotypes.

The *Infrastructure* is composed of multiple, generic *Unit*s, each one with its own identifier, location, description, and configuration properties. In particular, a configuration represents the settings associated to the *Unit*. *Unit*s are distinguished in *PhysicalUnit*s and *VirtualUnit*s that represent hardware and software resources, respectively. Both physical and virtual units are complex elements and can be composed of other physical and virtual units, respectively. A *PhysicalUnit* has associated *Actuator*s and *Sensor*s , which, in turn, are themselves particular kinds of *PhysicalUnit*s.

An *Actuator* represents a hardware component that changes the status of the surrounding environment. Each *Actuator* realizes one or more *PhysicalCapabilities*. A *Sensor* is a component through which a *PhysicalUnit* monitors its environment (e.g., temperature sensor, humidity sensor). Each *PhysicalUnit* has associated *Metric*s that it is capable to collect. For example, a thermostat physical unit can include both a sensor to collect temperature and humidity (i.e., the physical capability to collect two metrics), and an actuator that has the capabilities to modify temperature and humidity of the surrounding environment. A particular kind of *PhysicalUnits* are *IODevices* that can be found in CPS like gateway, router, switch, hub, and protocol converter. For this purpose, an IODeviceType enumeration type is defined with a distinct EnumerationLiteral for each of them.

Each *PhysicalUnit* has associated one or more *VirtualUnit*s that run on top of it (e.g., PLC code running and governing machines within a production system). As anticipated, we assume a specular set of concepts to describe the software architecture of the *Infrastructure*. A *VirtualUnit* has associated *VirtualActuator*s and *VirtualSensor*s, which, in turn, are themselves particular kinds of *VirtualUnit*s. A *VirtualActuator* represents the software component through which the owning *VirtualUnit* controls the hardware platform elements that interact with the environment. Each *VirtualActuator* realizes one or more *VirtualCapabilities* A *VirtualSensor* is a software component through which physical sensors are controlled.

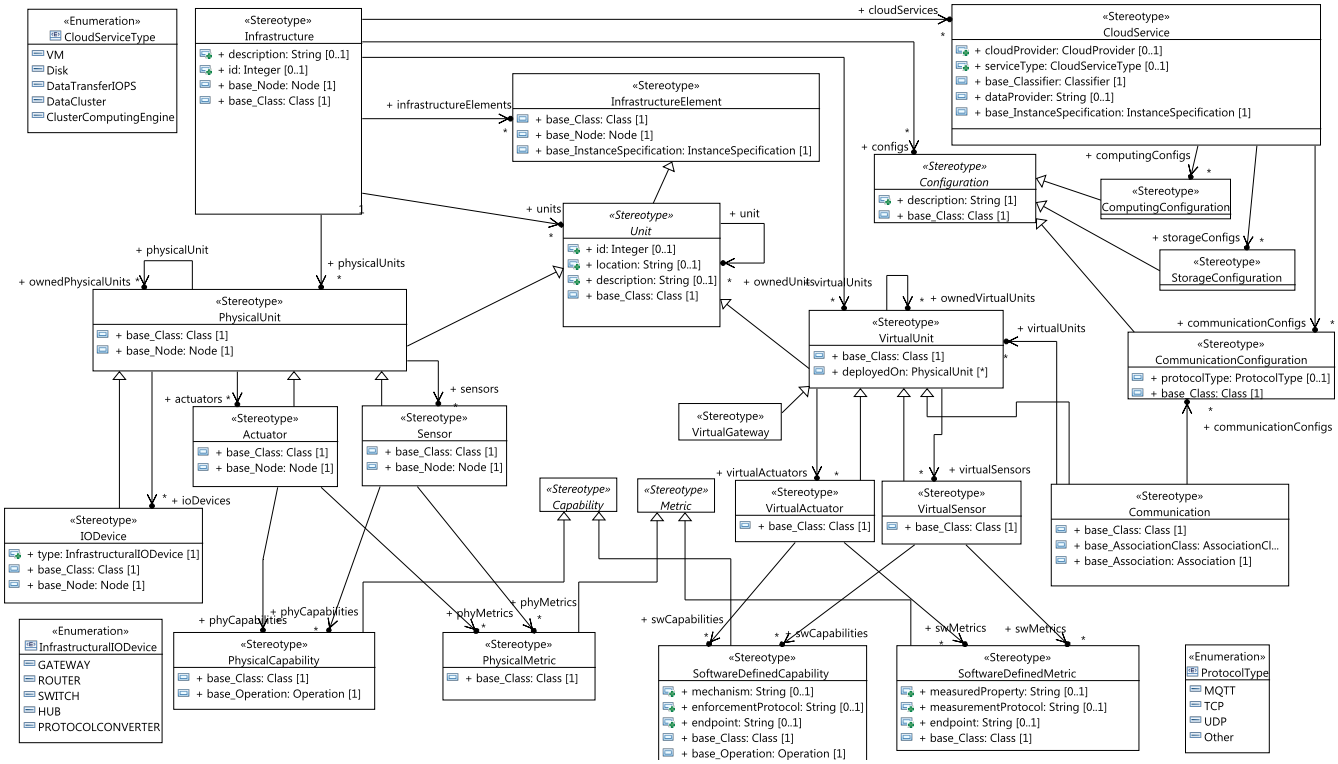

Figure 2: Infrastructure profile: excerpt of stereotypes and their relationships.

Each *VirtualUnit* has associated one or more *SoftwareDefinedMetrics* that it is capable to collect and measure, e.g., temperatures, defined dynamically. The *SoftwareDefinedMetric* has an id, name, description, endpoint, period, measuredProperty, and measurementProtocol. These are the attributes necessary for accessing the sensor information.

Concerning the modeling of communication units and protocols, it is worth noting that we did not introduce a specific stereotype for communication devices, like routers or cables, but we model them as *PhysicalUnits*. On the contrary, we introduce a specific *Communication* stereotype to model interactions between *VirtualUnits*. Each communication realizes a particular *ProtocolType* between different infrastructure elements. The supported protocol values are popular protocols, such as MQTT, HTTP, TCP, UDP, and AMQP. Finally, any *Infrastructure* provide *CloudServices* of different types (see CloudServiceTypes enumeration including VM, Disk, StorageService, and DataAnalyticsEngine) corresponding to cloud offerings by cloudProvider and dataProvider.

3.2 Uncertainty Modeling

In testing uncertainties of IoT Cloud Systems, we need to capture possible uncertainties and model them in a systematic way. Based on our uncertainty taxonomy [2] we develop an Uncertainty Profile to model uncertainties inherent in IoT Cloud Systems.

An excerpt of the profile is depicted in Figure 3. The *InfrastructureUncertainty* extends the core *Uncertainty* stereotype and it is characterized by the following properties modeled as UML Enumeration types, namely: *TemporalManifestationType*, *LocationType*, *NonFunctionalDimensionalityType*, *CauseType*, *ObervationTimeType*, *FunctionalDimensionalityType*, *EffectPropagationType*. We then identify different *UncertaintyFamilies*, namely: *DataDeliveryUncertainty*, *ActuationUncertainty*, *ExecutionEnvironmentUncertainty*, *GovernanceUncertainty*, *ElasticityUncertainty*, *StorageUncertainty*.

Each family is characterized by a particular set of values assigned to infrastructural uncertainty properties that determine whether an uncertainty belongs to a particular family. The Infrastructure Level Uncertainty profile is part of the Uncertainty Modeling Framework (UMF) provided by U-Test project [3]. In [2] a detailed domain model is provided. Due to space limitation, detailed uncertainty family descriptions are given in [2].

3.3 Test Configurations Modeling

In order to test IoT Cloud Systems, we need to further extend the system model with testing-specific concepts. Therefore, we define a Testing Profile (see Figure 4). Together with the Infrastructure and Uncertainty Profiles (see Figures 2 and 3, respectively), it represents a core asset to create the UML input artifacts to our tool pipeline.

Given the system *Infrastructure* and the associated *InfrastructureUncertainty* information, different testing goals are possible, e.g., i) testing the *DataDeliveryUncertainty* between *Units* like *VirtualGateways* and *CloudServices* and/or ii) testing the performance of *VirtualGateways* and *CloudServices*. Furthermore, there are many providers and infrastructures that can be selected by developers

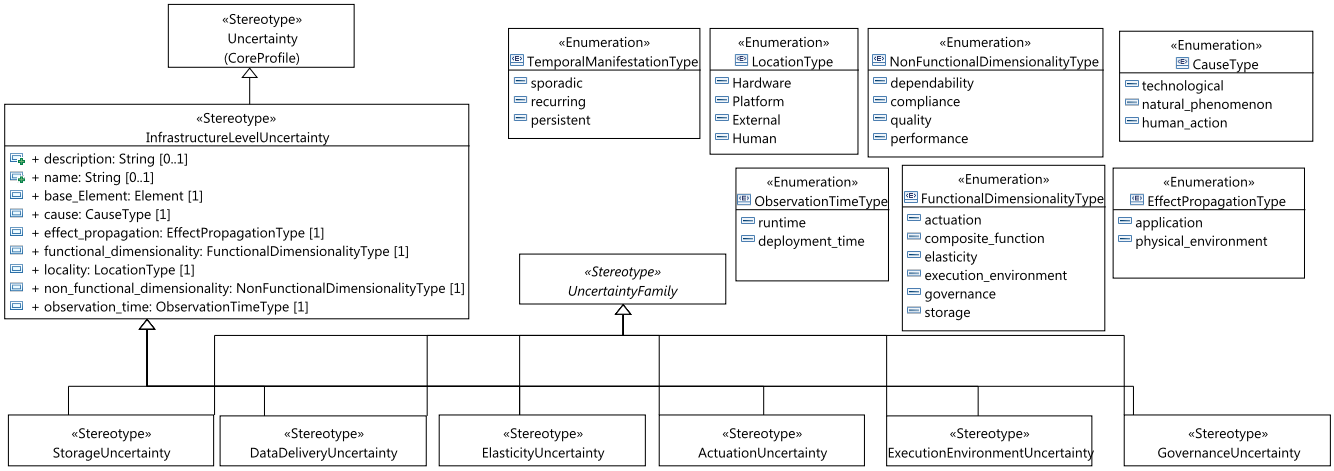

Figure 3: Infrastructure Uncertainty profile.

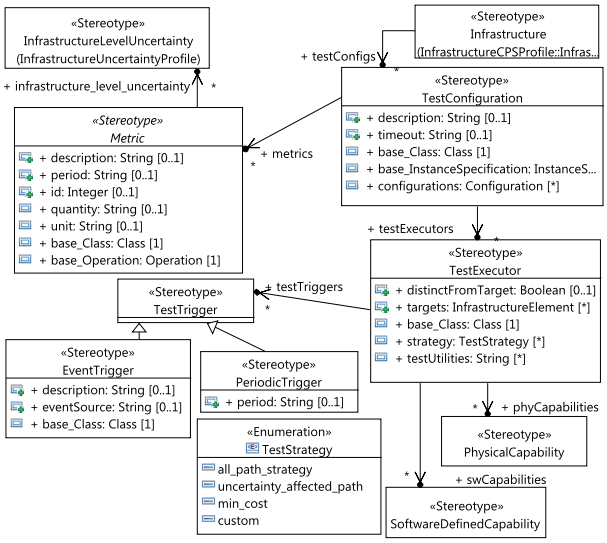

Figure 4: Infrastructure Testing profile.

and then used for testing. Because of that, we need to capture *TestingConfigurations*, depicted in Figure 4. *Metrics* to be tested is then associated to *TestConfigurations* and strongly influenced by *InfrastructureUncertainties*. The *TestConfiguration* has a name, description and a timeout. The timeout gives the maximum amount of time in which the associated *TestExecutor* should answer the test.

The *TestExecutor* has a description, targets to be tested (i.e., SUT infrastructure elements), test utilities (used to perform the test) as well as SUT deployment. Therefore, TestExecutor is quite similar to test cases in contemporary testing systems but it is extended with dynamic deployment of IoT and Cloud resources for SUT. Multiple strategies (e.g, executing all the possible execution paths or minimizing a certain cost function) can be specified to influence its execution.

TestTrigger describes when a test should be executed and it is of two types, either *EventTrigger* or *PeriodicTrigger*. The *EventTrigger* has two attributes, description of the event, and the event source, and it is used for event-based testing (e.g., when, during system runtime, the quality is too low). The *PeriodicTrigger* has two attributes, i.e., the period and the time unit, and it is used for tests executed in specific periods described under various units of time.

Note that not all information need to be specified during the modeling. Many types of information can be generated from different tools. For example, the *Metric* and *TestExecutor* can be generated from models, uncertainties and other information. At runtime, for testing, we have descriptions of test plans. Such test plans can be generated in different ways. One example is to generate the test plans based on our DSL in [24] or model-based test case generation [33].

4 PROVISIONING SYSTEM UNDER TEST

4.1 Extracting Information from Models

IoT and Cloud resources modeled, communication protocols, expected uncertainties to be tested, etc., are extracted from models into JSON-based descriptions. The key thing is to enable various tools to use the extracted information for different purposes. In this paper, the extracted information is used to determine test configuration and deployment (but not for performing the test - which is out of the scope of this paper). We implement the information extraction using the EPSILON framework. EPSILON provides a tool ecosystem for MDE activities, including a template-based model-to-text language for generating code, documentation and other textual artifacts from (UML) models. Listing 1 shows an executable extraction rule (using EPSILON EGX co-ordination program) that transforms each UML Class annotated with the *VirtualSensor* stereotype to JSON (using a EGL template), shown in Listing 2. During the extraction process, the EPSILON framework replaces dynamic placeholders (i.e., variables declared within [%%]) with property values extracted from stereotype applications.

Listing 1: Transformation rule for VirtualSensor

```
rule VirtualSensor2JSON
transform virtual_sensor : Class {
guard : virtual_sensor.hasStereotype("
    VirtualSensor")
template : "JSONTemplate.egl"
target : "virtualsensors/" + virtual_sensor.name+
    ".json"
}
```

Listing 2: Template for VirtualSensor JSON files

```
{
"name": "[%=virtual_sensor.base_Class.name%]",
"swCapabilities": [%=virtual_sensor.swCapabilities
    %],
"deployedOn": [%=virtual_sensor.deployedOn%],
"type": "VirtualSensor"
}
```

4.2 Artifact Repositories and Runtime Information Services

In order to support testing, SUT model elements have to be bound to concrete artifacts and testing utilities, usually stored in repositories. SUT elements, such as, virtual machines, MQTT brokers, gateway artifacts, are typically prebuilt and managed in different repositories. Similarly, testing utilities are also diverse, due to differences between the underlying infrastructure providers. In this paper, we consider that artifacts for testing (P_a, SUT_a and TU_a) are available[1]. We leverage software artifacts from *i)* public repositories (such as Docker hub and Google Registry), which provide state-of-the-art technologies, such as, for deploying and managing containers and VMs, and *ii)* user-provided repositories with similar technologies, such as using Google Storage and Docker Registry. For the developer of SUT, our tool needs to connect different repositories to search suitable artifacts. Therefore, we develop a metadata service based on MongoDB for our artifacts. While artifacts can be stored in different repositories, the developer will need to provide metadata so that our configuration generation tool can search the right artifacts for the right underlying infrastructures. We also need to rely on resource information services to provide information about running instances of artifacts and SUT elements. For example, when we know that a VM instance P_a is available for use, we can just deploy a SUT_a onto P_a, instead of deploying another VM. For runtime information, we use HINC [17] and SALSA [19].

4.3 Selecting SUT-related Artifacts

To enable the search of suitable artifacts for SUT, we impose a set of guidelines to describe artifact capabilities (there is no model that describing testing artifacts). After such artifacts are built, they are deposited into a repository and metadata will be stored into our services. In order to search the right artifacts, we have different metadata associated with artifacts. Such metadata will be searched by using information extracted from the model (e.g., type of service units, and protocol supports). We use the following convention for metadata:

- *t4u*/*abstractelement*/*concreteelement* : *tag*: where as *t4u* is the name of our system, *abstractelement* indicates the abstract type of SUT elements, e.g., `VirtualSensor`, and *concreteelement* indicates concrete type of SUT elements, such as `ElectricitySensor`. *tag* is used to add new information. For example, the string `t4u/VirtualSensor/Electricitysensor:raspberrypi` indicates images and testing utilities for instances of `ElectricitySensor` in Raspberry PI.
- Each artifact has meta information about how to invoke and reconfigure it. For this, we use a convention: *startup*, *shutdown*, *configure* scripts with input (JSON) parameters. To be generic, we do not guarantee the correctness of these functions but we require these functions in order to reconfigure and start SUT elements. This requirement is conventional, widely used in practice, for dealing with configuration of software components, which can be implemented through REST (for Web service), gRCP (for RPC call-based objects), and shell scripts (for executable artifacts).

For example, a developer can i) develop a virtual sensor in Python/Java as an element of the CPS and ii) create a Docker file for the sensor. A Docker image can be built and deposited into the repository. Listing 3 shows an example of a Docker file that bind a concrete MQTT broker to the corresponding CloudService, and make it available to developers on a repository.

Listing 3: Example of storing artifacts and metadata

```
$docker tag mqttsensor localhost:5000/t4u/
    cloudservice/mqttbroker:v01
$docker push localhost:5000/t4u/cloudservice/
    mqttbroker
$t4u_metadata add localhost:5000/t4u/cloudservice/
    mqttbroker:v0
```

4.4 Configuration Generation and Deployment

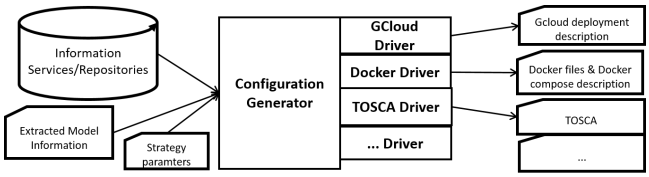

Figure 5: Deployment configuration and description generator

From the extracted information, we connect to repositories of artifacts and we have different techniques to generate deployment configurations and deployment descriptions. Figure 5 shows the design. *Configuration Generator* will provide different possibilities of deployment configurations for elements of SUTs, e.g., whether a software sensor will be executed in a small virtual machine or not. After that, several drivers will be used to provide the detailed deployment descriptions, which are used to deploy several instances of SUT elements for testing. For example, a deployment description can include how many sensors, virtual machines, message brokers,

[1]How to build such artifacts are out of the scope of this paper. In our work, currently, the developer develops such artifacts using different techniques.

cloud data services, etc., should be deployed and to where (e.g., local cloud or Google). In our work, we use SALSA [18], docker tools, and cloud-specific tools (e.g., Google gcloud) for deployment (but this can be done by many other tools). We will provide further examples in Section 5.2.

5 EVALUATION

5.1 Modeling IoT Cloud Systems under Test

To illustrate our approach, we perform uncertainty modeling and testing for an IoT cloud system for monitoring infrastructures of Base Transceiver Stations (**BTS Monitor**)[2]. The SUT includes a set of hardware sensors reading various electricity parameters (*HwElectricitySensor*), e.g., high or low AC voltages, backup powers and power interruption, temperature (*HwTemperatureSensor*) (outdoor and indoor of the BTS), status of air conditioners (*HwAirConditioningSensor*) (e.g., failures, alarms, etc.), and the corresponding software counterparts governing these hardware devices through APIs (*ElectricitySensor*, *TemperatureSensor*, and *AirConditioningSensor*). Within a BTS, these sensors push monitoring data to an IoT Gateway using Raspberry PI. The *IoTGateway* uses MQTT protocol to push monitoring data to the cloud and receives commands from the cloud (e.g., for controlling air conditioners). At the cloud side, we have NodeJS and Python data ingest components (*IngestClient*) that take the data and store into a storage service, which, in turn, can be realized through Google *BigQuery* and/or *Cassandra*. For testing purposes, we emulate sensors using Java/Python-based software sensors and reused historical data from the real system. Other components are deployed in the real system (although configurations are not exactly like in the production).

Figure 6 shows an excerpt of the BTS system architecture model created using Papyrus UML tool. A Class Diagram shows classes and associations annotated with stereotypes from our profile. They represent sensors, their virtual counterparts, and Raspberry PI gateway. Clearly, this way enables the developers to stay at the high-level design and to customize different parameters for their tests. It is worth noting that we partially reuse the resource modeling support and guidelines of the OMG MARTE profile to obtain a parameterized model. MARTE defines foundation concepts for embedded system modeling in UML, defining different kinds of resources (processing, storage, and communication) at different level of abstraction (software, hardware) [27, 30]. In particular, we add i) resource multiplicities (via *Resource* stereotypes), ii) speed factors of computing resources (via *HwProcessor*) and memory sizes (via *HwMemory* and *HwRAM*). However, strict compliance with MARTE modeling guidelines is out of scope and left as future work.

The bottom of Figure 6 shows test configuration model elements. A test configuration (e.g., *Test001*) is a collection of different configurations for computing, storage, and communication resources modeled in the SUT architecture depicted above:

- The *MQTTConfigClient* class collects all the information required to set up the communication between SUT components via the MQTT protocol.
- The *CassandraConfig1* class collects all the information required to set up the storage service using Cassandra.
- The *GoogleBigQueryConfig1* class collects all the information required to set up the storage service using the Google BigQuery.

In our example, sources of uncertainty are incomplete/wrong test configurations. Figure 7 shows the expected uncertainty to be tested. All stereotypes annotations that we have not explicitly introduced in our profiles (see Figures 2,3 and 4) come from the U-Test core Uncertainty profile [3]. A *BeliefAgent*, (e.g., a developer), is in charge of testing the SUT to verify whether beliefs (i.e., BeliefElements) about the SUT hold or not, and to identify the indeterminacy source(s) for that beliefs. In our example, a developer knows that the MQTT protocol will be used for communication among many infrastructural elements. He/she further believes that at least the 90% of messages sent by each unit are delivered. He/she knows that MQTT provides a quality of service level (qosLevel of `MQTTConfig1` in Figure 6) as an agreement regarding the guarantees of messages delivery. There are three QoS levels in MQTT expressed as integer: at most once (0), at least once (1), and exactly once (2)). However, he/she does not know which QoS level configurations for communication channels (i.e., the *Communication* associations in Figure 6) fit best with network reliability. Therefore, `MQTTConfig1` represents an indeterminacy source, caused by missing information, for beliefs hold by developers. It means that developers have to deal with *DataDeliveryUncertainty* (see Figure 7).

For this reason, diffrent SUT deployments and configuration can be generated from the same system infrastructure model, depending on the identified uncertainties. Note that by connecting to the configuration generation and information services, the developer could obtain further information and update his/her models. For example, in Figure 6, `MQTTConfig2Client` can be concrete due to the availability of a concrete MQTTBroker configuration available in a Google virtual machine (with a real IP address). Such information might not be available at the beginning of the modeling due to the lack of knowledge about existing resources. However, such a lack of knowledge can be solved through the configuration generation and deployment of test configuration (even though the test has not been started).

5.2 SUT Deployment and Configuration

Consider, for example, two different cases in testing data uncertainty. In the first case, the developer is interested in only the data uncertainty from the gateway to the cloud (one may assume that this part has many types of uncertainty). In the second case, the developer is also interested in uncertainties of the *IngestClient* configured with *BigQuery*. This requires us to create different deployment configurations. As our prototype is currently being implemented and due to the lack of space, we just illustrate some examples[3].

From the extracted information in models, we generate different types of information: (i) JSON based configurations for single resources and their communications, and then (ii) concrete deployment descriptions. For (i) we obtain a valid input model for the Generating Test Configuration phase (see Figure 1) to generate multiple deployment configurations, represented in JSON files. In this step, given the extracted information from an element in the model, e.g., a `BTSBroker`, depending on testing strategies (e.g., Local or

[2]We rely on a real system for BTSs in Vietnam from our partners. Due to confidentiality constraints, we abstract only the main parts in our example

[3]The prototype will be continously updated in https://github.com/rdsea/

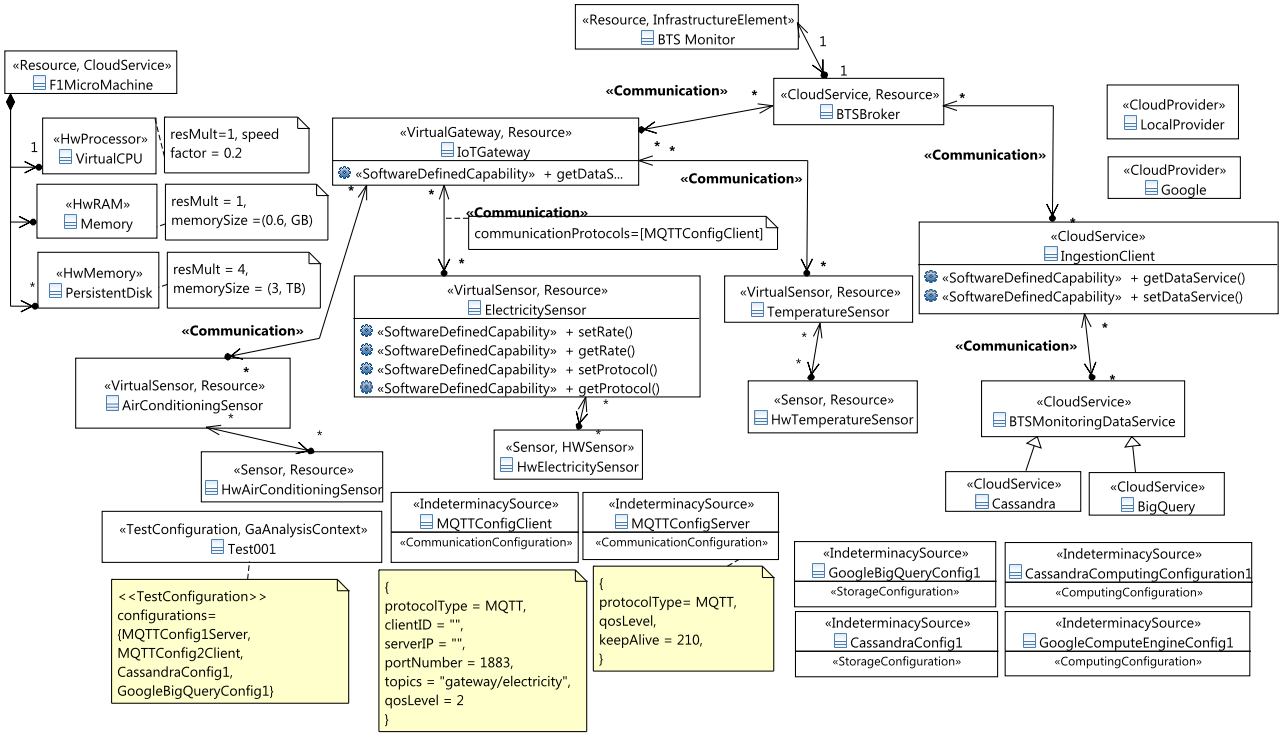

Figure 6: Annotated Class Diagram of the Telco BTS monitoring system showing structure and test configuration.

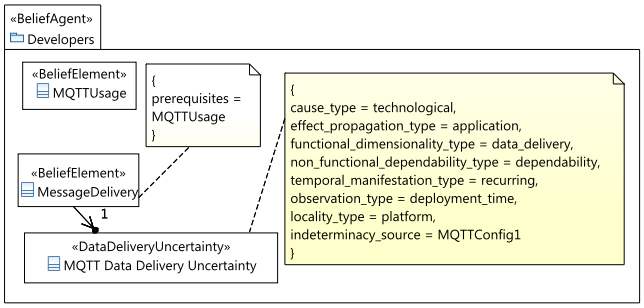

Figure 7: Data uncertainty modeling affecting the TemperatureSensor behavior modeled as UML StateMachine.

Cloud) and uncertainties (e.g., DataUncertainty), possible deployment configurations will be created, e.g., `BTSBroker1Local` and `BTSBroker2Google`. Shown in Listing 4, examples of deployment configurations are:

- `BTSBroker` can be in a local deployment or Google deployment, leading to two deployment configurations: `BTSBroker1Local` and `BTSBroker2Google`.
- `MQTTConfig1` for any `IoTGateway` instances when `IoTGateway` supports MQTT communication.
- `MQTTConfig2` can be used in the case an instance of `ElectricitySensor` connects to an instance of `IoTGateway` deployed in Google VM through `CommElectricitySensorIoTGateway1`.

In this case, the configuration has specific IP address of the IoT-Gateway instance and concrete QoS, due to test strategies.

Listing 4: Extracted simplified example of deployment configurations

```
"BTSBroker1Local":  {
   "name": "BTSBroker",
    "cloudProvider": ["local"],
    "communicationConfigs": ["MQTTConfig1"],
    "type": "CloudService"
}
"BTSBroker2Google":  {
   "name": "MQTTBroker",
    "cloudProvider": ["Google"],
    "communicationConfigs": ["MQTTConfig1"],
    "type": "CloudService"
 }
"MQTTConfig1":  {
       "name": "MQTTConfigServer",
       "protocolType": "MQTT",
       "qosLevel": [],
       "keepAlive": 210,
       "type": "CommunicationConfiguration"
}
"MQTTConfig2":  {
      "name": "MQTTConfigClient",
       "protocolType": "MQTT",
       "clientID": "",
       "serverIP": "35.189.187.208",
       "portNumber": 1883,
```

```
        "topics": ["/gateway/electricity"],
        "qosLevel": [2],
        "type": "CommunicationConfiguration"
}
"ElectricitySensor1":{
        "name": "ElectricitySensor",
        "swCapabilities": ["setRate", "getRate"],
        "ownedUnits": ["HwElectricitySensor"],
        "type": "VirtualSensor"
}
"CommElectricitySensorIoTGateway1":     {
        "name": "CommElectricitySensorIoTGateway",
        "connection_end_points": ["
            ElectricitySensor1", "
            IoTGateway1Google"],
        "communicationConfigs":"MQTTConfig2",
        "type": "Communication"
 }
```

From the JSON-based deployment configurations, we determine concrete deployment descriptions; the descriptions must be concrete for existing tools to perform the deployment. This is also based on a concrete decision on where to test, cost, availability of underlying infrastructures. As shown in Section 4.4, this will be done by processing all deployment configurations and using different drivers to provide concrete deployment descriptions. For example, Listing 5 shows docker compose files for deployment description in local machines for testing. List 6 shows TOSCA-based descriptions for SALSA to deploy the SUT. Similarly, one can also have gcloud-based scripts (https://cloud.google.com/sdk/gcloud/) to deploy SUT.

Listing 5: Extracted simplified example of a Docker-compose deployment description

```
version: '3'
services:
  ingest:
    build: .
    volumes:
      - ./:/t4u
  electricitysensor:
     image: "localhost:5000/t4u/mqttsensor/
         realsensor:v01"
  iotgateway:
     image: "localhost:5000/t4u/cloudservice/
         mqttbroker:v01"
```

Listing 6: Example of TOSCA deployment description

```
<ns2:NodeTemplate maxInstances="10"
  id="electricitysensor" type="salsa:os">
  <ns2:Properties>
   <MappingProperties>
    <MappingProperty type="salsa:os">
    <property name="provider">...</property>
     <property name="instanceType">...</property>
      <property name="baseImage">...</property>
    </MappingProperty>
  </MappingProperties>
 </ns2:Properties>
</ns2:NodeTemplate>
```

6 RELATED WORK

Various applications and systems based on IoT and Cloud resources have been developed, such as [5, 16]. While various aspects in terms of design of such systems and applications have been discussed, e.g., network and protocols, we have not seen the issues of testing and how to connect from the design to the deployment of systems in an integrated manner. Recently, some work have focused on analyzing uncertainties and failures [4, 21], but they do not focus on connecting IoT/CPS design and engineering tasks with testing. Nevertheless, they recognize the importance of dealing with uncertainties.

Modeling IoT Cloud Systems Elements: Most MDE approaches and tools for cloud just map model elements to cloud resources, e.g. [7, 14]. IoT is a new field for modeling [25] and model-driven techniques are recognized as a mean to develop applications for the IoT [28]. Many papers address the modeling IoT and CPS [11, 31]. Our paper goes in this direction i) by explicitly representing uncertainty as first class concept in our models and, then ii) supporting model-driven deployment and testing under uncertainty. Our approach is built on the view that programming of IoT cloud systems will be moved to higher levels [28]. Most of existing work do not include uncertainties in their modeling, let alone the SUT provisioning and testing.

Modeling Test Configuration and Uncertainty: Testing is well-known domain but testing uncertainty in IoT Cloud systems and CPS is very new, emerging direction. Our focus in this paper is to enable testing uncertainty, not on the test process itself. Nevertheless, our SUT is provisioned based configurations generated from various types of information, including uncertainties to be tested, artifact information, test strategies, etc. Due to the lack of space and as the prototype is still being developed, we have not detailed all possibilities but presented main designs and examples.

Generating and Deploying Infrastructures: Deploying and configuring IoT and cloud are hot topics. In the Cloud it is quite mature [20, 34], but it is ongoing work for IoT [6]. In [23] an approach to the generation of IoT infrastructures has been presented, but it has no connection to testing purpose. In this paper, we leverage existing techniques, including our own previous development. However, existing techniques have not well-integrated with modeling and testing. Different from existing work, which focus on deployment techniques based on existing system description, we generate deployment of SUT in IoT Cloud infrastructures based on the need of uncertainty testing. Therefore, our work is complementary to others.

7 CONCLUSIONS AND FUTURE WORK

In this paper we show the need to model and provision IoT Cloud Systems under test for uncertainty testing from the perspective of software and system developers. To simplify the task of the developers, we enable them to model SUT and uncertainties at a high-level and generate required infrastructures for testing. We presented a tool pipeline, ranging from extracting modeling information, generating test configuration, creating deployment and provisioning IoT cloud infrastructures. With such a tool pipeline, one can enable testing uncertainties with various underlying cloud and IoT providers. .

Prototyping and validating IoT Cloud Systems configurations and descriptions of SUT for uncertainty testing are challenging. While in our prototype the modeling part is mature, our implementation for possible algorithms for generating optimal deployment configurations and descriptions, based on various parameters of uncertainties, test strategies, costs and underlying cloud providers, is just at an early stage that needs to be addressed in the future. Furthermore, we will focus on building different adapters that enable the integration with various underlying IoT and cloud platforms and providers.

ACKNOWLEDGMENT

This work was partially supported by the European Commission in terms of the U-Test H2020 project (H2020-ICT-2014-1 #645463). Georgiana Copil and Ivan Pakovic contributed to the development of profiles when they were at TU Wien. We thank Daniel Moldovan for his initial contribution on test configuration structures.

REFERENCES

[1] ARTIST Project: Advanced software-based seRvice provisioning and migraTIon of legacy SofTware. http://www.artist-project.eu/. (????). Accessed: 2017-06-22.

[2] U-Test H2020 Deliverable: Revision of deliverable report D1.2: Updated Report on U-Taxonomy. https://www.simula.no/file/d12pdf/download. (????). Accessed: 2017-06-22.

[3] U-Test H2020 Project Web Site. http://www.u-test.eu/. (????). Accessed: 2017-04-14.

[4] Ilge Akkaya, Yan Liu, and Edward A. Lee. 2016. Uncertainty Analysis of Middleware Services for Streaming Smart Grid Applications. *IEEE Trans. Services Computing* 9, 2 (2016), 174–185. https://doi.org/10.1109/TSC.2015.2456888

[5] Pandarasamy Arjunan, Mani Srivastava, Amarjeet Singh, and Pushpendra Singh. 2015. OpenBAN: An Open Building ANalytics Middleware for Smart Buildings. In *Proceedings of the 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services on 12th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MOBIQUITOUS'15)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 70–79. https://doi.org/10.4108/eai.22-7-2015.2260256

[6] Paolo Bellavista and Alessandro Zanni. 2017. Feasibility of Fog Computing Deployment Based on Docker Containerization over RaspberryPi. In *Proceedings of the 18th International Conference on Distributed Computing and Networking (ICDCN '17)*. ACM, New York, NY, USA, Article 16, 10 pages. https://doi.org/10.1145/3007748.3007777

[7] Amine Benelallam, Abel Gómez, Massimo Tisi, and Jordi Cabot. 2015. Distributed Model-to-model Transformation with ATL on MapReduce. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015)*. ACM, New York, NY, USA, 37–48.

[8] Marie-Luce Bourguet. 2016. Designing More Robust Ubiquitous Systems. In *Proceedings of the European Conference on Cognitive Ergonomics (ECCE '16)*. ACM, New York, NY, USA, Article 39, 4 pages. https://doi.org/10.1145/2970930.2979719

[9] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. *Model-Driven Software Engineering in Practice*. Morgan & Claypool.

[10] Federico Ciccozzi and Romina Spalazzese. 2016. MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering. In *International Symposium on Intelligent and Distributed Computing*. Springer, 67–76.

[11] Federico Ciccozzi and Romina Spalazzese. 2017. *MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering*. Springer International Publishing, Cham, 67–76.

[12] Ana C. Franco da Silva, Uwe Breitenbücher, Kálmán Képes, Oliver Kopp, and Frank Leymann. 2016. OpenTOSCA for IoT: Automating the Deployment of IoT Applications Based on the Mosquitto Message Broker. In *Proceedings of the 6th International Conference on the Internet of Things (IoT'16)*. ACM, New York, NY, USA, 181–182. https://doi.org/10.1145/2991561.2998464

[13] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. 2015. Edge-centric Computing: Vision and Challenges. *SIGCOMM Comput. Commun. Rev.* 45, 5 (Sept. 2015), 37–42.

[14] Michele Guerriero, Saeed Tajfar, Damian A. Tamburri, and Elisabetta Di Nitto. 2016. Towards a Model-driven Design Tool for Big Data Architectures. In *Proceedings of the 2Nd International Workshop on BIG Data Software Engineering (BIGDSE '16)*. ACM, New York, NY, USA, 37–43.

[15] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. 2011. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. 471–480.

[16] Sylvain Kubler, Jérémy Robert, Ahmed Hefnawy, Chantal Cherifi, Abdelaziz Bouras, and Kary Främling. 2016. IoT-based Smart Parking System for Sporting Event Management. In *Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MOBIQUITOUS 2016)*. ACM, New York, NY, USA, 104–114. https://doi.org/10.1145/2994374.2994390

[17] Duc-Hung Le, Nanjangud C. Narendra, and Hong Linh Truong. 2016. HINC - Harmonizing Diverse Resource Information across IoT, Network Functions, and Clouds. In *4th IEEE International Conference on Future Internet of Things and Cloud, FiCloud 2016, Vienna, Austria, August 22-24, 2016*, Muhammad Younas, Irfan Awan, and Winston Seah (Eds.). IEEE Computer Society, 317–324. https://doi.org/10.1109/FiCloud.2016.52

[18] Duc-Hung Le, Hong Linh Truong, Georgiana Copil, Stefan Nastic, and Schahram Dustdar. 2014. SALSA: A Framework for Dynamic Configuration of Cloud Services. In *CloudCom*. IEEE Computer Society, 146–153.

[19] Duc-Hung Le, Hong Linh Truong, and Schahram Dustdar. 2016. Managing On-Demand Sensing Resources in IoT Cloud Systems. In *2016 IEEE International Conference on Mobile Services, MS 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, Manish Parashar, Hemant K. Jain, and Hai Jin (Eds.). IEEE Computer Society, 65–72. https://doi.org/10.1109/MobServ.2016.20

[20] Duc-Hung Le, Hong-Linh Truong, G. Copil, S. Nastic, and S. Dustdar. 2014. SALSA: A Framework for Dynamic Configuration of Cloud Services. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*. 146–153.

[21] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F. Karlsson, Dongmei Zhang, and Feng Zhao. 2016. Systematically Debugging IoT Control System Correctness for Building Automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys '16)*. ACM, New York, NY, USA, 133–142. https://doi.org/10.1145/2993422.2993426

[22] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. 2013. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering* 39, 6 (2013), 869–891.

[23] S. Mohamed, M. Forshaw, and N. Thomas. 2017. Automatic Generation of Distributed Run-Time Infrastructure for Internet of Things. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 100–107. https://doi.org/10.1109/ICSAW.2017.51

[24] Daniel Moldovan and Hong Linh Truong. 2016. A Platform for Run-Time Health Verification of Elastic Cyber-Physical Systems. In *24th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2016, London, United Kingdom, September 19-21, 2016*. IEEE Computer Society, 379–384. https://doi.org/10.1109/MASCOTS.2016.14

[25] B. Morin, N. Harrand, and F. Fleurey. 2017. Model-Based Software Engineering to Tame the IoT Jungle. *IEEE Software* 34, 1 (Jan 2017), 30–36. https://doi.org/10.1109/MS.2017.11

[26] Inc Object Management Group. 2015. Unified Modeling Language, UML, version 2.5. http://www.omg.org/spec/UML. (2015). Accessed: 2017-04-14.

[27] Object Management Group (OMG). 2016. *UML Profile for MARTE*. Version 1.1 http://www.omg.org/spec/MARTE/1.1/PDF.

[28] Pankesh Patel and Damien Cassou. 2015. Enabling high-level application development for the Internet of Things. *Journal of Systems and Software* 103 (2015), 62 – 84. https://doi.org/10.1016/j.jss.2015.01.027

[29] Ragunathan (Raj) Rajkumar, Insup Lee, Lui Sha, and John Stankovic. 2010. Cyber-physical Systems: The Next Computing Revolution. In *Proceedings of the 47th Design Automation Conference (DAC '10)*. ACM, New York, NY, USA, 731–736. https://doi.org/10.1145/1837274.1837461

[30] Bran Selic and Sébastien Gérard. 2013. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. Elsevier.

[31] Kleanthis Thramboulidis and Foivos Christoulakis. 2016. UML4IoT-A UML-based Approach to Exploit IoT in Cyber-physical Manufacturing Systems. *Comput. Ind.* 82, C (Oct. 2016), 259–272. https://doi.org/10.1016/j.compind.2016.05.010

[32] Hong-Linh Truong and S. Dustdar. 2015. Principles for Engineering IoT Cloud Systems. *Cloud Computing, IEEE* 2, 2 (Mar 2015), 68–76.

[33] Mark Utting and Bruno Legeard. 2007. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[34] Matthew S. Wilson. 2009. Constructing and Managing Appliances for Cloud Deployments from Repositories of Reusable Components. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing (HotCloud'09)*. USENIX Association, Berkeley, CA, USA, Article 16. http://dl.acm.org/citation.cfm?id=1855533.1855549