Model-Driven Engineering of Monitoring Application for Sensors and Actuators Networks

Thibault Béziers La Fosse, Zheng Cheng, Jérôme Rocheteau, Jean-Marie Mottu

Model-Driven Engineering of Monitoring Application for Sensors and Actuators Networks

Thibault Béziers la Fosse
IMT Atlantique
Nantes, France
thibault.beziers-la-fosse@imt-atlantique.fr

Zheng Cheng
ICAM Nantes
Nantes, France
zheng.cheng@icam.fr

Jérôme Rocheteau
ICAM Nantes
Nantes, France
jerome.rocheteau@icam.fr

Jean-Marie Mottu
LS2N
Nantes, France
jean-marie.mottu@ls2n.fr

***Abstract*—Cyber-Physical Systems (CPSs) encompass both Information Infrastructures and networks of physical devices. Applications for monitoring them are usually implemented after the physical systems that they have to monitor: the CPS has to be running in order to monitor it. Monitoring applications define communications between the elements of those systems, and are compliant with their designs. Such applications can be complicated to develop, maintain and evolve, especially if the physical system changes. This paper brings an approach for engineering CPS monitoring applications, relying on model-driven techniques for generating a platform using the MQTT communication protocol for monitoring the CPS. First, it provides a meta-model for modeling sensors and actuators networks. Second, it introduces EMIT a monitoring platform for sensors and actuators networks. Third, it presents a transformation from models of sensors and actuators networks to automatically generate EMIT monitoring configuration. We illustrate our approach on a case study and discuss its limitations and improvement points.**

***Index Terms*—Cyber-Physical Systems, Model-Driven Engineering, Monitoring**

I. INTRODUCTION

A Cyber-Physical System (CPS) refers to a system that is composed of both physical devices and software artifacts. In such network, hardware entities communicate with an Information Infrastructure, and are orchestrated for certain tasks, such as home automation, car automation, e-health, smart cities, and "Industry 4.0" [4], [11], [27], [19], [1].

Over the past years, the number of CPS increased quickly, and is expected to grow even quicker in the forthcoming years [18]. This would make the design of their monitoring applications more and more challenging, as their complexity gets more important. In this article we consider a specific type of CPS: Sensors and Actuators Network (SAN). Sensors are small entities mainly used for sensing, processing and/or sending data, whereas Actuators are controllable entities requiring external input [2], [26]. Such network requires low latency, valid data output, good coordination between sensors and actuators, and a long lifetime. To fulfill these requirements, being able to efficiently monitor the SAN is mandatory.

A common way of designing the monitoring of a CPS is to setup physical devices to be monitored, and configure and adapt existing tools to combine them into a monitoring application dedicated to that CPS [12]. It requires a dispensable and error-prone dependency: the implementation of the physical part needs to be available first, in order to drive the implementation of the software part. This makes such application complicated to develop, maintain and update.

In the past decade, Model-Driven Engineering (MDE) has shown that using models instead of standard code-based engineering practices improves the development phases, especially in term of quality, reuse and productivity [7], [35]. MDE has been successfully used for years for designing, developing and analyzing components for CPSs [5], [13], [10], [21]. As the design of CPS monitoring application often suffers from recurrent coordination and coherence issues, we propose to leverage MDE in that purpose. Relying on models for designing both the physical part and monitoring part of the CPS would tackle those issues, as well as improving the quality, correctness and extendability of such monitoring platform.

Applying MDE to SAN has been investigated ([30], [28], [3]), and languages such as SensorML[8] based on a meta-model, or SOSA[17] and SSN[24], based on ontologies, have been proposed. However, we don't know any MDE approach to monitor the SAN execution while analyzing and persisting measurements. Even if a standard *Structured Metrics Meta-model* (SMM) [25] exists, it is not integrated into SAN modeling languages. This results in a lack of modularity and reusability since maintenance of both parts (the SAN itself and its monitoring applications) is made separately.

Therefore, in this work we leverage MDE and propose the three-step approach illustrated in Figure 1. First, the SAN is designed using a model, conforming to a SAN meta-model. The SAN meta-model we propose encompasses concepts from the OMG's Structured Metrics Meta-model (SMM) [25], the Semantic Sensor Network ontology (SSN) [24], as well as SensorML [8]. This model describes the entities in the system, data-types sent and received, as well as the events and processes happening in the system. It provides a better view and understanding of the system, instead of directly implementing it. Second, a model-transformation is performed on this model. This transformation generates a set of HTTP queries, that are executed in order to configure EMIT, the monitoring platform we propose. Third, using the MQTT protocol, EMIT is able to monitor the SAN, and perform additional analysis. By leveraging MDE for generating a monitoring platform for a CPS, this approach fosters a better reusability, understandability, and provides a safer way of structuring SANs than code-based engineering approaches.

This paper is organized as follow: Section II first describes the SAN meta-model, Section III presents EMIT, our monitoring platform, Section IV presents our model transformation for adding SAN's entities into EMIT and enabling their monitoring, a use-case is shown in Section V, we present some related work in Section VI and Section VII concludes the article.

II. SENSOR AND ACTUATOR NETWORK MODELING

The Meta-Object Facility (MOF) standard defines a meta-model as a model that defines the language for expressing a model. It typically describes the languages and processes from which to form a model [14]. We propose a SAN meta-model as shown in Figure 2. It aims at generically representing the structural and behavioral characteristics of each SAN. From a structural perspective, our meta-model enables the specification of network entities (e.g. sensors and actuators) inside areas, and the relationships between them. From a behavioral perspective, our meta-model allows users to specify the heterogeneous data that are gathered by sensors, the communications between the network entities, and the way actuators behave after receiving data. In what follows, we detail our SAN meta-model.

1) Foundations: As written in the introduction, ontologies' work has proposed several SAN modeling languages. However, since they are not integrated in a MDE approach, they lack modularity and reusability, in order to monitor the SAN execution while analyzing and persisting the measurements. We have created our SAN meta-model based on the following works.

OpenGIS proposes SensorML [8]. It can be used for modeling many aspects of sensors networks: data-types, positions, temporal data, encoding, observable properties, processes, etc. SensorML covers most of the needs but does not enable the representation of actuators.

Neuhaus et al. propose an ontology for describing sensor networks [24]. Sensors can be defined with positions, measures, and processes can be triggered when specified guards are validated. However, it lacks the description of *Measurands* on which measurements are performed, and a more reusable way of defining Sensors (and Actuators), by first defining their model.

To model measures and persist measurements, the *Object Management Group* has defined the *Structured Metrics Meta-model* (SMM) [25]. It provides a unified way for representing the measurement information and the measures performing such measurements, without detailing the entities measured.

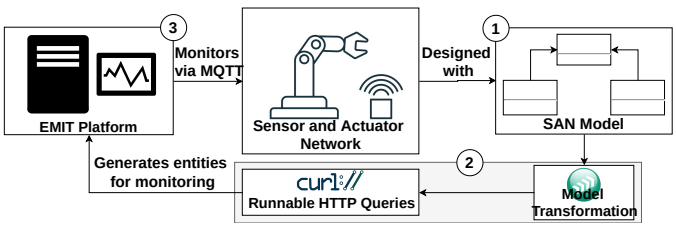


Fig. 1: Approach for SAN monitoring in EMIT

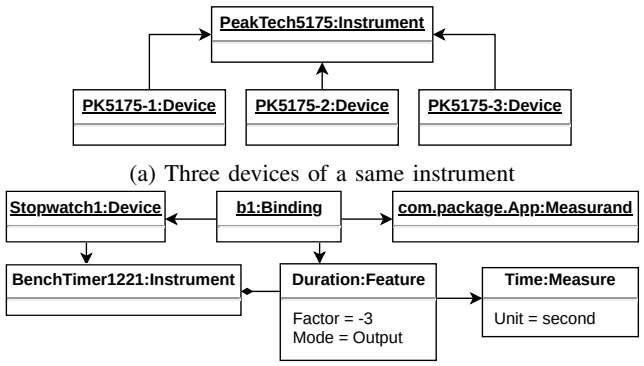
SMM also enables the specification of relationships between measures and measurements, the definition of units of measure, dimensions, measurement scope, measurement accuracy, and so on. Our SAN meta-model embeds several concepts of SMM. Relying on such standard offers a better reusability and compatibility for our approach.

2) SAN Meta-Model: The root of our SAN meta-model is the `Network` element. It defines the address of the server, later used by the devices of the system. `Network` enables the specification of the whole system structure through a set of `Measurands`. Each `Measurand` can be seen as individual equipment, nested places, or zones in which instruments are installed, and performing measures on. This hierarchical structure corresponds to the characterization of physical environment used by Javier Muñoz et al [22].

`Network` also includes a list of `Measure` elements. Each `Measure` specifies a type of measurement provided by a network `Instrument` (e.g. a sensor/actuator of the system). Both `Measure` and `Measurand` notions in our SAN meta-model are equivalent to the ones available in the *Structured Metrics Meta-model* [25]. `Measurands` are related to `Devices` and `Features` using the `Binding` element. Each `Binding` links a single `Measurand` to a single `Device` and a single `Feature`, in order to define where, which entity, and how it is gathering data in the system. `Devices` define the physical sensors and actuators deployed in the system.

`Instruments` and `Devices` are closely related: the `Instrument` defines the model of the sensor (or actuator), whereas the `Device` defines its physical instance in the system. Figure 3a proposes an example of this concept: the instrument `PeakTech5175` is a sound-level meter. Three devices of the same sound-level meter (e.g. `PT5175_1`) are deployed. `Instruments` are defined by a `name`, a list of `Features`, and a list of `Parameters`. A `Feature` has a `Mode`. `Mode` can be either `input` or `output`, which defines the communication mode of the related `Instrument`. In fact, actuators are basic instruments that follow orders sent by a monitoring application. Thus, their related `Features` have the `Mode` set to `Input`. On the contrary, sensors send measurements to a monitoring application, hence their `Mode` is `output`. The `Parameter` category corresponds to the elements that vary among different devices of the same `Instrument`, such as identifiers of IoT devices, security tokens, etc. The `Argument` entity available in `Devices` affects a value to this parameter. All the `Devices` related to the same `Instrument` comply with its `Features`, and every `Feature` can be related to a `Measure`. A `Feature` can apply a `factor` to its related measure.

Figure 3b illustrates the modeling of a measurement to be performed: a remote chronometer is used to measure the execution time of a program, represented by the measurand `com.package.App`. The measuring device is named `StopWatch1` and complies with the instrument `BenchTimer1221`. This instrument has a feature `Duration`, that applies a factor of -3 to the `second` unit given by its corresponding `Time` measure, meaning that this



(a) Three devices of a same instrument

(b) Feature for millisecond measurements

Fig. 3: Excerpts of SAN models

feature records milliseconds. Finally, a binding shows that the `Duration` feature of `StopWatch1` measures the execution time of `com.package.App`.

`Process`es define computation units that can be triggered when specific events occur. The `Trigger`'s role is to specify in which situation the `Process` shall start. `Triggers` are abstract, but extended by `Events`. An `Event` is related to a `Device` and a `Feature` by the means of a `Binding`, and can launch a `Process` when the user-specified event happens. The fact that the `Trigger` class is abstract makes possible to extend it to other kinds of triggers such as background tasks or finite state machines.

In conclusion, guided by a high level of abstracted design as shown in Fig 2, our SAN meta-model will help developers to implement their desired logic. It does not describe how the computations are performed, but when these computations can be started or triggered. The way the computations are performed is not important for mapping the SAN model to an IoT-based monitoring platform. However, the reusability and adaptability of MDE would enable an extension of this existing SAN meta-model, with the descriptions of such computations.

III. Monitoring platform

In realistic scenarios, most of SAN will be too complex to manage without software-support. The majority of monitoring platforms for CPS consist of web services enabling the management of the entities in the network [31] (configuration, connection, disconnection, etc...). Two types of TCP-based protocols are generally used to support communication within the CPS. The first one is the Message Queuing Telemetry Transport (MQTT) protocol to support communications from devices to applications via the publish-subscribe messaging paradigm (widely used between gateways and platforms within IoT architectures). It is lightweight, mature, and require little amount of code to be functional. For those reasons it has been heavily studied, and used, in both industrial and research worlds [16], [33], [20]. The second one is the HTTP protocol for enabling communication between the applications of the CPS via the request-response messaging paradigm [9].

Therefore, we design and prototype Emit[1], an open-source software for managing SAN. A first prototype of a monitoring application was used with power sensors measuring the energy consumption of Java programs [29]. We have created Emit based on that work in order to automate its execution and its configuration by integrating it in our approach. Moreover, Emit can now consider all kinds of MQTT clients, with an HTTP endpoint, and analysis features through *Callbacks*. It is composed of two layers. One relies on the MQTT protocol for communicating with the devices of the CPS it monitors. The other one relies on the HTTP protocol, and is focused on configuring the entities of the CPS that Emit listens to. These web APIs has been testified by our partners from the Measure project[2] in developing advance web applications, such as tools for security analysis and data mining.

A bird's eye view of Emit is shown in Figure 4a. It manages a set of `Clients` which publish/subscribe `Messages`, and a set of `Brokers` which conditionally dispatch `Messagess`

[1]Online repository of Emit: https://github.com/jeromerocheteau/emit
[2]The MEASURE project. http://measure.softeam-rd.eu/related-tools

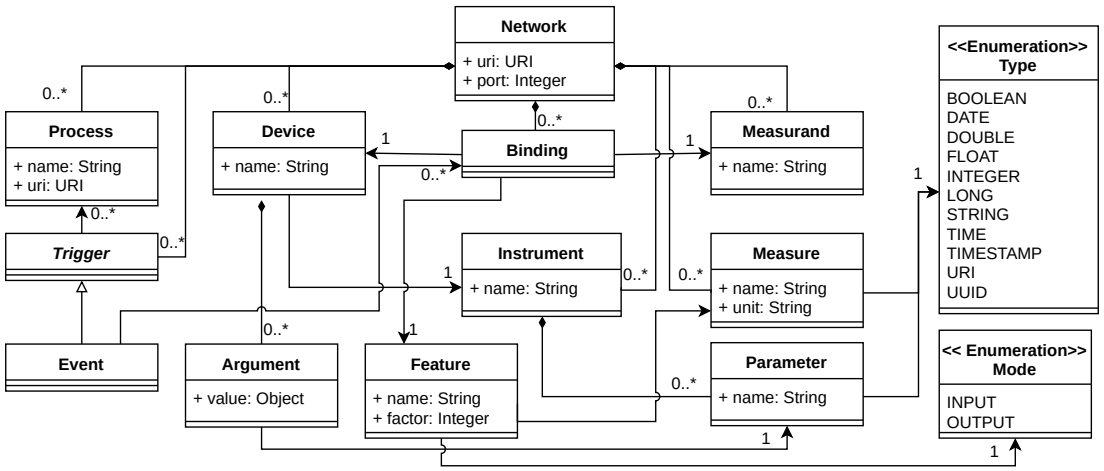


Fig. 2: Sensor and Actuator Network Meta-model

to Clients. It enables a set of standard Controls over Clients, such as Connect to a broker, Subscribe, and Publish to a topic. Last, to have finer-control over clients, EMIT also manages a set of Callback, i.e., functions to be called when clients receive messages.

1) Client Management: The first web API supported by EMIT has the purpose of managing MQTT clients. This implies being able to create, update, delete and retrieve MQTT clients among the instances of Brokers. After providing connection settings and credentials, MQTT clients can be connected to existing Brokers, in order to subscribe, unsubscribe, and/or publish on specific topics. One notable feature of EMIT is that it maintains an inner pool of registered clients that are trusted by EMIT. Foreigner clients need to correspond to a registered client in EMIT in order to be authenticated and to perform management operations via EMIT.

2) Client States Control: The second web API available in EMIT enables the management of the clients states. For each client, this implies being able to: (1) Connect and disconnect the client to a MQTT broker, (2) Subscribe and unsubscribe the client to a MQTT topic, (3) Publish a message to a topic.

Those operations are performed using existing MQTT libraries (https://mosquitto.org/). All performed modifications are logged by EMIT. For instance, calling the web API that connects a MQTT client to a broker creates a new Connect instance in the database with the started property defined by the current timestamp. When it disconnects, the web API updates the instance for its stopped attribute with the current timestamp. This works the same way for the Subscribe and Publish entities.

3) Callback edition: Callbacks are behaviors that are triggered when a specified event occurs. In EMIT, the web API enables the creation and edition of callbacks. Callbacks are created by Clients, and trigger events when a message corresponding to this Callbacks is received on a topic listened by the Client. In EMIT, we can create four types of atomic callbacks for MQTT message processing (as shown in Figure 4b):

- TopicCallback returns true if and only if the message topic matches the pattern attribute it contains.
- TypeCallback returns true if and only if the message payload can be cast to one of the types.
- StorageCallback enables the storage of a message content, into a given collection of underlying database of EMIT. Hence it means that the message persistence is programmatically defined using the callback edition services, with their attachment to a client.
- FeatureCallback returns true if and only if the message payload satisfies the logical condition expressed via Type and Symbol.

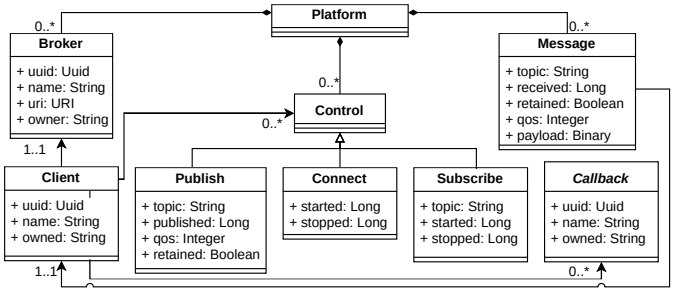
These atomic callbacks can be composed into more complex callbacks using GuardCallback for a conditional case analysis structure. Each GuardCallback specifies: 1) a test callback to check the condition; 2) a success callback to specify what should happen when the check passes; and 3) a failure callback to specify the case when the check does not pass. EMIT also enables the definition of callback when events at the user level occur.

- ConnectCallback corresponds to the connect / disconnect state control.
- SubscribeCallback corresponds to the subscribe / unsubscribe state control.
- AttachCallback corresponds to the attach/detach state control. The duality in client state controls (e,g. connect/disconnect, subscribe/unsubscribe and attach/detach) is defined by a Boolean property labeled enable.
- PublishCallback corresponds to the publish state control, where its message property defines a message payload.

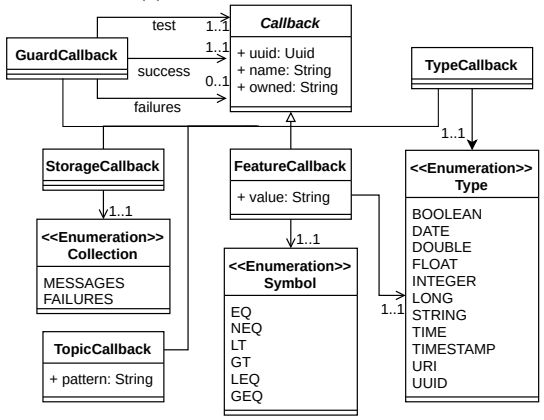
Finally, EMIT provides a web API for MQTT messages retrieval. This returns chunks of messages for a given MQTT client, on a specified topic. The service can be customized in order to only retrieve messages in a given time-span, by giving the API temporal bounds.

IV. MAPPING BETWEEN SAN META-MODEL AND EMIT

In previous sections, we presented our SAN meta-model to provide a way to model the diversities in SAN (Section II). We also presented EMIT, a generic CPS monitoring platform relying on Web-APIs for configuration and management (Section III). This separation of designs allows experts from both SAN modelling and SAN management to work independently, thereby increasing productivity and quality of software development. In this section, we describe a mapping between



(a) EMIT Core Meta-model



(b) EMIT Core callback meta-model

Fig. 4: Excerpts of EMIT meta-model

the two worlds, that allows domain experts in each world to exchange their information. Our mapping is documented in a model transformation fashion, which aims at generating entities inside EMIT, hence enabling monitoring the CPS corresponding to the SAN model. In addition, since the web APIs of EMIT can only be queried through HTTP requests, we briefly illustrate the corresponding HTTP queries generated out of the SAN model, that facilitate the usage of EMIT.

Notice that our mapping is not a **total** function, in the sense that several elements in the SAN meta-model does not have to be created in EMIT to enable the monitoring of the CPS. As an example, `Measurand` elements are physically bound to the devices performing measures on, and are modelled in the SAN models. Adding them in EMIT is not necessary, since we only are only interested in monitoring the `devices` performing the measurements.

In this paper we map our SAN meta-model to EMIT, however the same approach could be applied to models conforming to other meta-models, such as SensorML or SSN, using our SAN meta-model as a pivot.

1) Network Transformation: At the root of our mapping, we translate each `Network` element of a SAN model into a `Broker` entity for EMIT. From such `Broker` element, we subsequently launch a code generation of `HTTP POST` request in order to create a virtual broker in EMIT. The idea is to enable and follow the methodology of the MQTT protocol, i.e. using a `Broker` to manage communications of clients, later generated.

2) Feature Transformation: The `Mode` attribute defines if either the `Feature` sends data (i.e., is a sensor), or receive data (i.e., is an actuator). In both cases, EMIT has to listen to the messages going from or to the `Feature`. For that purpose, SAN's `Feature` are mapped to Client entities within EMIT. Later, EMIT can then subscribe to topic corresponding to the mapped `Feature` in order to monitor it. This topic is generated by concatenating the `name` attributes of the network, device and binding, and using backslash as a separator. We assume that the monitored sensors and actuators are publishing and subscribing, respectively, to this topic.

Furthermore, the `Measure` to which the `Feature` is bound to indicates which datatype this `Feature` is supposed to receive (or send). A `GuardCallback` is added to this generated client to ensure the data sent conforms to the right datatype. This `GuardCallback` checks the type of the message using a `Type Callback`. Two `StorageCallbacks` are then attached to this `GuardCallback`. If the type of the message conforms to the `Measure`, then the messages can be persisted in a "message" collection with the first `StorageCallback`. Otherwise, it is persisted in a "failure" collection with the second `StorageCallback`. After the creation of the client, EMIT can monitor the messages sent to, or by, this client and eventually add callbacks to it according to the needs.

3) Event Transformation: So far, only the events attached to features with the `output` mode (i.e., events triggered when a specific value is sent by a sensor) are considered by this transformation. In fact, actuators are not necessarily controlled through MQTT, and for that reason, mapping them into EMIT is out of this work scope. This Event transformation shows that it is possible to automatically create within EMIT: (1) a Callback element that corresponds to this process, and (2) a Client that subscribes to the topic on which a device broadcasts its measurement data.

The `Event` transformation has two steps. Firstly, it verifies that the `uri` of each `Process` actually refers to an existing entity within the execution environment. This can be done using software components provided as third-party libraries and which comply with the MQTT callback API provided by the Eclipse Paho library (Eclipse Paho. https://www.eclipse.org/paho/), used within EMIT. Secondly, it transforms every `Event` bound to `output Features` into a `GuardCallback`. The test of this `GuardCallback` is defined by a `FeatureCallback`, which checks if the output of the `Feature` verifies the specified condition. The success of this `GuardCallback` is defined by a hand-made callback launching the `Process` behaviour. Such transformation provides flexibility: custom MQTT callbacks can be developed and integrated into the monitoring application later. However, it does not ensure a full control over the provided callbacks which have to be user-defined in order to launch `Process`es.

V. APPLICATION

In this section, we apply our approach on a case study, following the three steps presented in Figure 1. This case study is designed for a smart cooling system of an IT infrastructure: a thermometer measures the temperature in a building, when this temperature reaches a given threshold, a cooling process starts. The infrastructure is modeled conforming to our SAN meta-model (Section II) as shown in Figure 5[3]. We then describe how it is mapped to EMIT for enabling its monitoring. The goals are: (1) To show the possibility of pipelining our technologies for developing and monitoring CPS. (2) To provide this case study as a reference that allows SAN users to define their own mappings in order to interact with EMIT.

1) Modeling a case study: The IT infrastructure is composed of three locations in which measurements are performed: two floors `Floor1` and `Floor2` located inside a building `Building1`. One thermometer device (`TILM35B`) is attached to the whole building for measuring its current temperature. It refers to the `TILM35` element, which describes the measuring features it uses. We labelled it `CurrentTemp`, as it produces a temperature with the Celcius unit. A binding labelled `b3` associates together `Building1`, `CurrentTemp` and `TILM35B`. This models that the thermometer `TILM35B` is deployed in `Building1`, and is measuring its temperature (in Celcius) with the temperature sensor it embeds. Furthermore, a power meter (measuring watts) is attached to each floor for tracking their respective power consumption. They are

[3]The figure is simplified to fit the level of conciseness needed for this paper. In particular, reference names are omitted, containment relationship of Network entities hidden, and only entities that are demonstrated in the paper are shown.

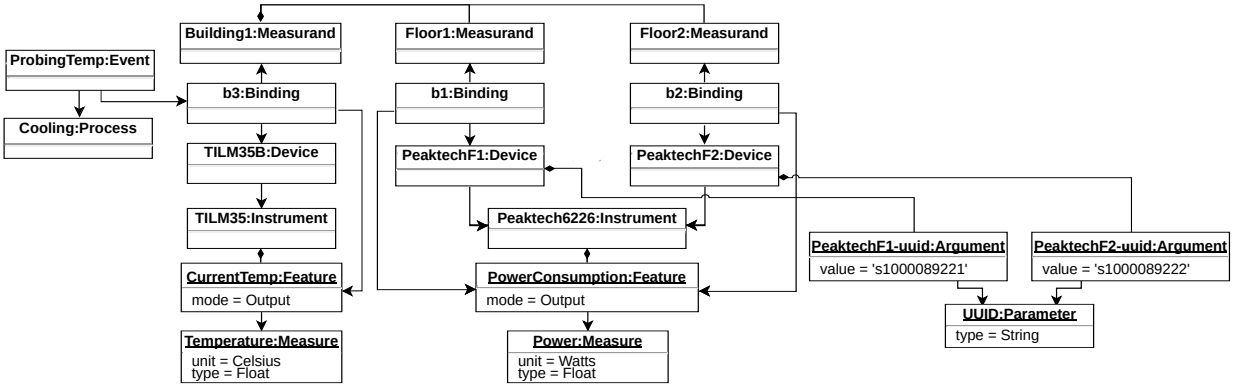


Fig. 5: Cooling System Modeling for IT Infrastructure using SAN model

modeled by the `PeaktechF1` and `PeaktechF2` elements, and refer to `Peacktech6226`, which describes its power measuring output device. The `b1` binding associates together `Floor1`, `PeaktechF1` and `PowerConsumption` (`b2` binds `Floor2`, `PeaktechF2` and `PowerConsumption`, respectively). Finally, an entity "`ProbingTemp`" is referring `b3`. This entity triggers a `Cooling` process as soon as the temperature measured by the `TILM35B` is too important. We modeled this SAN, as shown in the first step of Figure 1, the next step it to automatically generate its monitor application.

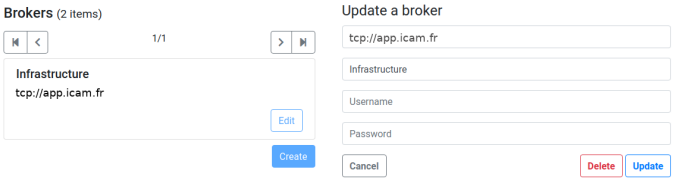
2) Mapping to Emit: In order to enable monitoring, the modeled SAN has to be added into EMIT. Running the transformation (as described in Section IV) on the input SAN model generates several entities in EMIT.

Since EMIT relies on the MQTT protocol (as described in Section III), the first step of the transformation adds into EMIT the broker used in the SAN. As shown in Figure 6a, a broker labeled `Infrastructure` is added to EMIT, which corresponds to the `Infrastructure` element in the SAN model. Subsequently, it allows EMIT to subscribe to different topics that this broker hosts, thereby monitoring the sensors and actuators available in the system.

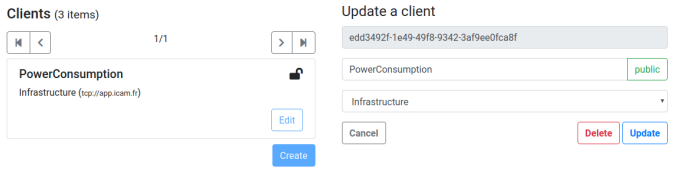
The second step of the transformation maps the three devices to EMIT as Clients. Figure 6b shows the interface of EMIT that displays a client generated for the `PowerConsumption` feature of `PeaktechF1`. The user can switch between clients by using the left panel. The right panel shows different parameters of this feature that the user can edit. As we can see in the figure, `PowerConsumption` is declared as a public client, which means it can be seen by any EMIT user (private client would only be seen by the user that added it). The last text field shows the Broker in EMIT, to which this client is connected. Once registered to a broker in this way, client's data is sent and monitored by EMIT. This allows further analysis or computations to be performed.

The third step of the transformation maps the event in the SAN model into EMIT. For instance, the `ProbingTemp` event is transformed into a set of callbacks. First, a guard is created into EMIT. This guard is a callback, activated every time a message is sent by the corresponding client. The test of this guard checks if the power measured by the `PowerConsumption` sensor is above a specified threshold. A success from this guard launches the cooling process linked to the event. Figure 6c shows the interface of callbacks in our example and the guard generated from the SAN model.

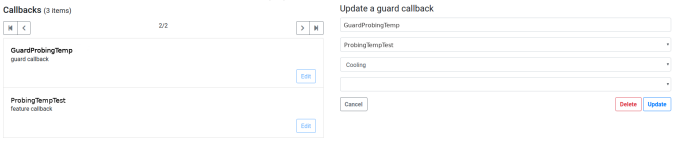
Upon this point, the smart cooling system is added in EMIT, as shown in the second step of Figure 1.



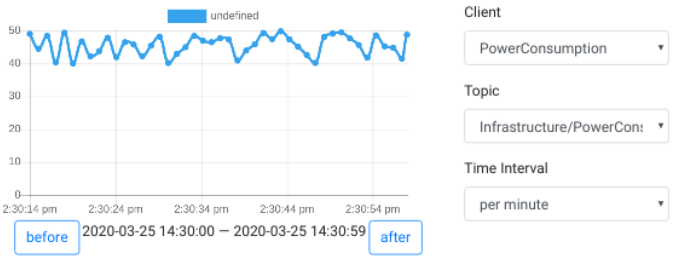
(a) Broker view



(b) Client view



(c) Callback view



(d) EMIT monitoring view

Fig. 6: Screen captures of EMIT running

3) Monitoring with Emit: Finally, EMIT can effectively be used to monitor the three clients created, as shown by the third step of Figure 1. This enables several functions: (1) Adding, removing, and managing more `Callbacks` for the clients. (2) Easily accessing the messages sent and received by the clients, using the RESTful API provided by EMIT. (3) Providing real-time metrics with graphical display through the front-end of EMIT, as shown in Figure 6d. Monitoring this system within EMIT enables different analysis and computation on the data gathered by the sensors of the system. E.g., energy metrics and statistics could be computed using both the power-meter and timestamp values providing the energy consumption of the building during the wanted duration.

This case study shows how we generate EMIT entities from our SAN model, in a real-life situation. This approach is automatized using a model transformation, ensuring a conformity between EMIT's configuration, and the SAN model. Compared to standard approaches, modifying the system would not break the monitoring functions of EMIT: re-running the transformation would update the entities of EMIT and hence maintain the monitoring, for a better reliability and maintainability.

The transformation in this example is implemented as a Model-To-Text (M2T) transformation, and is available on Github[4]. The SAN model is defined using the Eclipse Modeling Framework [32], and the M2T implemented using Acceleo [23], which generates a set of HTTP requests in the CURL format[5].

VI. Related Work

Applying MDE to SAN has been extensively investigated according to the literature([3], [28], [30]). In this section, we discuss related model-driven solutions that facilitate the design, analysis, and development of SAN.

Janowicz et al. propose SOSA[17], a lightweight ontology for Sensors and Actuators. It enables the description of Sensors and Actuators, the device on which measurements are performed, as well as the results it outputs. However, as it is lightweight, it is rather generic and coarse-grained, and is not meant to specify processes. Our SAN meta-model tackles those issues.

Ben Maissa et al. design a large set of primitives to specify physical deployment of sensors/actuators, or to specify different behaviors of sensors/actuators according to their location within the network [5]. The main limitation is that the result designs lack of modularity, which makes maintenance difficult.

Doddapaneni et al. separately model the architecture of SAN, the low-level hardware specification of its nodes, and the physical environment of deployed nodes [13]. This kind of separation of concerns is similar to our way of modularizing design of physical and software parts when design CPSs, but in their work they model CPSs for simulation purposes, whereas our work focuses on monitoring running systems.

On the same track of enhancing design modularity, Dantas et al. design a language called LWiSSy to encourage domain specific knowledge in design of SAN [10]. Specifically, the language is organized in three views: structural, behavioral, and optimization. Each view is to specify a distinct aspect of CPS under development by different domain experts. The DSL proposed offers little expressivity on the data measured by sensors. Our SAN meta-model enables a finer specification of those, in order to enable their monitoring.

Vidal et al. design MindCPS [34], which provides modelling primitives to explicitly model autonomic behaviours of system under development. Then, it can enable model transformations to automatically generate Java and SQL code for SAN (e.g. control loops). Compared to our SAN meta-model, MindCPS's meta-model is more complex, which is composed of built-in classes for data filtering, error handling, etc. This design choice results a wider scope for code generation. However, in our opinion, our SAN meta-model is more flexible, since when certain elements (e.g. complex data processing) cannot be expressed by primitives, we provide generic elements in our meta-model to model them systematically (which lacks support in MindCPS). Moreover, in terms of real-time data processing, MindCPS provides a more mature solution that publishes them on the MQTT broker in order to provide real-time feedback, whereas we store them all into database. This feature of MindCPS is what we intend to extend for our approach in the near future.

Harrand et al. define the ThingML framework to support design and implementation of CPS [15]. It includes a textual extendable modeling language, and support customizable code generation targeting heterogeneous platforms (e.g. [6]). A key distinct feature of ThingML to EMIT is that it uses state machine to allow more precise modeling of the behavior of CPS, which subsequently enables more mature code generation from designed state machine. However, ThingML does not extensively consider callbacks as we do in EMIT. Therefore, We are currently investigate how to naturally bridge between the two worlds to enhance interoperability of designed methodologies / technologies, e.g. by sub-classing ThingML state-machine designs as a special kind of *callback* in EMIT.

Two mature monitoring platforms for MQTT devices can be considered: IBM's Watson IoT[6] and Eclipse MQTT Spy[7]. The first one is a powerful private cloud-based solution, also driven through HTTP. Compared to Watson IoT, EMIT is Free, Open Source and can be self hosted and adapted. Eclipse MQTT Spy offers better message filtering and processing services than EMIT, but no HTTP endpoint for remote control and monitoring.

VII. Conclusion and Perspectives

In this work, we present a meta-model of SAN, enabling the co-design of the structural and behavioral parts of such network. We also design and implement an open-source Web-API, namely EMIT, for monitoring sensors and actuators networks, and providing an endpoint to the Information Infrastructure for running analysis and computations on the data

[4]https://github.com/veriatl/emit-san-metamodel

[5]https://curl.haxx.se/

[6]https://www.ibm.com/cloud/watson-iot-platform

[7]https://www.eclipse.org/paho/components/mqtt-spy/

gathered from the network. We orchestrate our SAN meta-model and EMIT via model transformation to allow them work hand in hand. We apply our whole approach on a smart cooling system case study. The result shows that by leveraging MDE for generating an MQTT-based SAN monitoring application, we can foster reusability of CPS design. The abstract view of the system eases its understandability, providing a safe way of structuring the SAN network. Furthermore, the standard format provided by EMF enables reusability and maintainability, for multiple purposes such as refactoring, refining, code generation. Finally the mapping used for transforming our SAN model to the EMIT model can be easily customized, in order to adapt to the client's needs.

Our future work will focus on extending the compatibility of our approach with existing modeling approaches. MDE allows technology transfer with other meta-models and we plan to consider how to transform ThingML models [15] or SensorML to and from SAN models, for instance. In addition, we will consider how to transform the measurements obtained into generic SMM models.

Another perspective would be to directly extends our meta-model and transformations with existing model-based mature technologies, such as ThingML and SMM. In fact, their meta-models have been written using standard formats, and based on MDE principles, they could be associated with our SAN meta-model. Combining models of different aspects of the system would help providing a holistic view of CPS.

Acknowledgements. The research presented in this paper is funded by the ITEA3 Project no. 14009 called MEASURE (from 1st December 2015 to 31st August 2019).

REFERENCES

[1] Akyildiz, I.F., Kasimoglu, I.H.: Wireless sensor and actor networks: research challenges. Ad hoc networks **2**(4), 351–367 (2004)

[2] Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Computer networks **38**(4), 393–422 (2002)

[3] Anwar, M.W., Azam, F., Khan, M.A., Butt, W.H.: The applications of model driven architecture (mda) in wireless sensor networks (wsn): Techniques and tools. In: Future of Information and Communication Conference. pp. 14–27. Springer (2019)

[4] Baheti, R., Gill, H.: Cyber-physical systems. The impact of control technology **12**(1), 161–166 (2011)

[5] Ben Maissa, Y., Kordon, F., Mouline, S., Thierry-Mieg, Y.: Modeling and analyzing wireless sensor networks with verisensor: An integrated workflow. In: Transactions on Petri Nets and Other Models of Concurrency VIII. Springer (2013)

[6] Berrouyne, I., Tisi, M., Mottu, J.M., Adda, M., Royer, J.C.: Cypriot: Framework for modelling and controlling network-based iot applications. In: Proceedings of the 34th ACM/SIGAPP SAC. ACM (2019)

[7] Bézivin, J.: On the unification power of models. Software & Systems Modeling **4**(2), 171–188 (2005)

[8] Botts, M., Robin, A.: Opengis sensor model language (sensorml) implementation specification (ogc 07–000) (2007)

[9] Daniel, L., Kojo, M., Latvala, M.: Experimental evaluation of the coap, http and spdy transport services for internet of things. In: International Conference on Internet and Distributed Computing Systems. pp. 111–123. Springer (2014)

[10] Dantas, P., Rodrigues, T., Batista, T., Delicato, F.C., Pires, P.F., Li, W., Zomaya, A.Y.: Lwissy: A domain specific language to model wireless sensor and actuators network systems. In: 4th International Workshop on Software Engineering for Sensor Network Applications. pp. 7–12. IEEE (2013)

[11] Daugherty, P., Banerjee, P., Negm, W., Alter, A.E.: Driving unconventional growth through the industrial internet of things. Accenture Technology (2015)

[12] Derler, P., Lee, E.A., Tripakis, S., Törngren, M.: Cyber-physical system design contracts. In: Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems. pp. 109–118 (2013)

[13] Doddapaneni, K., Ever, E., Gemikonakli, O., Malavolta, I., Mostarda, L., Muccini, H.: A model-driven engineering framework for architecting and analysing wireless sensor networks. In: Proceedings of the 3rd International Workshop SESENA. pp. 1–7. IEEE Press (2012)

[14] Génova, G.: What is a metamodel: the omgs metamodeling infrastructure. Software and Systems Modeling **4**(2), 171–188 (2005)

[15] Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: ThingML: a language and code generation framework for heterogeneous targets. In: 19th ACM/IEEE International Conference MoDELS. ACM (2016)

[16] Hunkeler, U., Truong, H.L., Stanford-Clark, A.: Mqtt-sa publish/subscribe protocol for wireless sensor networks. In: 3rd International Conference on Communication Systems Software and Middleware and Workshops. IEEE (2008)

[17] Janowicz, K., Haller, A., Cox, S.J., Le Phuoc, D., Lefrançois, M.: Sosa: A lightweight ontology for sensors, observations, samples, and actuators. Journal of Web Semantics **56**, 1–10 (2019)

[18] Kim, K.D., Kumar, P.R.: Cyber–physical systems: A perspective at the centennial. Proceedings of the IEEE **100**(Special Centennial Issue), 1287–1308 (2012)

[19] Kubler, S., Främling, K., Buda, A.: A standardized approach to deal with firewall and mobility policies in the iot. Pervasive and Mobile Computing **20** (2015)

[20] Light, R.A.: Mosquitto: server and client implementation of the mqtt protocol. The Journal of Open Source Software **2**(13), 265 (2017)

[21] Muñoz, J., Pelechano, V.: Building a software factory for pervasive systems development. In: International Conference on Advanced Information Systems Engineering. pp. 342–356. Springer (2005)

[22] Muñoz, J., Valderas, P., Pelechano, V., Pastor, O.: Requirements engineering for pervasive systems. a transformational approach. In: 14th IEEE International Requirements Engineering Conference (RE'06). pp. 351–352. IEEE (2006)

[23] Musset, J., Juliot, É., Lacrampe, S., Piers, W., Brun, C., Goubet, L., Lussaud, Y., Allilaire, F.: Acceleo user guide. See also http://acceleo.org/doc/obeo/en/acceleo-2.6-user-guide. pdf **2**, 157 (2006)

[24] Neuhaus, H., Compton, M.: The semantic sensor network ontology. In: AGILE workshop on challenges in geospatial data harmonisation, Hannover, Germany. pp. 1–33 (2009)

[25] Object Management Group, O.M.G.: Structured metrics metamodel, https://www.omg.org/spec/SMM/1.2/

[26] Van de Panne, M., Fiume, E.: Sensor-actuator networks. In: Proceedings of the 20th annual conference on Computer graphics and interactive techniques (1993)

[27] Pavithra, D., Balakrishnan, R.: Iot based monitoring and control system for home automation. In: 2015 global conference on communication technologies (GCCT). pp. 169–173. IEEE (2015)

[28] Priego, R., Armentia, A., Estévez, E., Marcos, M.: Modeling techniques as applied to generating tool-independent automation projects. at-Automatisierungstechnik **64**(4), 325–340 (2016)

[29] Rocheteau, J., Gaillard, V., Belhaj, L.: How green are java best coding practices?. In: SMARTGREENS. pp. 235–246 (2014)

[30] Rodrigues, T., Dantas, P., Pires, P.F., Pirmez, L., Batista, T., Miceli, C., Zomaya, A.: Model-driven development of wireless sensor network applications. In: IFIP 9th International Conference on Embedded and Ubiquitous Computing. IEEE (2011)

[31] Singh, K.J., Kapoor, D.S.: Create your own internet of things: A survey of iot platforms. IEEE Consumer Electronics Magazine **6**(2), 57–68 (2017)

[32] Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)

[33] Thangavel, D., Ma, X., Valera, A., Tan, H.X., Tan, C.K.Y.: Performance evaluation of mqtt and coap via a common middleware. In: Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP). pp. 1–6. IEEE (2014)

[34] Vidal, C., Fernández-Sánchez, C., Díaz, J., Pérez, J.: A model-driven engineering process for autonomic sensor-actuator networks. International Journal of Distributed Sensor Networks **2015**, 18 (2015)

[35] Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE software **31**(3), 79–85 (2014)