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

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

To cite this version:

Thibault Béziers La Fosse, Zheng Cheng, Jérôme Rocheteau, Jean-Marie Mottu. Model-Driven Engineering of Monitoring Application for Sensors and Actuators Networks. Software Engineering and Advanced Applications, Aug 2020, Portorož, Slovenia. hal-02612730v2

HAL Id: hal-02612730
https://inria.hal.science/hal-02612730v2
Submitted on 29 Sep 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 EMISS a monitoring platform for sensors and actuators networks. Third, it presents a transformation from models of sensors and actuators networks to automatically generate EMISS 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], [20]. 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 EMISS, the monitoring platform we propose. Third, using the MQTT protocol, EMISS 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 heterogenous 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. 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 specificiation 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 Stopwatch 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
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 [11] (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 [10], [13], [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. http://measure.softeam-rd.eu/related-tools

1Online repository of EMIT: https://github.com/jeromerocheteau/emit

Fig. 2: Sensor and Actuator Network Meta-model

Fig. 3: Excerpts of SAN models
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 Callbacks, 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 design 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
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 Processes.

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 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

3The 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.
modeled by the PeaktechF1 and PeaktechF2 elements, and refer to Peaktech6226, 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 5, 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.

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

Fig. 6: Screen captures of Emit running

(a) Broker view

(b) Client view

(c) Callback view

(d) Emit monitoring view
3. **Monitoring with Emit:** Finally, Emit can effectively be used to monitor the three clients created, as shown by the last 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 client, 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 2. Monitoring this system within Emit enables different analysis and computation on the data gathered by the collectors 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 service 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 automated 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. The SAN model is defined using the Eclipse Modeling Framework, and the M2T implemented using Accceleo, which generates a set of HTTP requests in the CURL format.

VI. RELATED WORK

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

Janowicz et al. propose SOSA, 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. 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. 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. 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, which provides modeling primitives to explicitly model autonomic behaviors 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 metamodel 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. It includes a textual extendable modeling language, and support customizable code generation targeting heterogeneous platforms (e.g., APIs). 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 investigating how to naturally bridge between the two worlds to enhance interoperability of designed methodologies/technologies, e.g., by subclassing 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 and Eclipse MQTT Spy. 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.

---

https://github.com/veriatl/emit-san-metamodel
https://curl.haxx.se/
gathering 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 extend 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