A Model-Driven Methodology to Accelerate Software Engineering in the Internet of Things

Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Massimo Tisi

A Model-Driven Methodology to Accelerate Software Engineering in the Internet of Things

Imad Berrouyne[1], Mehdi Adda[2], Jean-Marie Mottu[3], and Massimo Tisi[3]

***Abstract*—The Internet of Things (IoT) aims for connecting Anything, Anywhere, Anytime (AAA). This assumption brings about several software engineering challenges that constitute a serious obstacle to its wider adoption. The main feature of the Internet of Things (IoT) is genericity w.r.t the variability of software and hardware technologies. Model-Driven Engineering (MDE) is a paradigm that advocates using models to address software engineering problems. It can help to meet the genericity of the IoT from a software engineering perspective. Existing MDE approaches for the IoT focus only on modeling the internal behavior of things but lack a comprehensive approach dedicated to network modeling. In the present paper, we introduce a network-oriented methodology based on MDE to unify the IoT's heterogeneous concepts. Fundamentally, we avoid the intrinsic heterogeneity of the IoT by separating the network's specification (i.e., the things, the communication scheme, and its constraints) from its concrete implementation (i.e., the low-level artifacts such as source code and documentation). Technically, the methodology relies on a model-based Domain-Specific Language (DSL) and a code generator. The former enables the modeling of the network's specification, and the latter provides a procedure to generate the low-level artifacts from this specification. Our results show that this methodology makes IoT's software engineering more rigorous, helps prevent bugs earlier, and saves time.**

***Index Terms*—Internet of Things, Software Engineering, Model-Driven Engineering, Model Transformation, Policy Enforcement, Code Generation**

I. Introduction

The IoT aims for connecting Anything, Anywhere, Anytime (AAA) [1]. In particular, connecting things of different sizes ranging from bacterias [2] to supercomputers. Manifestly, each thing has its requirements. For instance, a tiny sensor may require a Constrained Application Protocol (CoAP) [3] client to connect to the World Wide Web (WWW), because of its limited resources, while a laptop may require a standard Hypertext Transfer Protocol (HTTP) client. Indeed, the premises of the IoT are that (1) anything with computing power (2) can connect to the Internet. While the former premise (1) is hardware-related, the latter (2) could be tackled using an appropriate software engineering approach.

Software engineering for the IoT is hard [4]. On the one hand, many stakeholders are involved (e.g., Security, Business, Network), each using its own tools and methods [5]. On the other hand, heterogeneity of software technologies (e.g., languages, protocols) [6] causes, among other things, an interoperability problem by hindering communication between things [7]. Commonly, a typical IoT application contains multiple computing platforms, languages, and protocols from various ranges. Besides, a new thing emerges every day, with often non-standard proprietary technologies. Nevertheless, although it is problematic, heterogeneity constitutes the differentiating factor between the IoT and the conventional Internet. Indeed, numerous studies [8], [9], [10], [11], [12] suggest that this heterogeneity is necessary to connect things from different ranges by means of different protocols.

The IoT generates much hype; still, only a few software engineering approaches have been proposed to meet its requirements. Today, the existing approaches are time-consuming, require a good deal of expertise, lack separation of concerns [13], and lead to poorly tested and insecure IoT applications [14]. In fact, heterogeneity provokes the involvement of more human resources and expertise than usual. Hence, most companies, often with inappropriate or limited human resources, fail to respond adequately; this may result in flawed applications that sometimes lead to large-scale network attacks such as Mirai and Persirai [15], [16].

Clearly, the IoT needs a novel software engineering approach adequate to its requirements. MDE is a promising paradigm having the potential to meet some of these requirements. It can help in designing robust and reliable IoT applications by separating the specification (source of intent) from the implementation (source of heterogeneity). Particularly by enabling the design of a complete IoT application in a unified manner using models on the one hand and interpreting these models using an automatic code generator on the other. For instance, the Unified Modelling Language (UML) [17] is a generic modeling language to design, using models, any Object-Oriented (OO) application. UML models are used for illustration purposes, code generation [18] or test cases generation [19] to cite a few. In the present paper, we are heading towards a similar goal, i.e., enabling the design of an IoT application using models and the automatic generation of its low-level software artifacts.

This paper extends two of our conference papers [20], [7] by introducing a comprehensive software engineering methodology for the IoT, with clearly defined modeling concerns (Section IV-A) and software engineering tasks (Section III), a conflict detection mechanism (Section V-F), additional code generation procedures (Section VI), and a quantitative evaluation (Section VII). Concretely, the methodology enables a) the modeling and validation of a network of things, b) the ability to constrain this network with a policy, and c) the generation of the network's artifacts by means of state-of-the-art MDE

[1] SVV Team, SnT, Université du Luxembourg — 29, avenue JF Kennedy L-1855 Luxembourg — firstname.lastname@uni.lu

[2] Mathematics, Computer Science and Engineering Dep. University of Quebec At Rimouski, Rimouski, QC, Canada — firstname_lastname@uqar.ca

[3] Naomod Team, IMT Atlantique, LS2N, Nantes, France — firstname.lastname@ls2n.fr

techniques.

For the sake of clarity, we aim to keep this paper as concrete as possible by targeting some specific IoT tasks. The present contribution is structured as follows. Section II provides the background. Section III points out the specific targeted software engineering tasks. Sections IV, V, and VI detail how our methodology contributes to the adequate execution of these tasks. Finally, Sections VII,VIII and IX respectively evaluate the methodology, points out its limitations, and concludes.

II. Related Work

The IoT still faces several obstacles related to its engineering, scalability, and deployment. It lacks a set of best practices to build scalable, robust, and secure applications [21]. In addition, the intrinsic heterogeneity of the IoT creates an interoperability problem between things [22], restricting the ability to achieve seamless smart scenarios [23].

Presently, most of the existing software engineering approaches for the IoT consist of programming each thing to fit the needs of the application [24], [25], [26], [27]. These approaches require a significant amount of time as many skills, ranging from networking to security and programming, are needed. For instance, a typical IoT application may have multiple things based on heterogeneous programming languages, heterogeneous protocols and require some degree of control of the network. These software engineering strategies are less likely to scale, typically in very large networks.

Recently, several model-based approaches emerged to tackle heterogeneity in the IoT. In this respect, many approaches chose a Domain-Specific Language (DSL) to provide a user-friendly interface to specify a network of things [28], [29], [30], [31], [32]. However, most of these approaches target specific usecases, and lack the mechanisms necessary to tackle generic IoT applications. From a conceptual perspective, the MDE literature for IoT draws a distinction between two underlying concepts in the IoT; the concept of *thing* and the concept of *network*. The modeling of a thing consists of using abstract concepts to describe its internal behavior. We can distinguish two ways to achieve it; the first one consists of mapping the behavior into some established formalism such as a statechart [33], [34] or a workflow chart [35], and the second one consists of defining the behavior with an unconventional formalism [36]. The former benefits from the interoperability with the established formal tools, while the latter generally aims to reflect reality using some intuitive concepts. For our methodology, we opted for a statechart-based behavior to model things based on the work of Harrand et al. [33], namely ThingML[1]. This work constitutes our baseline.

The modeling of a network consists of wiring things through their external interfaces to form a network of things. Because of the implicit heterogeneity of the IoT, the wiring is complicated at the code level. Hence, higher abstractions are needed, free from the technical considerations to enable seamless wiring. The existing approaches for network modeling are disparate; some approaches target Wireless Sensor Networks (WSN) [37], [38] (a network of many tiny sensors dedicated to collect data) [39], others target the web of things [40] (i.e., the IoT using existing web technologies), and others target a specific category of IoT applications such as Smarthomes or TinyOS-based things [41], [31], [30], [32]. According to our literature review, we noticed a lack of a full-scale, documented, and open-source approach for modeling a network of things generically. Moreover, only a few approaches [42], [43], [44], [45] leverage the power of MDE for the automatic code generation of a complete network, which can consequently reduce the redundant tasks and help tackle the interoperability problem of the IoT [7].

[1]https://github.com/TelluIoT/ThingML

III. Software Engineering Tasks

The engineering process for an IoT application involves multiple tasks. For simplicity, we point out the issues of four essential engineering tasks with the current state of practice. Further, we show how our model-based methodology can help improve their execution. In the future, our methodology may be extended to more tasks.

A. Task 1: Wiring Heterogeneous Things

The wiring of heterogeneous things is a task that enables the communication between two things that use either different programming languages or protocols. For instance, wiring a thing t_j programmed in Java with a thing t_c programmed in C via a publish and subscribe communication based on MQTT. To be implemented flawlessly, this example may require two experts, one in Java and the other in C, and some degree of synchronization between them to ensure correct wiring of t_j and t_c.

Issues: a) The need of too many human resources and expertise for this simple usecase b) The synchronization between the experts is time-consuming, untraceable, and a vector for the introduction of bugs.

B. Task 2: Smart Scenarios

Smart Scenarios' design is a task that consists of enabling interdependence between things according to some imposed policy. In addition to the requirements of Task 1, this interdependence requires programming each thing independently using its inner concepts to conform with the policy. For instance, a smart scenario may require turning on a Python-based system for heating, ventilation and air-conditioning t_{ac} when a C-based temperature sensor t_{ts} indicates the current temperature. The engineer needs here to add the instructions, non-uniformly, in t_{ts} using Python and in t_{ac} using C to enable their interdependence.

Issues: a) The lack of common concepts for a seamless implementation b) The manual implementation is difficult to scale.

C. Task 3: Controlling Communication

Controlling communication is a task that consists of defining how these presumably heterogeneous things can communicate with each other in the network using some specific policy. For instance, we may need to define rules that govern the

communication flow between adverse things, such as a private camera and a public Webserver.

Issues: a) The lack of common concepts to control the communication flow uniformly b) The manual implementation may lead to information leaks.

D. Task 4: Designing Large Networks

The previous tasks become even more complicated when one needs to create a large network. The traditional software engineering approaches suffer from many redundant tasks in this context, such as wiring each thing separately or writing manually various artifacts (e.g., code documentation, user manual, and access control) although these tasks are based on a unique source information, i.e., the specification of the network. Moreover, in a typical network, these artifacts need to be synchronized at each update.

Issues: a) The redundancy of tasks and difficulty to scale b) The risk of overlooking essential artifacts due to the workload required.

The Proposed Solution

By and large, software engineering for IoT is made difficult by low-level heterogeneity, interoperability problems, difficulty in controlling the network, and overlapping concerns. The root cause of these issues is often difficult to identify through traditional software engineering approaches.

We propose to unravel these issues using various MDE layers, primarily by separating the specification of the network (using unified concepts at the model-level) from its implementation. This separation aims to make these issues more visible and easier to tackle using an appropriate software engineering tool. Indeed, we offer software engineers a DSL to specify a network of heterogeneous things in a unified environment. Then, within this DSL, we provide means to define smart scenarios based on behavioral and temporal factors. Finally, we introduce an extensible code generator capable of interpreting the network specification and generating its low-level artifacts (e.g., code, documentation, and access control rules).

IV. Network Specification

The function of the network specification is to enable the interoperability of heterogeneous things using unified concepts at a higher level. The literature contains mature approaches to model a thing's behavior, yet it lacks a comprehensive modeling solution for networking.

A. Modeling Concerns

The specification focuses on the commonalities that can unify, at the model-level, the heterogeneous low-level concepts based on a few primitive relations. Still, if something more specific is needed, it can be tackled during the interpretation of the specification (cf. Section VI). We assume that these primitive concepts must apply to all things, regardless of their characteristics. We also aim to make these concepts readable to avoid the need for learning new (time-consuming) skills.

Models make it possible to avoid the technical details, which are a source of heterogeneity. A software engineer can create a model specifying a network of things using only the reified primitive concepts. Thus, only the aspects necessary to develop the network's business logic are expected, namely in the form of a model. Code generation is the process that allows moving from the model to code, and it reproduces this model using the concepts of the target low-level programming language or protocol.

Further, we discuss in detail the primitive concepts. Before, it should be noted that within this methodology, we define the following responsibilities:

- **Thing Designer:** one responsible for writing the behavior of the things.
- **Network Designer:** one responsible for writing the behavior of the network.
- **Policy Designer:** one responsible for writing the policies aiming to ensure the correct functioning of the network from a specific angle (e.g., security angle, business angle).

B. Thing Modeling

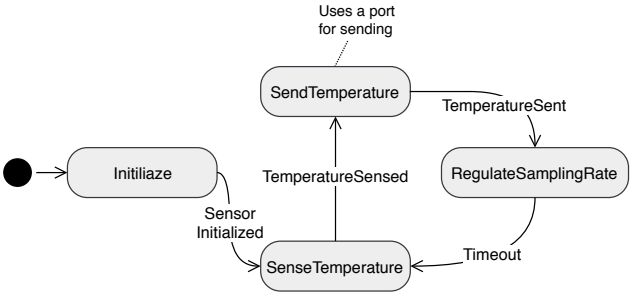

Fig. 1. Statechart-based behavior of a temperature sensor.

We present in this section ThingML, authored by Harrand et al. [46], as it is the main requirement of our methodology. We presume that the model encapsulating the thing's behavior, called ThingML-Model (TH-Model), is specified using ThingML-DSL (TH-DSL)[1]. ThingML-DSL (TH-DSL) permits specifying a statechart-based behavior along with functions and properties that can be called within any state. TH-DSL provides a syntax based on the Xtext[2] grammar. This grammar provides a way to write the model in a textual form.

Figure 1 depicts an example of the behavior of a temperature sensor thing based on a statechart, and Listing 1 shows its equivalent in TH-DSL. Each state accomplishes a specific goal, and each state can have a transition specifying its next state. In this example, the state *SendTemperature* uses the port *sendingTemperaturePort* to send the temperature in Line 44. The port provides a means to route some data within the statechart to a specific internal address identified by the port name. Within our methodology, this statechart needs to be specified by a **Thing Designer**.

TH-DSL offers the concept of *external connector*. This concept lets us wire the port with the outside by specifying the protocol and the serialization format. The ThingML Code

[2]https://www.eclipse.org/Xtext

Generator (TH-CGEN) reproduces the same statechart specified in TH-DSL as well as the *external connector* in a target programming language (e.g., C/C++, Arduino, JavaScript, Go).

```
1  thing temperatureSensor
2  @c_header "#include <DHT.h>" // DHT is a library to read the
       temperature from a sensor
3  @c_header "dht sensor;"
4  {
5    property currentTemperature : UInt8 // Storing the temperature
6    property sensorPin : UInt8 = 8 // the pin where to read the data
7    property samplingRate : UInt8 = 3000 // the sampling rate
8    message temperatureMessage(temperatureValue: UInt8)
9
10   provided port sendingTemperaturePort {
11     sends temperatureMessage // the sending port
12   }
13
14   function sense() do
15     'sensor.read11('&sensorPin&')' // embedding arduino code to
           read the temperature; &sensorPin& sets the value 8 in
           the low–level code
16     currentTemperature = 'sensor.temperature' // assiging the
           temperature value to the currentTemperature property
17   end
18
19   statechart temperatureSensorBehavior init initialize {
20     state initialize {
21       on entry do
22         println "initialize"
23         'sensor.begin();'
24       end
25       transition –> senseTemperature
26     }
27     state senseTemperature {
28       on entry do
29         println "senseTemperature"
30         sense()
31       end
32       transition –> regulateSampling
33     }
34     state regulateSampling {
35       on entry do
36         println "regulateSampling"
37         'delay('&samplingRate&')' // setting the sampling rate
38       end
39       transition –> sendTemperature
40     }
41     state sendTemperature {
42       on entry do
43         println "sendTemperature"
44         sendingTemperaturePort!temperatureMessage(
               currentTemperature) // sending the current
               temperature via the sending port
45       end
46       transition –> senseTemperature
47     }
48   }
49 }
```

Listing 1. The behavior of the temperature sensor in ThingML-DSL; The syntax for embedding code is: '<EMBEDDED CODE>'.

We created a DSL called CyprIoT-DSL (CY-DSL)[3], dedicated exclusively to networking. Listing 2 depicts the declaration of a **thing**. For instance, Line 1 imports the model (i.e., ThingML statechart) of a light sensor, and has been assigned the role sensor (cf. Section IV-C2 for more details on roles). It consists of a name as an identifier (i.e., TemperatureSensor) and the relative path in disk of the ThingML-Model (TH-Model) (i.e., "temperatureSensor.thingml").

It is noteworthy that when it is not possible to express an instruction using TH-DSL syntax, TH-DSL permits embedding low-level code at the model-level. The TH-CGEN places the embedded code, as such, in the statechart of the target programming language. Thus, at worst, expressing low-level concepts from the model-level is still possible (e.g., Lines 16 and 37 of Listing 1).

```
1  import LightSensor "lightSensor.thingml" assigned sensor
2  import TemperatureSensor "temperatureSensor.thingml"
       assigned sensor
3  import Gateway "gateway.thingml" assigned actuator, sensor
```

Listing 2. Declaration of a thing.

In summary, ThingML is useful for us to specify a thing's behavior in the form of a statechart with a communication interface (i.e., port wired via an external connector) and the generation of its equivalent in the low-level code using TH-CGEN.

C. Network Modeling

This section provides the underlying primitive concepts to wire things, group them, and create a network.

1) Channel: The concept of channel constitutes the medium between the sender and receiver interface. The utility of a **channel** concept arises because of the need to wire various things without concerns about the concrete details of their communication means, namely the protocol or the message format. This point is a crucial requirement to foster collaboration between heterogeneous things.

Indeed, the concept of **channel** decouples the communication of things from their programming languages, thus enabling seamless networking at the model-level using only abstract and unified concepts. Listing 3 shows an example of a **channel**, containing two paths, where the second is a fork of the first. The concept of **path** offers a uniquely identified way to exchange messages via the channel, while a **fork** enables us to organize paths in the form of a tree [47]. It is inspired by existing standard protocols. For instance, in MQTT [48] a message is exchanged via a topic, while in HTTP via a Uniform Resource Locator (URL). Both topics and URLs can be unified under the concept of **path**.

```
1  channel mySimpleChannel {
2    path indoor
3    path temperaturePath (temperatureMessage:JSON) fork indoor
4  }
```

Listing 3. Declaring channel, path and fork (Line 1 to 6).

For a more concrete example, in HTTP we consider the URL *https://atlanmod.org/cypriot/* as a path and *https://atlanmod.org/cypriot/smarthome* as one of its forks, likewise in MQTT we consider the topic *org/atlanmod/cypriot/* as a path and *org/atlanmod/cypriot/smarthome* as one of its forks. As shown in Listing 3, a **path** consists of an identifier (temperatureTopic), a declaration of the accepted mes-

[3]CyprIoT Github : https://github.com/atlanmod/CyprIoT/

sage (temperatureMessage) and the message serialization format (**JSON**). Hence, an exchange via a **path** is transparent, thus easing detection of incompatibility between a message and a path or a port. The compatibility refers to the fact that a message needs to be accepted (i.e., understood) by the sending port, the receiving port and the path that connects them, to reach its destination correctly. For instance, the sendingTemperaturePort is compatible with the path temperatureTopic as both understand internally the message temperatureMessage.

2) Users and Roles: We also reified the concepts of user and role. Lines 2-3 of Listing 4 show an example of a **user** declaration. It consists of an identifier (i.e., Bob or Alice) and optionally a token for identification at a low-level (i.e., pa$$word). The declaration of a user serves to specify the owner of a thing. A user can own several things, but only one user can own a thing. The owner has the right to access and share the thing's messages within a policy. For the sake of focus, users' ability to share the same thing is not supported for now. The concept of user gives more context w.r.t the place of the thing in the network and provides the opportunity to group the things by user.

```
1 // User declaration
2 user Bob:pa$$word
3 user Alice:pa$$word
4 // Role declaration
5 role sensor
6 role actuator
```

Listing 4. Declaration of users and roles.

The concept of role in the IoT is useful because it can help to attach a specific responsibility to a thing or a group of things. This responsibility enables a more specific control in the network, especially large ones (cf. Section V for more details). Declaring a **role** has a similar syntax than a **user**. Lines 5-6 of Listing 4 show an example of a **role** declaration. It consists of an identifier (i.e., sensor or actuator) and can be assigned to several things, as shown in Listing 2. Also, a thing can have simultaneously several roles.

3) Network: The **Network Designer** declares things, channels, roles, and users, then use them in the **network**. The things and channels need to be instantiated to be used inside the **network**. Several instances may be derived from the same declaration. The concept of a network describes the global network configuration. It defines what things to instantiate and what channels are available. It also contains the wiring of the things' ports to the channels. As shown in Listing 5, a **network** has an identifier (mySimpleNetwork) and a **domain** (org.atlanmod.mynetwork). A **domain** has to be unique and serves as a global identifier for the network at a low-level. For instance, we can use the domain in the path structure as the root path of the networks' channels.

Inside the **network**, we can declare an **instance** of a thing (based on the imported TH-Model). An **instance** of a thing consists of an identifier (e.g., myTempSensor, myGW), the **platform** specifying the target programming language (e.g. C/**POSIX** based), and if necessary the **owner** (e.g., Bob). We can also declare an **instance** of a **channel**. An instance of a **channel** sets a **protocol** (e.g., **MQTT**). Finally, we can specify to **bind** a port of an **instance** of a thing to one or multiple paths of any of the available instances of channels (e.g., Line 10 of Listing 5).

A **network** can also enforce a policy. Multiple policies can be enforced. For instance, in Listing 5, both roleBasedPolicy as well as smartpolicy are enforced. Policies and control strategies are discussed in detail in the next sections.

The network serves as a glue to make distributed statecharts communicate and exchange messages from a conceptual perspective. It constitutes the entry point for the code generator (cf. Section VI-A1).

```
1  network mySimpleNetwork {
2    domain org.atlanmod.mynetwork
3    enforce roleBasedPolicy, smartpolicy
4    // Instances of things
5    instance myTempSensor : TemperatureSensor platform POSIX
         owner Bob
6    instance myGW : Gateway platform JAVA owner Bob
7    // Instance of a channel
8    instance zigbeeChannel:ptpChannel protocol ZIGBEE
9    // Binding : Sending (i.e., =>) the sensed temperature by
         myTempSensor
10   bind myTempSensor.TempDataPortSend => zigbeeChannel{
         temperaturePath}
11   // Binding : Receiving (<=) the sensed temperature
12   bind myGW.TempDataPortRec <= zigbeeChannel{
         temperaturePath}
13 }
```

Listing 5. Specification of a network; sending (=>) and receiving (<=) messages via a path.

4) Forwarding: Usually, in the IoT, a thing may need to pass through a more powerful intermediary thing before reaching its final destination because of its limited resources. In the IoT literature, this mechanism is cited as *multihop routing* or as *intermediary gateway*. Implementing this simple mechanism using low-level concepts is arduous because of heterogeneity.

```
1    network mySimpleNetwork {
2      domain org.atlanmod.mynetwork
3      enforce roleBasedPolicy, smartpolicy
4      // Instance of things
5      instance myRD : RemoteDisplay platform JAVASCRIPT
           owner Bob
6      ...
7      // Instance of a channel
8      instance zigbeeChannel:brokerChannel protocol ZIGBEE
9      instance mqttBroker:brokerChannel protocol MQTT(server="
           mqtt.atlanmod.org:1883")
10     ...
11     // Binding : We add an identifier (i.e., tempBindGw) to the bind
12     bind tempBindGw : myGW.TempDataPortRec <=
           zigbeeChannel{temperaturePath}
13     // Forwarding
14     forward tempBindGw to mqttBroker{temperatureTopic}
15     // Binding : Receiving the forwarded temperature message
16     bind myRD.receivingTemperaturePort <= mqttBroker{
           temperatureTopic}
17   }
```

Listing 6. Forwarding of an existing binding

The concept of forwarding enables to forward an *existing binding* (i.e., **bind**) to another **path**. For instance, in Line 14 of Listing 12, we **forward** the temperature received via **ZIGBEE** by myGW to temperatureTopic, a path of mqttBroker that is

using **MQTT** as a protocol. The **Network Designer** has the responsibility to ensure that myGW supports both protocols. Then, we bind the port receivingTemperaturePort of myRD to receive the message sent to tempMQTTPath. myGW plays the role of an intermediary thing between myTempSensor and myRD . In this particular usecase, we link two things using different protocols without dealing with the low-level heterogeneity as we are designing our network using model-based and unified concepts.

In fact, due to variation in sizes of things, an interoperability issue is a common trait of connectivity of things in the IoT [7]. The ability to forward an existing binding at the model-level allows navigating freely between various ranges of things. The implementation of these forwardings is kept for the code generation phase. During code generation, some specific procedures interpret these model-based forwardings and reproduce their equivalent in the things' target programming languages (cf. Section VI-A2).

V. Control Specification

The previous section shows how to create a model of the network based on unified concepts. This section shows how we can use this model to control the network's behavior based on a declarative policy. By control, we refer to the ability to inject monitors either for restricting communication or triggering actions according to some conditions. We will solely discuss the controls' specification in this section; its enforcement is developed in the next section.

A. Policy

Generally speaking, a policy can serve various purposes [49] (e.g., communication control, administrative goal). However, this study focuses on two main aspects: communication control and smart scenarios. A policy aims to ensure that the IoT application is behaving as expected from a stakeholder's perspective, such as a security officer, government, or the owner of the network.

It contains a set of rules. We define a rule as the composition of a subject (e.g., thing, instance, port, or role), an action type (e.g., permission, trigger), an action (e.g., send, receive, goToState, executeFunction), an object (thing, instance, port, message or path) and time (e.g., specific date, period). As shown in Line 3 of Listing 5, one or more policies can be enforced in the network; in this particular example, the network enforces roleBasedPolicy and smartpolicy.

```
1  policy smartPolicy {
2    rule ...
3    rule ...
4  }
```

Listing 7. Declaration of a policy.

Listing 7 shows a specification of a **policy** written in CY-DSL. It consists of an identifier (i.e., smartpolicy) and two rules, that we discuss further. This policy is enforced in mySimpleNetwork (Line 3 of Listing 5).

The policy's specification is readable and relieved from the low-level technical details. It needs to be written by a **Policy Designer** and made available to the **Network Designer** who can decide to enforce it. The concrete enforcement inside the low-level code is the concern of the code generator (cf. Section VI), assumed by experts. Experts are responsible for developing the enforcement strategies in the code generator and mapping the model's abstract concepts to their concrete equivalents at a low-level.

TABLE I
The combination of rule entities.

Subject	Action Type	Action	Object	Control Type
Port Instance of thing Thing User Role	Permission	Send Receive Send-Receive	Port Instance of thing Thing User Role Path Channel	Communication
State	Trigger	goToState executeFunction	State Function	Thing Behavior

B. Rule

A rule specifies the conditions necessary for an action to be applied to an object. Listing 8 depicts its structure. It comprises 5 parts: *Subject*, *ActionType*, *Action* and *Object* and *Time*. The subject is the entity applying an action, and the object is the entity undergoing the action. The time permits to delimit the effect of the rule over time. Specifying a rule in this structure is meant to be readable as a sentence in plain English. The ability to use model-based abstractions permits the dissociation of the network's control from its concrete implementation (source of heterogeneity).

```
rule <Subject> <ActionType>:<Action> <Object>
                                  when time:<start>-<end>
```

Listing 8. Rule syntax.

Table I depicts the possible entities of a rule for each of its parts and how they can be combined. The first column contains the subject's supported entities, and the fourth column, the object's entities.

C. Types of Control

The rule structure previously mentioned offers two types of control: communication control and behavioral control. The separation of the specification from the implementation permits focusing on specifying the constraints *we wish* to see in the implementation. This structure can serve various control purposes down the road and may be subject to extension. So far, we chose to implement these two types of control first as a proof of concept:

- **Communication Control [50]:** consisting of *denying* or *allowing* the sending or receiving of messages. For instance, a rule can deny or allow the port p_y from sending its messages to the port p_x. We may apply the same control for two things, two users or the combination of these entities.

- **Behavioral Control [7]:** consisting of triggering an action (i.e., goToState, executeFunction) on the object based on the current state of the subject. Indeed, as the behavior consists of a statechart, the control aims to change this statechart to satisfy the rule's intent. For instance, a rule can specify that a thing t_x should go to the state s_i when the current state of a thing t_y is s_j (Task 2).

In summary, this section introduced the central notions of the rule-based system we use for control. In the next sections, we show how they may be used in more detail.

D. Communication Control

The communication flow is a critical asset of a network of things, hence the need to regulate it. However, its regulation at a low-level is burdensome because of heterogeneity. The proposed solution consists of specifying this regulation at the model-level and enforcing it by the code generator using a dedicated procedure.

Communication control rules aim to specify the constraints on this flow—many entities of the network where this flow transit are *controllable*. For instance, in a thing, we can control what to receive and what to send at the port level, in a channel, we can control what message to accept at the path level, and at the user level, we can control the messages that can be sent or received by the thing s/he owns.

We presume that all communications are denied by default unless a rule allows things to communicate. The communication control rules allow controlling communication between things, users, roles, and the combination of all of them.

We can control communication using ports, things, users, and roles. The smallest level of granularity among these entities is the port. The port is a checkpoint, i.e., where an action (e.g., deny, allow) can take place concretely. Then comes, in that order, an instance of a thing, type of thing, and user/role. When the subject/object is a type of thing, the action applies to all its instances. When it is a user, it applies to all things and instances s/he owns, and when it is a role, it applies to all things and instances where this role is assigned. This scheme enables an action to be enforced at various levels of granularity.

Moreover, the object can be of type **channel** or **path**. The path is also a checkpoint, i.e., it is where the action (e.g., deny, allow) can take place concretely. When the object is a channel, the action applies to all its paths.

1) Potential application: The communication control type has various applications in the real world, such as access control [51], content-based control [52], or privacy control, to cite a few.

Our communication control approach is singular and aims to be generic, yet, we could see some commonalities with Attribute-Based Access Control (ABAC) and Role-Based Access Control (RBAC) from an access control perspective. Indeed, while we do not cover all the theories behind these two models exhaustively, we offer similar mechanisms. For instance, if we consider that the elements of the network are the attributes, we cover, to some extent, the concepts of ABAC. Whereas for RBAC, we provide a dedicated concept of **role** that can be used in our communication control rules.

It is worth noting that this study focuses on the software engineering aspects of the IoT. However, we believe that extending our methodology to cover comprehensively some access control models such ABAC and RBAC is a promising avenue of research. All the more that Access Control (AC) is an important milestone for the IoT [53], [54]. Separating the specification from the implementation should ease the enforcement of such access control models.

E. Smart Rules

The interoperability of things to achieve Smart Scenarios suffers from heterogeneous concepts at a low-level. **The proposed solution** consists of specifying these scenarios at the model-level and implementing them by a code generator using a dedicated procedure.

The previous section shows how we can specify communication flow control; this section presents how we can control the network's behavior according to contextual factors, such as the state of things and time.

This type of control enables the specification of Smart Scenarios, i.e., the ability to trigger certain actions according to the context of the network [55]. The interoperability problem of the IoT is one of its major obstacles [7]. Indeed, the difficulty to implement smart scenarios is often caused by the heterogeneity of things at a low-level. Our methodology avoids this heterogeneity by relying on unified concepts at the model-level.

1) Potential applications: By lacking a common representation of the behavior at a low-level, it is not easy to implement a smart scenario where things collaborate towards a common goal. This kind of scenario implies the ability of a thing to impact another thing's behavior, regardless of its resources. The hypothesis of relying on a statechart-based thing, i.e., a specific and unified way to specify the behavior, enables its control according to its state.

The specification of such Smart Scenarios must be seamless, i.e., not impacted by the low-level technical details, such as the communication protocols or the programming languages that are often heterogeneous. This heterogeneity may distract us from achieving the interoperability of these heterogeneous things which is enabling these smart scenarios.

The network may be influenced by two types of factors: behavioral factors, i.e., the properties and the states of things, and temporal factors, i.e., the physical time. For instance, in a typical smart scenario, we may need to specify that when a thing t_x is on the state s_i, the thing t_y has to be on the state s_j. Thus, t_y adjusts its behavior according to the state of t_x. Specifying this simple example requires many skills and resources with a traditional software engineering approach, as many heterogeneous concepts are involved at a low-level.

2) Behavioral factors: The behavioral factors correspond to the control according to the states of things. A smart rule can activate two actions: a) *goToState*, instructing the thing to go to a specific state, and b) *executeFunction*, executing a function in the thing. These actions are triggered based on the current states of the subject or object thing. For instance, the rule in Line 2 of Listing 9 specifies that the state isLow of

myTempSensor (i.e., **instance** of a temperature sensor), triggers myAC (i.e., **instance** of an air conditioner) to be at the state isOn (typically for an optimal cooling).

Also, as a thing can provide a function (i.e., a sequence of instructions), there are cases where a function needs to be executed depending on the state of another thing. For instance, the rule in Line 11 specifies that the state isHigh of myTempSensor, triggers the function setTemperature(25) of mySAC (thus, setting the temperature to 25°C when it's warm).

```
1 policy smartPolicy {
2   rule myTempSensor->state:isLow trigger:goToState myAC->
        isOn
3   rule myTempSensor->state:isLow trigger:executeFunction
        myAC->setTemperature(25)
4 }
```

Listing 9. Go to a state (Line 2) and execute a function (Line 11) according to another thing.

3) Temporal factors: The temporal factors correspond to a specific date and time or a period. When a temporal factor is set, the control consists of applying the action only when the temporal condition is met—for instance, controlling a communication or triggering an action only at a specific time of the day.

The syntax consists of adding the keyword **when** and specifying either a specific time or a period using the keyword **time**:. Listing 10 shows an example. The rule states that mySAC must go the state isOn if the myTempSensor reaches the state isLow, but only when the time is between 20/01/2020 at 11:00:00 and 20/01/2020 at 13:00:00.

The specification of these temporal factors is straightforward. Still, their implementation may be difficult in a decoupled system as there is no standard way to define the physical time (cf. Section VI). On another note, logical time is unsuitable here, as it is not a problem of ordering. Instead, things need to depend on the physical world, incidentally one fundamental promise of the IoT.

```
1 policy smartPolicy {
2   rule myTempSensor->state:isLow trigger:goToState mySAC
        ->isOn when time:20012020@11:00:00-20012020@13
        :00:00
3 }
```

Listing 10. Go to a state (Line 2) and execute a function (Line 11) according to another thing.

The temporal factors are useful in controlling a network according to the physical environment. Separating the specification from the implementation helps better tackle the difficulty of implementing time. We show in the next section some implementation strategies.

F. Conflict Detection and Resolution

The detection of conflicts at deployment is costly and difficult to debug. The proposed solution consists of detecting and resolving them early in the editor.

The detection of conflicts between rules can help for a safe deployment. We consider that two rules conflict when they cannot be enforced simultaneously. These conflicts can be prevented at various steps of the software engineering process. However, as a general norm, detecting them as early as possible is recommended.

We aim to enable the detection of most conflicts between any rule in the long run. Still, so far, we only offer some conflict detection and resolution mechanisms w.r.t the communication control rules. Given their variability, there could be some implicit conflicts that are difficult to notice. For instance, a rule involving only users may conflict with a rule involving a more fine-grained entity such as a port or a thing. We provide mechanisms to detect them directly in the editor and ask the software engineer to resolve them.

1) Early Detection: The conflicts are displayed in real-time in Eclipse Integrated Development Environment (IDE) (cf. example in Figure 2). The model is deemed invalid until the conflict is resolved. This mechanism prevents the engineer from generating inconsistent code that may contain bugs. Figure 2 shows an example of the error that may be displayed. The error reports an inconsistency of the model due to two conflicting rules; namely, the first rule allows device2 to send to pubsub1, while the second states the opposite.

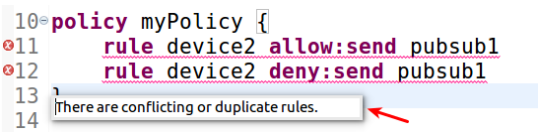

Fig. 2. A conflict detection error in the editor.

2) Conflict Detection Algorithms: We present a few algorithms that we use in our implementation to detect conflicts between rules. The goal here is to show that conflict detection can be automated. Some aspects, such as their efficiency or scalability, did not receive detailed treatment. These algorithms are executed in the editor in real-time. An error is shown when a conflict is detected, asking the engineer to resolve it. We implemented these algorithms as a proof of concept; more of such algorithms will be developed for more fine-grained conflict detection in the future.

So far, we have implemented two algorithms: (1) Detection of a conflict between a group (i.e., a role or a user) and a thing, and (2) Detection of a conflict between a role and a user.

The algorithm (1) aims at detecting conflicts between a group (user or role) and a thing. A group consists of several things. The algorithm returns the conflicting rules, w.r.t an input rule. It iterates over all rules and checks whether the subject and the object (or type role or user) contains the object of the input rule on the one hand and whether the actions are opposed on the other. If so, it adds the rule to the *Conflicting* collection and returns it.

The algorithm (2) aims at detecting conflicts between two groups, i.e., when the subject and the object are either a user or a role. The algorithm returns the conflicting rules, w.r.t an input rule. It checks whether any thing contained in the subject/object set of the input rule is also contained in the subject/object set of any other rule. Then, it checks whether

Algorithm 1: Detecting a conflict between a thing and a group.

Input: $InputRule$: Rule to test, $AllRules$: Collection of all rules
Output: $Conflicting$: Array of conflicting rules
$Conflicting \leftarrow \emptyset$;
if $size(AllRules) > 0$ **then**
  foreach $r \in AllRules$ **do**
    if $subject(r) = subject(InputRule)$ & $object(r) \cap object(InputRule) \neq \emptyset$ & $action(r) \neq action(InputRule)$ **then**
      $Conflicting \leftarrow Conflicting \cup \{r\}$;
  end
return $Conflicting$;

the actions are opposed. If these two conditions are met, it adds the rule to the *Conflicting* collection and returns it.

Algorithm 2: Detecting conflict between two groups.

Input: $InputRule$: Rule to test, $AllRules$: Collection of all rules
Output: $Conflicting$: Array of conflicting rules
$Conflicting \leftarrow \emptyset$;
if $size(AllRules) > 0$ **then**
  foreach $r \in AllRules$ **do**
    if $subject(r) \cap subject(InputRule) \neq \emptyset$ & $object(r) \cap object(InputRule) \neq \emptyset$ & $action(r) \neq action(InputRule)$ **then**
      $Conflicting \leftarrow Conflicting \cup \{r\}$;
  end
return $Conflicting$;

3) Resolution Strategies at Enforcement: The detection of conflicts in the editor asks the software engineer for an intervention to resolve the conflict. This mechanism is far from complete, and in some cases, it may not be able to detect a conflict. Hence, the resolution of conflicts at enforcement consists of deciding what action takes precedence in conflict during code generation. In that respect, the software engineer needs to specify the resolution strategy for each enforced policy. We propose a few resolution strategies: Best-Effort, Deny-First, Allow-First. We discuss some of these strategies in the next section (cf. Section VI-C2). Line 3 of Listing 11 shows an example of the specification of a resolution strategy; namely Deny–First for roleBasedPolicy and **Best–Effort** for smartpolicy.

```
1 network mySimpleNetwork {
2   domain org.atlanmod.mynetwork
3   enforce roleBasedPolicy Deny–First, smartpolicy Best–Effort
4   ...
5 }
```

Listing 11. Forwarding of an existing communication.

VI. Code Generation

Code generation aims to eliminate redundant tasks using Model Transformation (MT). MT [56] is a process based on transformation rules (usually written by experts) that takes one or more input models to produce a target artifact. There are two types of transformations: Model-to-Model Transformation (M2MT) and Model-to-Text Transformation (M2TT). The former produces a *target model* while the latter produces a *target text*, from the input model. We use both types for the interpretation of the network and control specifications.

The Transformation Process (T-PROCESS) (cf. Figure 3), i.e., the transformation work accomplished inside the CyprIoT Code Generator (CY-CGEN), has the function to transform the input models, i.e., the CyprIoT-Model (CY-Model) and TH-Models, into the target artifacts by changing the behavior of the TH-Models according to the CY-Model, and if necessary generating any related textual artifact. In this process, the experts are expected to map the abstract concepts into low-level concepts for the artifacts' automatic generation.

As shown in Figure 3, we use the AtlanMod Transformation Language (ATL)[4] for Model-to-Model Transformation (M2MT) and Acceleo[5] for Model-to-Text Transformation (M2TT), two state-of-the-art transformation tools. ATL uses rules to apply the transformation, where it matches an element in the input model to produce another element in the output model that is satisfying the rule. Acceleo, on the other hand, relies on templates, where it fills a template's placeholders with information from the input model. The final output of the T-PROCESS consists of either the implementation code (e.g., C, JAVA, Arduino), the simulation code (to run on the local machine for testing the network before deployment) or textual artifacts (e.g., access control rules, documentation). Generating the simulation code provides a cost-effective means to test the network's scalability and performance before generating the implementation code.

A. Model-to-Model Transformation

M2MT allows us to decorate the TH-Model (i.e., the behavior of the thing) according to the CY-Model (i.e., the specification of the network) at the model-level. Indeed, it takes information from the CY-Model and adds *only what is needed* to the TH-Model to conform to the specification of the network. As this process takes place at the model-level (using unified concepts), interoperability is preserved. The transformed TH-Models are then used to generate their equivalent in the low-level code using TH-CGEN.

The ATL rules[6] of this process are depicted using a Graph Transformation Rules (GTR) [57] for illustrative purposes. In the GTR, the left-hand side (LHS) shows the TH-Model before transformation, while the right-hand side (RHS) shows the TH-Model after transformation. All the elements added in the RHS (in white) are added according to the specification of the network in the CY-Model.

We present in this section how we use two M2MT to adapt TH-Models according to the CY-Model: (1) *adding the communication interface according to the network* (i.e., adding the protocol and path) and (2) *forwarding an existing binding*. (1) and (2) tackle the networking issues mentioned respectively in Tasks 1 and 4. In Section VI-C, we show how we tackle the enforcement of the policies.

1) Wiring: The upper part of Figure 4 depicts the GTR of (1). On the RHS, the elements in white (i.e., *MyExternalConnector*, *MyProtocol* and some annotations) are added according

[4]https://www.eclipse.org/atl/
[5]https://www.eclipse.org/acceleo/
[6]CyprIoT Github > generator/transformations/

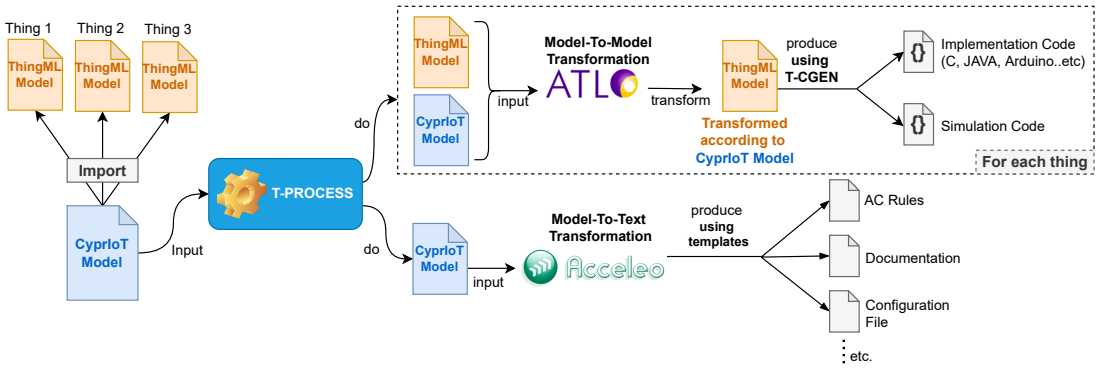

Fig. 3. Generation of network artifacts using the T-PROCESS.

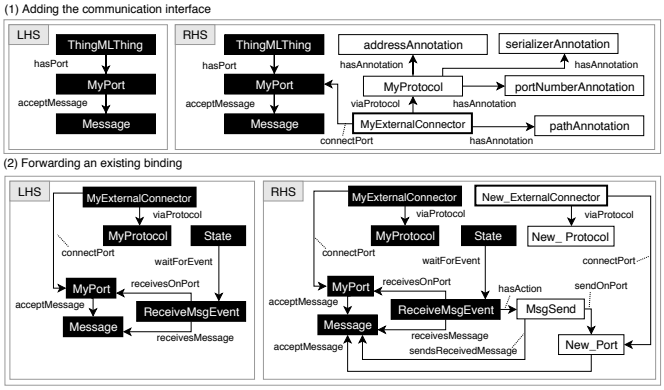

Fig. 4. Upper Part: Adding the communication interface according to the specification of the network GTR (1); Lower Part: Forwarding an existing binding GTR (2); White boxes are added based on the CY-Model; The added External Connector box is thicker for readability purpose.

to the specification of the network in the CY-Model. An external connector links a port to a protocol and uses annotations for the configuration of the protocol (e.g., for Message Queuing Telemetry Transport (MQTT), the annotations specify the broker address, the port, the topic, and the serialization format). Indeed, a **bind** consists of adding *MyExternalConnector* to the TH-Model along with the connection information specified in the CY-Model. TH-CGEN reproduces the equivalent of this external connector in the low-level code. For instance, for myRD in Listing 12, it adds the external connector to connect the port receivingTemperaturePort via the protocol MQTT (with the configuration: addressAnnotation="mqtt.atlanmod.org", serializerAnnotation="JSON" and the portNumberAnnotation="1883") through the path (i.e., topic) pathAnnotation="temperatureTopic ". This gives TH-CGEN all the information needed to generate the low-level code containing the correct communication interface for myRD.

If the target programming language is C/Posix and the communication protocol is MQTT, then TH-CGEN generates a statechart communicating via MQTT in C/Posix language; the same would apply in the case of Java or Arduino. Thus, as the low-level code is a mere translation of the TH-Model, interoperability is preserved at the low-level code. The T-PROCESS applies this M2MT to all things in the network. So, in summary, the function of this transformation is to connect things in the form of a network.

2) Forwarding: The lower part of Figure 4 depicts the GTR of (2). To forward the binding corresponding to *MyExternalConnector* (via *MyProtocol* using *MyPort*), we create a *New_ExternalConnector* with a *New_Protocol* (i.e., creating a new external connector using the protocol needed for forwarding). For the sake of readability, we omitted adding the annotations (that are similar to (1)) of the *New_ExternalConnector* and *New_Protocol*. The transformation looks for any state waiting to receive the message to forward (i.e., *ReceiveMsgEvent* waiting for *Message*) and adds the action *MsgSend* to the event. *MsgSend* consists of sending the received message as such using the *New_ExternalConnector* (via *New_Protocol* using *New_Port* that accepts the same received message). This transformation is useful to enable a seamless cross-range interoperability using an intermediary thing as a bridge between ranges. It enables to forward a received message as such with, e.g., a protocol p_i via a new protocol, e.g., p_j, presuming that the thing is physically equipped to support both protocols.

It is important to note that the transformed TH-Models *"interoperate"* at the model-level as only abstract and unified concepts (e.g., port, path, bind) are used. TH-CGEN (cf. Figure 3) reproduces the same concepts at the low-level code for each TH-Model (i.e., same statechart, same communication interface) according to the specified target programming language for each thing.

B. Model-to-Text Transformation

M2TT allows us to generate any related textual artifact that is not part of a thing's internal behavior (e.g., access control rules, configuration file, documentation). The information necessary to make these artifacts is usually contained in the network's specification, yet it needs to be written in the right format.

The lack of unification of low-level concepts adds another layer of complexity to the interoperability between things. Indeed, the heterogeneity of low-level concepts (e.g., user, topic, URL, permission, documentation, configuration) inhibits connecting things safely and leads to poor synchronization between all the network elements. This subsection shows how we use the unified network specification to automatically generate some network artifacts using M2TT.

The typical infrastructure of a network of things includes artifacts that are not part of a thing's internal behavior but remain essential for the network's correct functioning. For instance, in the example of Listing 12, we need to ensure a secure access control into the MQTT broker between the myGW and myRD. The information about access control is contained in the CY-Model. We need to write it in the right syntax, i.e., the syntax of the target broker at a low-level (e.g., Mosquitto[7], RabbitMQ[8]). The goal of M2TT is to generate this kind of artifacts based on templates. It takes advantage of the network's unified specification to synchronize, using an automatic process based on M2TT, the low-level textual artifacts. By synchronization, we refer to the ability to reproduce the same information uniformly through all the artifacts of the network (e.g., the equivalent information that appears in the access control rules should appear in the documentation file).

The syntax and semantics of these textual artifacts are specified using Acceleo templates. Figure 5 depicts an illustration of how Acceleo textual generation works. To show that this process can be applied to generate various textual artifacts based on the same source model, we demonstrate how to generate the same access control rules for two MQTT brokers, namely Mosquitto and RabbitMQ. As shown in the figure, the M2TT fills their respective templates automatically using the information contained in the CY-Model.

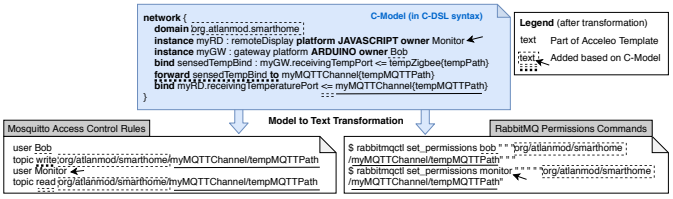

Fig. 5. Generation of Access Control Rules for Mosquitto and RabbitMQ using M2TT; Same information indicated with same sign; For RabbitMQ syntax, first argument (" ") is for *configure* permission, second for *write* permission and third for *read* permission.

Moreover, the artifacts may be diverse; the CY-CGEN offers an interface to the T-PROCESS via the plugin system. Thus, a developer can make a custom plugin to generate any textual artifact in a traceable manner by leveraging any step of the code generation process. In a large network, writing many of these textual artifacts uniformly is time-consuming when using traditional software engineering approaches (e.g., requires learning a new syntax, synchronizing the artifacts manually) and exposes the IoT engineer to introduce more bugs. These transformations help tackle the issues of Task 4.

C. Enforcement Strategies

The enforcement refers to the implementation of the policies in the generated artifacts. It enables the concrete implementation of smart scenarios (Task 2) and communication control (Task 3). It also relies on MT. In Section VI-C2, we present our strategy to implement the enforcement of communication control rules, and in Section VI-C3, we present our approach

[7]https://mosquitto.org
[8]https://www.rabbitmq.com

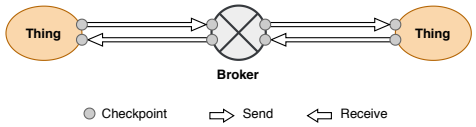

Fig. 6. Enforcement checkpoints.

to implement the smart rules. We also point out the various other strategies that could be implemented.

The enforcement of communication control rules consists of interpreting the rules presented in Section V-D to include them inside the deployable network artifacts. This section first discusses the possible enforcement checkpoints in a network, then the enforcement mechanisms.

1) Enforcement Checkpoints: As depicted in Fig. 6, controls can be enforced at various checkpoints of the network architecture: 1) in the broker (if any), by controlling the access to it, or 2) in the thing by changing its internal behavior in the TH-Model, on send or receive. The choice of the correct enforcement checkpoint depends on the strategy. Some checkpoints may be more or less preferable for various reasons, such as security, trust, or implementation challenges in some scenarios.

From a security perspective, controlling the communication on receive requires checking whether the message satisfies the control conditions before the reception. The message can still be intercepted while transiting and demands additional processing on receive. This processing may waste (scarce) resources if the received message does not satisfy the conditions. Whereas, when communication is controlled on send, the message remains until it meets the control conditions; this is more secure and privacy-friendly as the thing keeps control over the message. Moreover, sometimes distributed control can be more scalable and flexible and avoids the "single point of failure" risk associated with the control on the broker as customary.

2) Enforcement Mechanisms: Our methodology relies on separating the specification from the implementation; the policy's enforcement depends on the specified network's parameters. Hence, many enforcement strategies at the implementation may be automated according to these parameters.

The enforcement mechanisms of a communication control rule consist of programmatically allowing or denying sending or receiving on the checkpoints, at the model level, using M2MT, i.e., transforming the behavior inside the TH-Model to satisfy the rule. We show a few examples of such mechanisms using illustrative figures. All these mechanisms are applied at the model-level using M2MT and are permitted by the TH-Model formalism.

Figure 7 shows an example of a simple rule consisting of three things. The rule denies *thing2* from receiving any message from *thing1*. In this case, the enforcement, as stated by the rule, occurs at *thing2* on the *receive checkpoint*. The same mechanism is applied in Figure 8, but on the *send checkpoint*. We consider this enforcement correct as it translates the exact meaning of the rule in the implementation. Sometimes the correct enforcement may not be possible at low-level for

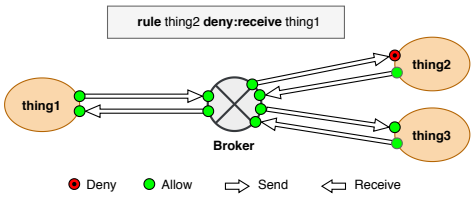

Fig. 7. Enforcement of ”on the receive” checkpoint.

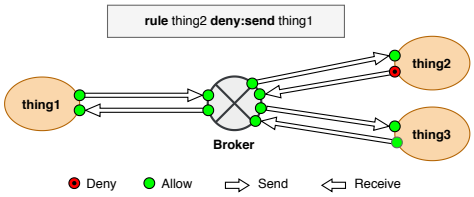

Fig. 8. Enforcement of ”on the send” checkpoint.

technical reasons.

Figure 9 shows the enforcement of a rule involving users. *Alice* is denied to receive any message from *Bob*. The enforcement consists of preventing any message, sent by any thing owned by *Bob*, to be received by any thing owned by *Alice*.

Figure 10 shows the enforcement of a rule based on roles. The first rule denies any thing with the role *sensor* from receiving a message from the broker, as a sensor is only expected to send data. The second rule denies any thing with the role *actuator* from sending a message to the broker, as an actuator is only expected to receive instructions. The enforcement targets all the things having these roles and blocks the reception of messages on the *receive checkpoint* for sensors and on the *send checkpoint* for actuators.

In some cases, the correct enforcement of the rule is a deadlock, i.e., the CY-CGEN can't decide the correct enforcement. For instance, the rule in Figure 11 denies *thing1* from sending a message to *thing2*. *thing2* and *thing3* consumes messages from the same path. If we deny *thing1* from sending to the path of *thing2*, we will be preventing *thing3* from receiving the message. Thus, the correct enforcement, as stated by the

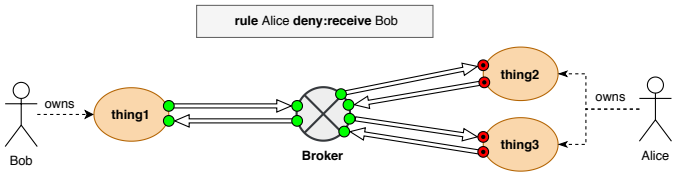

Fig. 9. Enforcement of a user-based rule.

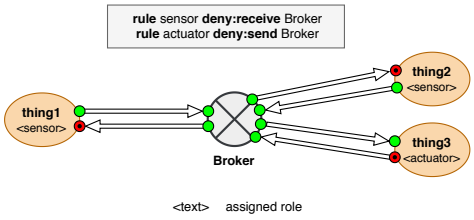

Fig. 10. Enforcement of a role-based rule.

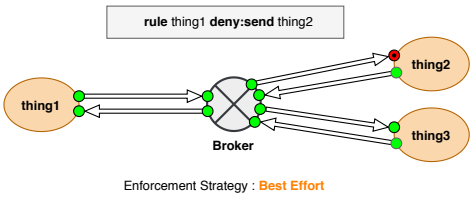

Fig. 11. Enforcement of a Best-Effort strategy.

rule, is not possible in this configuration. In this case, the tie break is the enforcement strategy specified in the network, as shown in Line 3 of Listing 12.

The other enforcement strategies apply the same principles, i.e., interpret the model and implement the policies according to their drive. The strategy Deny–First presumes that all communications are denied unless there is a rule allowing a communication. While the strategy Allow–First assumes the opposite. It is important to note that, unlike traditional software engineering, these enforcement mechanisms have to be implemented only once by an expert and are later applied automatically by CY-CGEN.

```
1 network mySimpleNetwork {
2   domain org.atlanmod.mynetwork
3   enforce myPolicy Best–Effort
4   ...
5 }
```

Listing 12. Specifying the enforcement strategy.

3) Enforcement of Smart Rules: As the network specification relies on unified concepts at the model-level, we avoid the interoperability issues to achieve smart scenarios. The enforcement of smart rules consists of adding *only what is needed* in each thing so that the output model conforms to the rule. As shown in Figure 3, the enforcement relies on a M2MT based on ATL. The transformation inputs are the CY-Model and the TH-Model of the thing to be transformed. The output is a transformed TH-Model incorporating the needed part from the smart rule.

There may be various ways to enforce smart rules, we present here our implementation. This implementation serves as a proof of concept, more advanced implementations covering other concerns may be implemented in the future. Figure 12 depicts the GTR of *trigger:executeFunction* rule (cf. Line 11 of Listing 7) and Figure 13 the GTR of *trigger:goToState* rule (cf. Line 2 of Listing 7).

The enforcement of a *trigger:executeFunction* rule consists of two M2MTs; one for the subject thing and the other for the object thing. On the one hand, in our implementation, this transformation needs to add in the subject thing a way to inform the object thing that it entered the subject state and, on the other hand, to add a way for the object thing to receive this information. For this, we create a message and send it on the entry of the subject state (i.e., we use *MessageSendAction OnEntry* to send *CommandMsg*). An event waits for the message inside all states of the object thing (i.e., we use *ReceiveMsgEvent*). This event is added to every state to ensure that the function can be executed anytime regardless

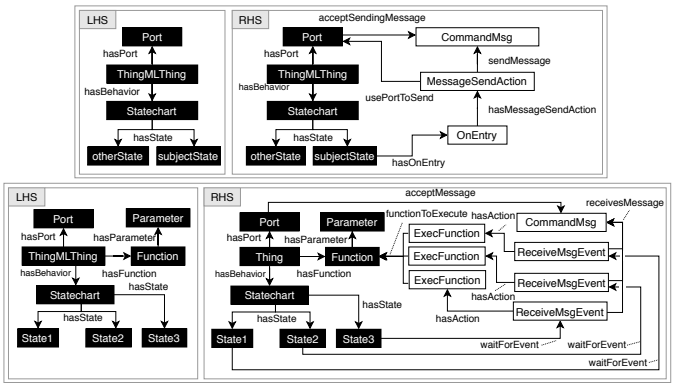

Fig. 12. Upper Part for Subject, Lower Part for Object; Applying the trigger:executeFunction rule GTRs (3) with the subject thing having two states and the object thing having three states; White boxes are added based on the CY-Model.

of a thing's current state. Once the object thing receives the message, it executes the function (i.e., we use *ExecFunction*) with the specified *Parameter* in the rule. We try to send the message using an existing path between the two things. If no direct path exists between them, we search for an indirect path.

The *trigger:goToState* rule in Figure 13, on the other hand, uses the same principle, but adds, inside all states of the object thing, a transition to the state to go to (instead of an *ExecFunction* in comparison with a *trigger:executeFunction* rule). These transformations enable the enforcement of a smart scenario at the model-level. TH-CGEN reproduces the equivalent statechart for each thing in the low-level code.

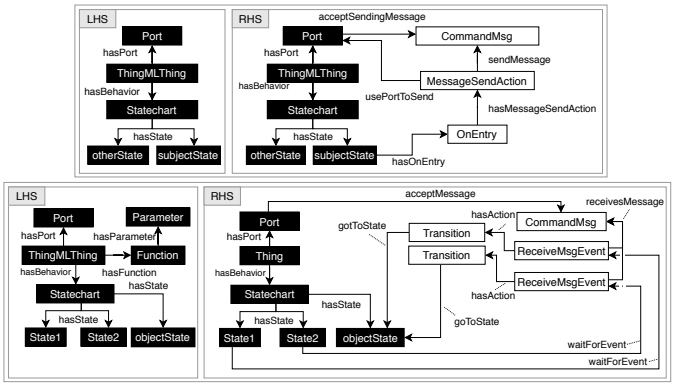

Fig. 13. Upper Part for Subject, Lower Part for Object; Applying the trigger:goToState rule GTRs (4) with the subject thing having two states and the object thing having three states; White boxes are added based on the CY-Model.

The last step of our methodology consists of generating network artifacts from the network and policies specifications. This section introduced an extensible code generator, named CY-CGEN, capable of interpreting the model and generating the specified network's artifacts.

VII. Evaluation

A. Comparison of Lines of Code

We tested our methodology on networks ranging from 1 to 25 things. We use traditional software engineering as a baseline, with things software written in C (suitable for things of any size), Java (ideal for medium and large things), and Arduino (suitable for small things). We compare the Lines of Code (LoC) needed between the baseline and our model-based methodology. The compared code for C, Java and Arduino is the output of CY-CGEN; code written by real software engineers should be within these ranges. As our focus is on networking, we presume that the behavior of things is not part of this evaluation as it has been evaluated in previous work on ThingML [46]. We compare only the LoC needed for networking and policy enforcement. We removed the comments and empty lines from the count.

TABLE II
Comparison of the required LoC with CY-DSL, C, JAVA and Arduino for each M2MT.

	Number of Lines of Code			
Transformation	**CY-DSL**	**C (% gain)**	**Java (% gain)**	**Arduino (% gain)**
Wiring	11	186 (94.09%)	137 (91.97%)	102 (89.22%)
Forwarding	5	237 (97.89%)	14 (64.29%)	24 (79.17%)
trigger:goToState	4	122 (96.72%)[1]	121 (96.69%)[2]	120 (96.67%)[3]
trigger:executeFunction	4	126 (96.83%)[4]	131 (96.95%)[5]	124 (96.77%)[6]

[1]Subject thing (Subj): 59 LoC + Object thing (Obj): 63 LoC ;
[2] Subj: 57 LoC + Obj: 64 LoC; [3] Subj: 58 LoC + Obj: 62 LoC;
[4] Subj: 63 LoC + Obj: 63 LoC; [5] Subj: 67 LoC + Obj: 64 LoC;
[6] Subj: 62 LoC + Obj: 62 LoC

For this experiment we used MQTT as a means of communication. C, Java, Arduino and MQTT are among the most used technologies in the IoT [58]. Table II depicts our results, we save *for each thing*:

- 175 LoC (94.09% gain) for C, 126 (91.97%) for JAVA and 91 (89.22%) for Arduino (CY-DSL: 11 vs. C: 186 vs. JAVA: 137 vs. Arduino: 102) with the wiring transformation.
- 233 LoC (97.89% gain) for C, 10 (64.29%) for JAVA and 19 for Arduino (79.17%) (CY-DSL: 5 vs. C: 237 vs. JAVA:14 vs. Arduino: 24) with the forwarding transformation.
- 118 LoC (96.72% gain) for C, 117 (96.69%) for JAVA and 116 (96.67%) for Arduino (CY-DSL: 4 vs. C: 122 vs. JAVA: 121 vs. Arduino: 120) with the trigger:executeFunction rule.
- 122 LoC (96.83% gain) for C, 127 (96.95%) for JAVA and 120 (96.77%) for Arduino (CY-DSL: 4 vs. C: 126 vs. JAVA: 131 vs. Arduino: 124) with the trigger:goToState rule.

The LoC correspond to the white elements in the GTRs (cf. Section VI), added automatically by CY-CGEN. Obviously, the more things are in the network, the more LoC are generated automatically, consequently saving time and bugs with the benefit of having a tangible specification of the network and a traceable transformation process. With traditional software engineering, we need to connect each thing separately (this task is automated by the wiring and forwarding M2MTs, cf. Section VI-A), and eventually make it part of a smart scenario (automated by the enforcement of policies, cf. Section VI-C). Traditional engineering is time-consuming, particularly for large networks (Task 4), and exposes software engineers to

the low-level heterogeneity that increases their chances of introducing bugs.

Also, the CY-CGEN generates automatically the textual artifacts based on M2TT. For the access control rules files of Mosquitto and RabbitMQ (two widely used MQTT brokers in the IoT), we generate approximately 60 characters *per thing and for each file* in the worst-case scenario. Factors such as the granularity of control may influence the complexity of the rules and the number of characters. More characters could understandably be saved if additional textual artifacts are needed; with a reasonable investment in time to write the plugin and the Acceleo template of the M2TT, set it up and forget it.

B. Real-World Experiment

We conducted a small scale experiment using a network of 3 things. This experiment consists of a temperature sensor, a cooling fan and a user interface. The temperature sensor should communicate the temperature to the cooling fan to adjust its speed, while the user interface should display the current temperature for monitoring purposes.

```
1  thing tempSensor import "temperatureSensor.thingml"
2  thing coolingFan import "coolingFan.thingml"
3  thing userInterface import "ui.thingml"
4  user bob
5  channel mqttChannel {
6    path tempPath(tempValue:JSON)
7  }
8  policy smartPolicy {
9    rule ts->state:isHigh trigger:goToState ac->isOn
10   rule ts->state:isLow
11   trigger:executeFunction ui->notify("Temperature is low.")
12 }
13 network experimentNetwork {
14   domain org.atlanmod.experiment
15   enforce smartPolicy
16   instance ts:tempSensor platform POSIX owner bob
17   instance cf:coolingFan platform POSIX owner bob
18   instance ui:userInterface platform POSIX owner bob
19   instance localNet:mqttChannel
20   protocol MQTT(server="mqtt.eclipse.org:1883")
21   instance externalNet:mqttChannel
22   protocol MQTT(server="mqtt.eclipse.org:1883")
23   bind tempBind: ts.tempPort => localNet{tempPath}
24   bind cf.tempPort <= localNet{tempPath}
25   forward tempBind to externalNet{tempPath}
26 }
```

Listing 13. Specification of the experiment using CY-DSL.

As shown in Listing 13, this network consists of 26 LoC in CY-DSL and includes two wirings, one forwarding, one *trigger:goToState* rule and one *trigger:executeFunction* rule. From this code our generator produces 857 LoC in C, and 540 LoC in Java, showing a reduction of an order of magnitude in LoCs.

Next, we want to give an intuition of the reduction in development time allowed by CY-DSL in our case study and offer a means to extrapolate for larger real-world networks. We measured the time required to write the code in Listing 13, and estimated the time required to write its equivalent in C and Java using the estimates provided in [59]. Writing the code in CY-DSL took an experienced developer approximately 1 person-hour (including coding and non-coding work), while writing its equivalent using C would have taken more than 17 person-hours, and using Java more than 12 person-hours. The result of this experiment emphasizes the fact that CY-DSL abstracts several redundant and time-consuming tasks (e.g., wiring, forwarding, triggers).

C. CY-CGEN Execution Time

Figure 14 shows the time required by CY-CGEN to generate the network according to the number of things. We notice that the execution time growth for low-level code generation is non-linear. Although the execution time remains rather reasonable in the context of our evaluation (90 seconds for a network of 25 things), this limitation may hinder scalability for the deployment of larger networks.

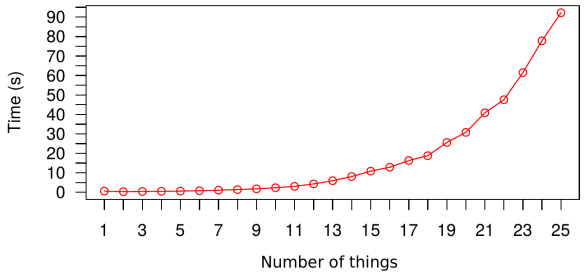

Fig. 14. CY-CGEN execution time according to the number of things.

VIII. Limitations

On the one hand, our approach suffers from some drawbacks of MDE [60], [61]. First, it may add a semantic level for software engineers, provoking resistance to its adoption. Second, it lacks a consistent integration with existing software engineering methods. Indeed, as it still lacks maturity, a software engineer may be tempted to use it partially, creating a maintainability problem. Third, as there is a separation between the specification and the implementation, there may be a fear of losing control over the code. Fourth, the approach, by its nature, advocates for a top-down development by presuming that all the network elements are known a priori, making the structure of the designed network rigid.

On the other hand, our approach needs more experimental work. First, the concepts are possibly biased; apart from our empirical experiments, we still cannot guarantee the inclusiveness (i.e., covering all possible low-level elements of a network) and genericity of these concepts. Second, we need an evaluation with real IoT engineers to assess the methodology's benefits concretely in terms of time. The current evaluation is based on the number of LoC for M2MT and on the number of characters for M2TT (as the textual artifacts are not necessarily coded) thus suggesting approximately the time that may be saved. Third, the current solution requires ThingML for the behavior of things, making the entry ticket rather expensive. However, we started working on a process for the reverse engineering of a low-level code into a TH-Model. Fourth, we need more experiments on more complex networking scenarios as the current results still do not guarantee the scalability of the methodology.

IX. Conclusion & Future Work

The present study advocates for an integrated model-based software engineering methodology to design and deploy a network of things. The methodology consists of the unified network abstractions to wire heterogeneous things, the control abstractions and mechanisms to define constraints on the network, and a code generator, based on model transformation, to process the model and generate the network artifacts. We showed that by separating the model from the low-level code, the code generator saves a significant amount of LoC, consequently saving time, automating redundant tasks and preventing bugs.

This work presents the first brick for a scalable methodology in the long run to build robust IoT applications. As future work, we plan to take full advantage of Model-Driven Reverse Engineering (MDRE) by reverse-engineering things programmed using traditional software engineering. A contribution in that sense may play a vital role in adopting our methodology. Moreover, we will continue to enrich the code generation process to cover more complex networks by exploring ways to support the generation of textual artifacts in natural language. This feature may be helpful to exempt the manual writing of any human-readable artifact, such as documentation or a security assessment report.

References

[1] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] R. Kim and S. Poslad, "The thing with e. coli: Highlighting opportunities and challenges of integrating bacteria in iot and hci," *arXiv:1910.01974*, 2019.

[3] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," 2014.

[4] F. Zambonelli, "Key abstractions for IoT-oriented software engineering," *IEEE Software*, vol. 34, no. 1, pp. 38–45, jan 2017. [Online]. Available: https://doi.org/10.1109%2Fms.2017.3

[5] D. Spinellis, "Software-engineering the internet of things," *IEEE Software*, vol. 34, no. 1, pp. 4–6, jan 2017. [Online]. Available: https://doi.org/10.1109%2Fms.2017.15

[6] M. Aly, F. Khomh, Y.-G. Guéhéneuc, H. Washizaki, and S. Yacout, "Is fragmentation a threat to the success of the internet of things?" *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 472–487, 2018.

[7] I. Berrouyne, M. Adda, J.-M. Mottu, J.-C. Royer, and M. Tisi, "A model-driven approach to unravel the interoperability problem of the internet of things," in *International Conference on Advanced Information Networking and Applications*. Springer, 2020, pp. 1162–1175.

[8] L. Farhan, R. Kharel, O. Kaiwartya, M. Quiroz-Castellanos, A. Alissa, and M. Abdulsalam, "A concise review on internet of things (iot) - problems, challenges and opportunities," in *2018 11th International Symposium on Communication Systems, Networks Digital Signal Processing (CSNDSP)*, 2018, pp. 1–6.

[9] A. Gluhak, S. Krco, M. Nati, D. Pfisterer, N. Mitton, and T. Razafindralambo, "A survey on facilities for experimental internet of things research," *IEEE Communications Magazine*, vol. 49, no. 11, pp. 58–67, 2011.

[10] T. Park, N. Abuzainab, and W. Saad, "Learning how to communicate in the internet of things: Finite resources and heterogeneity," *IEEE Access*, vol. 4, pp. 7063–7073, 2016. [Online]. Available: https://doi.org/10.1109%2Faccess.2016.2615643

[11] C. Sarkar, S. N. A. U. Nambi, R. V. Prasad, and A. Rahim, "A scalable distributed architecture towards unifying IoT applications," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*. IEEE, mar 2014. [Online]. Available: https://doi.org/10.1109%2Fwf-iot.2014.6803220

[12] A. Kazmi, Z. Jan, A. Zappa, and M. Serrano, "Overcoming the heterogeneity in the internet of things for smart cities," in *Interoperability and Open-Source Solutions for the Internet of Things*. Springer International Publishing, 2017, pp. 20–35. [Online]. Available: https://doi.org/10.1007%2F978-3-319-56877-5_2

[13] P. Patel and D. Cassou, "Enabling high-level application development for the internet of things," *Journal of Systems and Software*, vol. 103, pp. 62–84, may 2015. [Online]. Available: https://doi.org/10.1016%2Fj.jss.2015.01.027

[14] Y. Seralathan, T. T. Oh, S. Jadhav, J. Myers, J. P. Jeong, Y. H. Kim, and J. N. Kim, "Iot security vulnerability: A case study of a web camera," in *Advanced Communication Technology (ICACT), 2018 20th International Conference on*. IEEE, 2018, pp. 172–177.

[15] Trend Micro, "TrendLabs Security Intelligence BlogPersirai: New Internet of Things (IoT) Botnet Targets IP Cameras - TrendLabs Security Intelligence Blog," 2017. [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/persirai-new-internet-things-iot-botnet-targets-ip-cameras/

[16] N. Woolf, "Ddos attack that disrupted internet was largest of its kind in history, experts say," *The Guardian*, vol. 26, 2016.

[17] G. Booch, J. Rumbaugh, and I. Jacobson, *The unified modeling language reference manual*. Addison-Wesley Reading, 1999, vol. 2.

[18] I. A. Niaz and J. Tanaka, "Code generation from uml statecharts," in *Proc. 7 th IASTED International Conf. on Software Engineering and Application (SEA 2003), Marina Del Rey*, 2003, pp. 315–321.

[19] Y. G. Kim, H. S. Hong, D.-H. Bae, and S. D. Cha, "Test cases generation from uml state diagrams," *IEE Proceedings-Software*, vol. 146, no. 4, pp. 187–192, 1999.

[20] I. Berrouyne, M. Adda, J.-M. Mottu, J.-C. Royer, and M. Tisi, "CyprIoT: framework for modelling and controlling network-based iot applications," in *Proceedings of the 34th ACM/SIGAPP SAC*, 2019.

[21] X. Larrucea, A. Combelles, J. Favaro, and K. Taneja, "Software engineering for the internet of things," *IEEE Software*, vol. 34, no. 1, pp. 24–28, 2017.

[22] B. Di Martino, M. Rak, M. Ficco, A. Esposito, S. Maisto, and S. Nacchia, "Internet of things reference architectures, security and interoperability: A survey," *Internet of Things*, vol. 1-2, pp. 99–112, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660518300428

[23] J.-H. Oh, M.-K. Back, G.-T. Oh, and K.-C. Lee, "A study on dds-based ble profile adaptor for solving ble data heterogeneity in internet of things," in *Advances in Computer Science and Ubiquitous Computing*. Springer, 2016, pp. 619–624.

[24] D. Norris, *The Internet of things: do-it-yourself projects with Arduino, Raspberry Pi, and BeagleBone Black*. McGraw-Hill Education TAB, 2015.

[25] C. Doukas, *Building Internet of Things with the ARDUINO*. CreateSpace Independent Publishing Platform, 2012.

[26] A. N. Ansari, M. Sedky, N. Sharma, and A. Tyagi, "An internet of things approach for motion detection using raspberry pi," in *Proceedings of 2015 International Conference on Intelligent Computing and Internet of Things*. IEEE, jan 2015. [Online]. Available: https://doi.org/10.1109%2Ficaiot.2015.7111554

[27] M. Ibrahim, A. Elgamri, S. Babiker, and A. Mohamed, "Internet of things based smart environmental monitoring using the raspberry-pi computer," in *2015 Fifth International Conference on Digital Information Processing and Communications (ICDIPC)*. IEEE, oct 2015. [Online]. Available: https://doi.org/10.1109%2Ficdipc.2015.7323023

[28] M. Brambilla, E. Umuhoza, and R. Acerbis, "Model-driven development of user interfaces for iot systems via domain-specific components and patterns," *Journal of Internet Services and Applications*, vol. 8, no. 1, pp. 1–21, 2017.

[29] C. G. García, D. Meana-Llorián, V. García-Díaz, A. C. Jiménez, and J. P. Anzola, "Midgar: Creation of a graphic domain-specific language to generate smart objects for internet of things scenarios using model-driven engineering," *IEEE Access*, vol. 8, pp. 141 872–141 894, 2020.

[30] M. Amrani, F. Gilson, A. Debieche, and V. Englebert, "Towards user-centric dsls to manage iot systems." in *MODELSWARD*, 2017, pp. 569–576.

[31] A. Salihbegovic, T. Eterovic, E. Kaljic, and S. Ribic, "Design of a domain specific language and ide for internet of things applications," in *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*. IEEE, 2015, pp. 996–1001.

[32] A. F. Einarsson, P. Patreksson, M. Hamdaqa, and A. Hamou-Lhadj, "Smarthomeml: Towards a domain-specific modeling language for creating smart home applications," in *Internet of Things (ICIOT), 2017 IEEE International Congress on*. IEEE, 2017, pp. 82–88.

[33] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, "ThingML: a language and code generation framework for heterogeneous targets," *Proceedings of the ACM/IEEE 19th International Conference on*

Model Driven Engineering Languages and Systems - MODELS '16, pp. 125–135, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2976767.2976812

[34] A. Muelder, "Yakindu," *Yakindu Statechart Modeling Tools*, 2011.

[35] IBM Emerging Technologies, "Node-RED. A visual tool for wiring the Internet of Things," 2016. [Online]. Available: http://nodered.org/

[36] Eclipse, "Eclipse Vorto - IoT Toolset for standardized device descriptions." [Online]. Available: http://www.eclipse.org/vorto/documentation/overview/introduction.html

[37] M. M. R. Mozumdar, L. Lavagno, L. Vanzago, and A. L. Sangiovanni-Vincentelli, "Hilac: A framework for hardware in the loop simulation and multi-platform automatic code generation of wsn applications," in *Industrial Embedded Systems (SIES), 2010 International Symposium on*. IEEE, 2010, pp. 88–97.

[38] I. Malavolta, L. Mostarda, H. Muccini, E. Ever, K. Doddapaneni, and O. Gemikonakli, "A4wsn: an architecture-driven modelling platform for analysing and developing wsns," *Software & Systems Modeling*, pp. 1–21, 2018.

[39] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer networks*, vol. 38, no. 4, pp. 393–422, 2002.

[40] D. Guinard, V. Trifa, F. Mattern, and E. Wilde, "From the internet of things to the web of things: Resource-oriented architecture and best practices," in *Architecting the Internet of Things*. Springer Berlin Heidelberg, 2011, pp. 97–129. [Online]. Available: https://doi.org/10.1007%2F978-3-642-19157-2_5

[41] D. Dietterle, J. Ryman, K. Dombrowski, and R. Kraemer, "Mapping of high-level sdl models to efficient implementations for tinyos," in *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*. IEEE, 2004, pp. 402–406.

[42] T. Riedel, N. Fantana, A. Genaid, D. Yordanov, H. R. Schmidtke, and M. Beigl, "Using web service gateways and code generation for sustainable iot system development," in *2010 Internet of Things (IOT)*. IEEE, 2010, pp. 1–8.

[43] X. T. Nguyen, H. T. Tran, H. Baraki, and K. Geihs, "Frasad: A framework for model-driven iot application development," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. IEEE, 2015, pp. 387–392.

[44] R. M. Gomes and M. Baunach, "Code generation from formal models for automatic rtos portability," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 271–272.

[45] M. Hussein, S. Li, and A. Radermacher, "Model-driven development of adaptive iot systems." in *MODELS (Satellite Events)*, 2017, pp. 17–23.

[46] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, "Thingml: a language and code generation framework for heterogeneous targets," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 2016, pp. 125–135.

[47] G. Hohpe and B. Woolf, *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[48] A. Banks and R. Gupta, "Mqtt version 3.1. 1," *OASIS standard*, vol. 29, p. 89, 2014.

[49] P. Zhang, A. Durresi, and L. Barolli, "Policy-based mobility in heterogeneous networks," *Journal of Ambient Intelligence and Humanized Computing*, vol. 4, no. 3, pp. 331–338, 2013.

[50] I. Berrouyne, M. Adda, J.-M. Mottu, J.-C. Royer, and M. Tisi, "Towards model-based communication control for the internet of things," in *Federation of International Conferences on Software Technologies: Applications and Foundations*. Springer, 2018, pp. 644–655.

[51] A. Ouaddah, H. Mousannif, A. A. Elkalam, and A. A. Ouahman, "Access control in the internet of things: Big challenges and new opportunities," *Computer Networks*, vol. 112, pp. 237–262, 2017.

[52] H. Shen, "Content-based publish/subscribe systems," in *Handbook of Peer-to-Peer Networking*. Springer, 2010, pp. 1333–1366.

[53] I. Gudymenko, K. Borcea-Pfitzmann, and K. Tietze, "Privacy implications of the internet of things," in *International Joint Conference on Ambient Intelligence*. Springer, 2011, pp. 280–286.

[54] P. N. Mahalle, B. Anggorojati, N. R. Prasad, R. Prasad *et al.*, "Identity authentication and capability based access control (iacac) for the internet of things," *Journal of Cyber Security and Mobility*, vol. 1, no. 4, pp. 309–348, 2013.

[55] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the internet of things: A survey," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 414–454, 2014. [Online]. Available: https://doi.org/10.1109%2Fsurv.2013.042313.00197

[56] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE software*, vol. 20, no. 5, pp. 42–45, 2003.

[57] G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation*. WORLD SCIENTIFIC, feb 1997. [Online]. Available: https://doi.org/10.1142%2F3303

[58] E. I. W. Group *et al.*, "IEEE, Agile-IoT EU, and IoT Council. 2018. IoT Developer Survey 2018.(2018)."

[59] C. Jones, "Software economics and function point metrics," *Thirty years of IFPUG Progress. http://www. ifpug. org/wp-content/uploads*, 2017, [retrieved 2022-03-04]. [Online]. Available: https://www.ifpug.org/wp-content/uploads/2017/04/IYSM.-Thirty-years-of-IFPUG.-Software-Economics-and-Function-Point-Metrics-Capers-Jones.pdf

[60] F. Tomassetti, M. Torchiano, A. Tiso, F. Ricca, and G. Reggio, "Maturity of software modelling and model driven engineering: a survey in the italian industry," in *16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012)*. IET, 2012. [Online]. Available: https://doi.org/10.1049%2Fic.2012.0012

[61] P. Mohagheghi and V. Dehlen, "Where is the proof? - a review of experiences from applying MDE in industry," in *Model Driven Architecture – Foundations and Applications*. Springer Berlin Heidelberg, pp. 432–443. [Online]. Available: https://doi.org/10.1007%2F978-3-540-69100-6_31

Imad Berrouyne is a post-doctoral fellow in the SVV team at the Interdisciplinary Centre for Security, Reliability and Trust (SnT). He develops a model-based process to automate compliance checking of financial documents. He received a double Ph.D. degree in computer science from IMT Atlantique (France) and Université du Québec à Chicoutimi (Canada) in 2021 for his contributions on using MDE for the IoT. His current research interests include the IoT, MDE, and Compliance Checking.

Mehdi Adda is a professor of Computer Science at Université du Québec à Rimouski (UQAR), Canada since June 2010. From August 2008 to May 2010, he was an invited professor at the same university. He has been working on multidisciplinary projects centered around Data Science and Artificial Intelligence, IoT, and Cybersecurity. Mehdi Adda obtained two Ph.D. degrees in Computer Science from Université de Montréal, Canada and Université de Lille, France in 2008. He received two MSc. degrees in Computer Science from Joseph Fourier University in 2002 (Grenoble, France), and Université du Havre (Le Havre, France) in 2003 as well as an Engineering degree in Computer Science from Université des Sciences et Technologies Houari Boumediene (Algiers, Algeria) in 2001.

Jean-Marie Mottu received the Ph.D. degree in computing science from University of Rennes 1, France, in 2008 for his work on testing model transformations. He is an Associate Professor at the University of Nantes, France. His current research interests include Model Driven Engineering (MDE), Domain-Specific Languages (DSLs), Software Quality, Test Verification considering functional and non-functional properties.

Massimo Tisi is an associate professor in the Department of Computer Science of the Institut Mines-Telecom Atlantique (IMT Atlantique, Nantes, France), and deputy leader of the NaoMod team, LS2N (UMR CNRS 6004). Since 2019 he coordinates the Lowcomote Marie Curie European Training Network. He has been visiting researcher at McGill University and the National Institute of Informatics (NII) in Japan, and post-doctoral fellow at Inria. He received his Ph.D. degree in Information Engineering at Politecnico di Milano (Italy), where he was a member of the Database and Web Technologies group. His research interests revolve around software and system modeling, domain-specific languages and applied logic. He contributes to the design of the ATL model-transformation language and investigates the application of deductive verification techniques to model-driven engineering.