A Model-Driven Approach to Unravel the Interoperability Problem of the Internet of Things

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

A Model-Driven Approach to Unravel the Interoperability Problem of the Internet of Things

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

Abstract The Internet of Things (IoT) aims for connecting Anything, Anywhere, Anytime (AAA). This premise brings about heterogeneity that creates connectivity challenges. These challenges constitutes a serious obstacle to interoperability between things. Most existing approaches tackles the interoperability problem by avoiding heterogeneity with standards at runtime. While heterogeneity is an intrinsic feature of the IoT, there is a need for an approach that embraces it to connect AAA. In this paper we propose a model-based methodology to tackle the interoperability problem. It relies on a Domain-Specific Language (DSL) for a model-based specification of the network and a transformation process to generate the network artifacts from this specification. The principle consists of achieving interoperability at the model-level, then during a transformation process, ensuring that it is preserved in the low-level code. Adopting this methodology makes the engineering of the IoT more rigorous, prevents bugs earlier and saves time.

1 Introduction

The Internet of Things (IoT) aims for connecting Anything, Anywhere, Anytime (AAA) [2]. In particular, connecting things from different sizes ranging from bacterias [14] to supercomputers. Each thing has its own requirements. For instance, a tiny sensor may require Constrained Application Protocol (CoAP), because of its limited resources, while a laptop requires a standard Hypertext Transfer Protocol (HTTP) client. The premises of the IoT are that (1) any thing with a computing power (2) can connect to the Internet. While the former premise (1) is hardware-

[1] Naomod Team, IMT Atlantique, LS2N, Nantes, France — firstname.lastname@ls2n.fr
[2] Mathematics, Computer Science and Engineering Dep. University of Quebec At Rimouski, Rimouski, QC, Canada — firstname_lastname@uqar.ca
[3] UQAC, 555 Boulevard de lUniversite, Chicoutimi, QC, Canada — firstname_lastname@uqac.ca

related, the latter (2) could be tackled using an appropriate software engineering approach.

Engineering IoT applications is hard. On the one hand, many stakeholders are involved (e.g., Security, Business, Network), each using its own tools and methods. On the other hand, heterogeneity (e.g., languages, protocols) [1] causes, inter alia, an interoperability problem, by hindering communication between things.

Interoperability is an important milestone, to unlock the full potential of the IoT [20, 17]. Many approaches tackle the interoperability problem by avoiding heterogeneity, either with standards [3, 7] or a layer at runtime responsible for communication [13, 16].

While standardization is sufficient in a homogeneous context (e.g., HTTP browsers, specific range of things), the challenge of connecting AAA requires a more generic and inclusive engineering solution that embraces heterogeneity to solve the interoperability problem. Model-Driven Engineering (MDE) shares this commonality (i.e. genericity) with the IoT. Models offer the ability to unify the low-level concepts at higher-level, i.e. model-level. These models can thereafter be used to generate various artifacts. For instance, the Unified Modelling Language (UML) is a generic modeling language to design, using models, any Object Oriented (OO) application. UML models find their application in illustration purposes, code generation [18] or test cases generation [15] to cite a few. In this paper, we are heading towards the same goal, but for the IoT.

Moreover, tackling the problem of interoperability at low-level, i.e. adapting the code of heterogeneous things to interoperate, tends to be time-consuming and bug-prone and complicates communication between stakeholders. Whereas, going at the model-level, offers platform-agnostic abstractions to tackle the problem in a unified manner.

The contribution of this paper is a model-based methodology to tackle the interoperability problem of the IoT that relies on a Domain-Specific Language (DSL) based on models and a transformation process [22]. Indeed, it consists of achieving interoperability at the model-level, then during the transformation process, ensuring that interoperability is implemented in the concrete low-level code.

This paper is structured as follows. Section 2 presents a simple smarthome example as an illustrative test case for our methodology. Section 3 presents the DSL to specify the smarthome network. Section 4 shows how model transformation bridges the gap between the model-based specification of the network and its concrete implementation. Section 5 provides an overview of the related work. Finally, Section 7 presents the discussion, conclusion and future work.

2 Running Example: A Simple Smarthome

To illustrate our study, we use the example of a simple smarthome (Cf. Figure 1). It contains heterogeneous things communicating via heterogeneous paths, possibly indirect path (thicker dashed line in the figure). A path is similar to an edge in

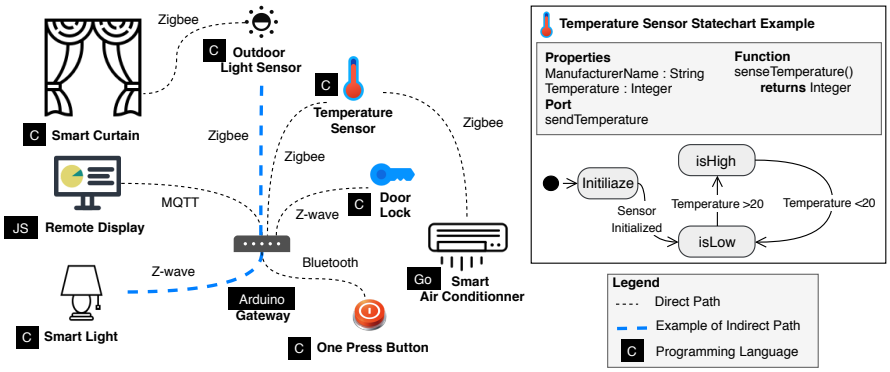

Fig. 1 A Smarthome Example

the communication graph, it offers the possibility to reach a thing via a protocol. The specification of the behavior of a thing is a statechart (an example is shown in Figure 1), consisting of a finite number of states and event-based transitions. Each thing can have states, functions (i.e. sequence of instructions achieving a goal), properties (i.e. variables containing information about the thing) and ports (i.e. for sending and/or receiving messages).

In the following subsections, we point out the difficulties to implement this example. We identify two main categories of problems that can contribute to better interoperability if done seamlessly. It should be noted that in the scope of our study, we presume that common issues proper to distributed systems (e.g. deadlock, synchronization) are either absent or handled by low-level protocols.

2.1 Seamless Networking

The first category consists of connecting things regardless of their implementation (e.g., languages, protocols). For instance, the Smart Air Conditioner (SAC) needs to receive the current temperature from the Temperature Sensor (TS) to adjust cooling. In this exchange, we need to ensure that both things are interoperable[23], i.e. can communicate either directly or indirectly, regardless of their protocols and/or programming languages. Implementing this simple exchange requires ensuring, separately, that the TS and the SAC have a way to exchange information, either directly using the same protocol or indirectly using an intermediary thing. When they use the same protocol, they may also be programmed using heterogeneous programming languages, thus leading to repeat the same task (i.e. implementing the protocol) for two programming languages. Such manual engineering is bug-prone, especially for large networks. Indeed, bugs may be introduced when linking the SAC and the TS.

These bugs may be discovered only once deployed, and late software bugs detection is more expensive [8, 24].

The intrinsic heterogeneity of the IoT creates barriers to interoperability between ranges. For instance, because of its limited resources the TS cannot send data directly to the Remote Display (RD), using Message Queuing Telemetry Transport (MQTT) (a relatively more energy-hungry protocol). We need a bridge between them. Implementing correctly such bridge is not trivial, as many expertise, platforms, protocols and hidden risks are involved. There is a need for a Cross-Range Interoperability (CRI) mechanism.

The lack of consensus for a standard, in addition to proprietary technologies, leads to interoperability issues within the same range. For instance, the Smart Curtain (SC) and the Outdoor Light Sensor (OLS) both support Zigbee, they can therefore communicate directly. The Smart Light (SL) supports Z-wave, and needs to go to the state *isOn* or *isOff* according to the OLS sensed light. Although they both use the same message format and belong to the same range, the SL and the OLS cannot communicate directly due the incompatibility of their protocols. We need to use the Gateway (GW), because it supports both protocols, as a pivot for an indirect path (thicker dashed line), to forward the message from the OLS to the the SL. Implementing this mechanism at low-level is also not always trivial.

These few examples show that such networking, yet simple, can be hard to implement concretely because of the lack of interoperability mechanisms.

2.2 Seamless Smart Scenarios

The second category consists of seamless collaboration, i.e. a thing may impact the behavior of another thing. By lacking a common representation of the behavior, things cannot interoperate to achieve a smart scenario[1]. The advantage of a statechart-based behavior is that it enables controlling the behavior using states according to the context. For instance, when the TS is on the state *high*, the SAC has to be on the state *isOn*. Thus, the SAC can adjust cooling according to the TS.

Also, as a thing can provide a function (sequence of instructions), there are cases where a function may need to be executed depending on the state of another thing. For instance, when the state of the OLS is *medium*, the function *setIntensity(intensityLevel : Integer)* of the SL should be executed with the parameter 50 (value for a medium lighting intensity).

In a more complex scenario, we may need to specify that when the One Press Button (OPB) is on the state *isOn*, the system should ask the SL to be on the state *isOn*, the SAC to execute the function *setTemperatureTo(25)* and the Door Lock (DL) to go to the state *close*.

It is important that the specification of such smart scenarios is not impacted by the low-level technical details, such as the communication means or the programming

[1] The concept of smart scenario refers to the ability to achieve a [smart] goal using multiple contextual factors (e.g., states of a thing, properties of a thing, space, time)

languages that are often heterogeneous. The heterogeneity may distract us from achieving the interoperability that is enabling these smart scenarios.

3 System Specification

This section describes the specification of the things and of the network to connect them. A network consists of two activities: specification of the behavior of a thing (Cf. Subsection 3.1) and specification of the network (Cf. Subsection 3.2).

3.1 Thing Behavior Specification

We presume that the behavior of a thing is specified using ThingML-DSL (TH-DSL)[2] [12]. TH-DSL specifies a statechart-based behavior (Cf. statechart example in Figure 1), and functions that can be called within a state. TH-DSL provides a syntax based on the Xtext grammar, *representing the textual form of a model*, called ThingML-Model (TH-Model). The latter is based on the Eclipse Modeling Framework (EMF). EMF is a state-of-art standard framework, maintained by Eclipse, to design models.

TH-DSL offers the concept of *external connector* to enable a thing to send or receive a message using a specific serialization format and protocol. The ThingML Code 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).

It is worth noting that when it is not possible to express an instruction using TH-DSL syntax, TH-DSL permits to embed 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.

In summary, ThingML is useful for us to specify the behavior of a thing in the form of statechart with a communication interface and the generation of its equivalent in the low-level code using TH-CGEN.

3.2 Primitive Concepts of Networking

We conceived CyprIoT-DSL (CY-DSL)[3] [4] for the specification of the network in an editor. CY-DSL is also based on the Xtext grammar, hence it is a textual repre-

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

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

sentation of an EMF model, called CyprIoT-Model (CY-Model). The full grammar could be found on Github [4]. The goal of CY-DSL is to define the primitive concepts to specify a network of things in a unified manner. These concepts are abstract representations of low-level networking concepts.

EMF offers a mechanism, called inter-model referencing, that enables to use other EMF models. We use this mechanism to import the TH-Model inside the CY-Model. Line 1 of Listing 2 shows how we import TH-Model of the SAC. It consists of a name as identifier (i.e. airConditioner) and the relative path in disk of the TH-Model (i.e. "airConditioner.thingml").

```
1  user bob
2  user monitor
3
4  network smartHomeNetwork {
5    domain org.atlanmod.smarthome
6    enforce smartPolicy
7    // Instances of things
8    instance myOLS : outdoorLightSensor platform POSIX owner bob
9    instance mySL : smartLight platform POSIX owner bob
10   // Instance of a channel
11   instance lightZigbee:lightChannel protocol ZIGBEE
12   // Binding : Sending (i.e. =>) the sensed light by myOLS to a Zigbee channel
13   bind myOLS.sendingSensedLightPort => lightZigbee{sensedLightPath, logsPath}
14   // Binding : Receiving (<=) the sensed light in mySL
15   bind mySL.receivingSensedLightPort <= lightZigbee{sensedLightPath}
16   ...
17 }
```

Listing 1 Specification of a network; Sending (=>) and Receiving (<=) Messages via a path

As shown in Listing 1, a **network** has an identifier (smartHomeNetwork) and a **domain** (org.atlanmod.smarthome). Inside the **network**, we can declare an **instance** of a thing (based on the imported TH-Model). An **instance** consists of an identifier (e.g., myGW, myOLS, mySL), the **platform** specifying the target programming language to generate the low-level code by TH-CGEN (e.g., **ARDUINO**, C/**POSIX** based), and if necessary its **owner** (e.g., bob).

We can also declare an **instance** of a **channel**. Listing 2 shows an example of a **channel**, containing two paths, where the second is a sub-path of the first. An instance of **channel** sets a **protocol** (e.g., **MQTT**, **ZIGBEE**). The concept of **channel** decouples the communication of things (path and protocol) from their programming languages, thus enabling seamless networking at the model-level. The concept of **path** offers a uniquely identified way to exchange messages, while a sub-path enables to organize paths in the form of a tree. It is inspired from existing standard protocols, for instance, in MQTT a message is exchanged via a topic, while in HTTP via a Uniform Resource Locator (URL). In fact, both (topic and URL) can be unified under the concept of **path**. As shown in Listing 2, a **path** consists of an identifier

[4] CyprIoT Gihub > /language/org.atlanmod.cypriot/src/org/atlanmod/cypriot/Cypriot.xtext

(sensedLightPath), declaration of the accepted message (lightSensorMessage) and the message serialization format (**JSON**). Hence, an exchange via a **path** is transparent, thus easing detection of incompatibility of channels and messages from the editor.

Inside the **network**, we can specify to **bind** a port of an **instance** of thing to one or multiple paths (e.g., Line 13 of Listing 1). We can also **forward** an existing binding to another **path** via an intermediary thing. For instance, in Line 11 of Listing 3, we forward the temperature received by myGW via **ZIGBEE** to tempMQTTPath, a path of myMQTTChannel that is using **MQTT**. Then, we bind the port receivingTemperaturePort of myRD to receive the message sent to tempMQTTPath. myGW plays the role of an intermediary thing between myOLS and myRD. The same principle is used to forward messages from the OLS (supporting Zigbee) to the SL (supporting Z-wave) as indicated in the running example.

This subsection shows that we can specify a network from an editor, using primitive concepts, without being distracted by the low-level technical details.

3.3 Policy Specification

Generally speaking, the purpose of a policy (Cf. Listing 5) is to control the behavior of a network. We focus in this paper on its ability to specify smart scenarios. A **policy** consists of a set of rules. A **rule** (Cf. syntax in Listing 4) aims to control in a readable way the network using its inner elements (e.g., channels, paths, users, instances). Specifying these rules at the model-level allows to decouple the business logic and safety from the concrete implementation (source of heterogeneity). We show a few examples of such rules in this subsection.

Figure 5 shows a **policy**. It consists of an identifier (i.e. smartPolicy) and two rules. This policy is enforced in smartHomeNetwork (Line 6 of Listing 1).

Rule in Line 2 specifies that the state isLow of myOLS (i.e. **instance** of the OLS), triggers mySL (i.e. **instance** of the SL) to be at the state isOn (for an optimal lighting). Rule in Line 3 specifies that the state isHigh of myTS (i.e. **instance** of the TS), triggers the function setTemperature(25) of mySAC (i.e. **instance** of the SAC) (thus, setting the temperature to 25°C when it's warm).

Specifying a policy is readable and relieved from the low-level technical details. A policy can serve various purposes [25] (e.g., communication control, administrative goal), we present only its smart aspects as it is the focus of this paper.

```
1 thing airConditioner import "airConditioner.thingml"
2
3 channel lightChannel {
4   path bobSmartHome
5   path sensedLightPath (lightSensorMessage:JSON) subpathOf bobSmartHome
6 }
```

Listing 2 Importing a thing (Line 1); Declaring channel, path and sub-path (Line 3 to 6)

```
1  network smartHomeNetwork {
2    domain org.atlanmod.smarthome
3    ...
4    instance myGW : gateway platform ARDUINO owner bob
5    instance myRD : remoteDisplay platform JAVASCRIPT owner monitor
6    instance myMQTTChannel:RDChannel protocol MQTT(server="mqtt.atlanmod.org
7                                                                    :1883")
8    // "lightBind" is an optional identifier for the bind
9    bind lightBind :  bind myGW.sensedLightPort <= lightZigbee{sensedLightPath}
10   // Forwarding the binding "sensedLightBind" to an MQTT channel
11   forward lightBind to myMQTTChannel{lightMQTTPath}
12   // Receiving Sensed Light in the Remote Display via MQTT
13   bind myRD.receivingLightPort <= myMQTTChannel{lightMQTTPath}
14 }
```

Listing 3 Forwarding an existing binding (continuation of Listing 1)

```
1  rule <Subject> <ActionType>:<Action> <Object>
```

Listing 4 Rule syntax

Ensuring its automatic enforcement is the concern of the Transformation Process (T-PROCESS), assumed by experts. Experts are responsible for the mapping of the abstract concepts at the model-level into their concrete representation at the low-level.

4 Transformation Process

Model-to-Model Transformation (M2MT) is a process that takes one or more input models to produce automatically a *target model* based on transformation rules (usually specified by experts).

M2MT allows us to adapt 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. It takes information from the CY-Model and adds *only what is needed* to the TH-Model, so as to be conform to the specification of the network. As this process takes place at the model-level (using abstract concepts), interoperability is preserved. The transformed TH-Models are thereafter used to generate their equivalent in the low-level code using TH-CGEN.

```
1  policy smartPolicy {
2   rule myOLS->state:isLow trigger:goToState mySL->isOn
3   rule myTS->state:isLow trigger:executeFunction mySAC->setTemperature(25)
4  }
```

Listing 5 Go to a state (Line 2) and execute a function (Line 3) according to another thing

As shown in Figure 2, we use AtlanMod Transformation Language (ATL)[5] for M2MT, a state-of-the-art transformation tool. ATL rules are depicted using Graph Transformation Rules (GTR) for illustrative purposes. 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 concepts of the TH-DSL syntax.

We show three examples of M2MT: (1) *adding the communication interface according to the network* (i.e. protocol and path), (2) *forwarding an existing binding* and (3) *applying the trigger:executeFunction rule*. (1) and (2) tackle the networking problems while (3) tackles the smart scenarios. It is worth noting that these examples are meant to show the principle of M2MT. We developed other M2MTs aiming to address other interoperability issues (e.g., *applying the trigger:goToState rule*, communication control), that we could not include for space reasons.

The upper part of Figure 3 depicts the GTR of (1). On the RHS, elements in white (i.e. *MyExternalConnector*, *MyProtocol* and some annotations) are added according 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 MQTT : broker address, port, topic, 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 1, we add 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="tempMQTTPath". This gives TH-CGEN all the information needed to generate the low-level code containing the correct communication interface of 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 applies in the case of Java or Arduino to cite a few. Thus, as the low level code is a mere translation of the TH-Model, interoperability is preserved.

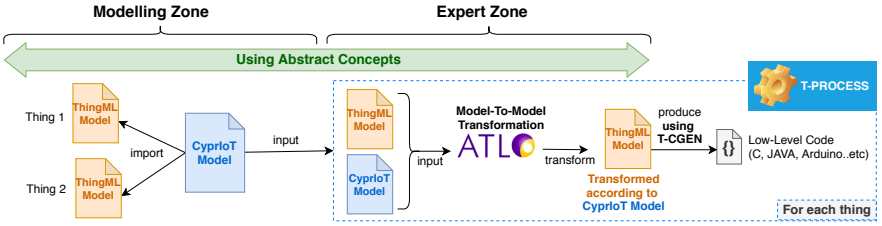

Fig. 2 Generation of network artifacts using the T-PROCESS; Modeling Zone: specification of the network and things; Expert Zone: interpretation of abstract concepts into low-level concepts by experts for the generation of artifacts

[5] https://www.eclipse.org/atl/

The T-PROCESS applies this M2MT to all things. This transformation enables to connect things in a network.

The lower part of Figure 3 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. protocol to be used for forwarding). We omitted adding the annotations (that are similar to (1)) of the *New_ExternalConnector* and *New_Protocol* to improve the readability of the GTR. 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 seamless CRI pointed out in the running example, via an intermediary thing.

Figure 4 depicts the GTR of (3) (Cf. Line 3 of Listing 5). To enforce this rule two M2MTs are applied, one to the subject thing and the other to the object thing. This transformation adds the necessary elements to make the subject thing sends (*MessageSendAction*) the message (*CommandMessage*) informing that it entered (*OnEntry*) the state in question. An event waits for the message (*ReceiveMsgEvent*), inside all states of the object thing. An event is added to all states to ensure that the function can be executed regardless of the current state of the object thing. Once the object thing receives the message, it executes the function (*ExecFunction*) with the specified *Parameter* in the rule. We try to send the message using an existing path between the two things, i.e. if they are already expected to exchange some information via a path. If no direct path exists between them, we try to find an

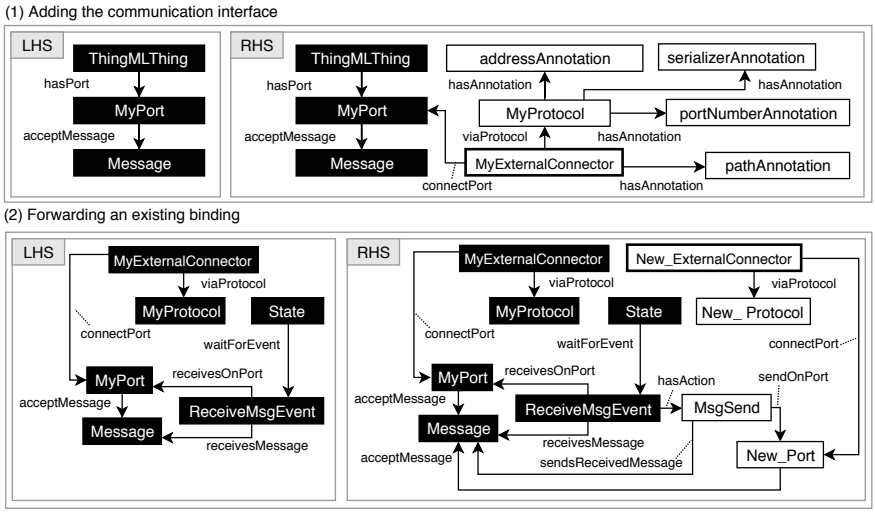

Fig. 3 Upper Part: Adding the communication interface according to the specification of the network GTR (1); Lower Part: Forwarding an existing binding GTR (2); White boxes are added based on the CY-Model; The added External Connector box is thicker for readability purpose

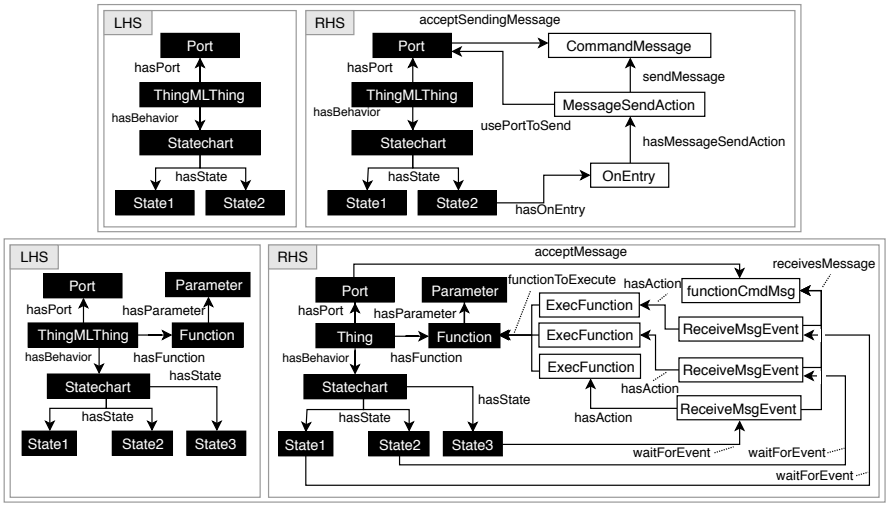

Fig. 4 Upper Part for Subject, Lower Part for Object; Applying the trigger:executeFunction rule GTRs (3) with the subject thing having two states and the object thing having three states; White boxes are added based on the CY-Model

indirect path to send the *CommandMessage*. This transformation enforces a smart scenario at the model-level. The *trigger:goToState rule* uses the same principle, but adds insides all states of the object thing a transition to the state to go to (instead of a *ExecFunction*).

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 2) 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.

5 Related Works

Many approaches tend to contain the interoperability problem using standardization (e.g., standard architecture, standard platform) [6, 21, 10, 9, 5]. Standards are not quite inclusive, as certain things would not afford their cost (e.g., cost of adoption, computing resources required) [14]. We believe that this is not a scalable strategy for the IoT to reach its ultimate goal, i.e. connecting AAA. We propose a generic engineering methodology based on MDE that is embracing heterogeneity and aiming to include any possible networking scenario ultimately.

ThingML [12] abstracts the heterogeneity of things by representing their behavior as a statechart, using TH-DSL. The TH-CGEN generates the low-level code for various programming languages (e.g., JAVA, C, Javascript). However, it only

proposes to connect an independent thing (i.e. by presuming that it is not part of a network) using an external connector to tackle networking. We propose abstract networking concepts to design a network of things.

Node-Red [13] allows to connect things, APIs and online services using a browser-based editor. It presumes that the nodes are already deployed. If Node-Red is used as an orchestration tool (i.e. wiring existing things), then it constitutes a single point of failure. In addition, it is rather hungry and could not be deployed in small devices. Our methodology aims to tackle the problem at the engineering phase and covers theoretically any device without imposing a standard or an additional infrastructure to deploy at runtime.

The fifth generation of cellular network (5G) covers connectivity of resource-constrained devices [19], namely for Low Power Wide Area (LPWA), but sill requires using different protocols (NB-IoT, eMTC) that fit best the requirements of this category of devices. Moreover, standardization takes time and has high upfront costs to be adopted at large scale by providers. Indeed, heterogeneity will undeniably persist, hence the need to embrace it instead of containing it in a standard.

6 Evaluation, Results & Discussion

We tested our methodology on networks ranging from 1 to 50 things. As our focus is on networking, we presume that the behavior of things is not part of this evaluation. We compare only the Lines of Code (LoC) needed for networking and smart scenarios, between our model-based methodology, and traditional engineering. We use C based things with MQTT as a communication mean to represent traditional engineering. C and MQTT are among the most used technologies in the IoT [11]. Our results show that, we save *for each thing*, 298 LoC (CY-DSL: 11 vs. C: 309) with M2MT (1), 287 LoC (1 vs. 288) with (2) and 164 LoC (3 vs. 167[6]) with (3), in addition to the benefit of having a specification of the network and a traceable transformation process. LoC correspond to the elements in the white of the GTRs that are added automatically after transformation. The more things are in the network, the more LoC are saved, and consequently time and bugs. With traditional engineering, we need to connect each thing separately (automated by (1) and (2)), and eventually make it part of a smart scenario (automated by (3)) as well as dealing with the low-level heterogeneity.

We provide an evaluation based on the number of LoC suggesting the engineering time that may be saved. As a future work we plan for an evaluation based on a real engineering experience. Also, the current solution requires ThingML but we are currently working on reverse engineering low-level code into a TH-Model.

Solving the interoperability problem in the IoT for any possible scenario is difficult. We showed that MDE offers promising tools to, at least, unravel the problem rigorously. Thus, enabling easier networking and smart scenarios. MDE could help

[6] Subject thing: 66 LoC - 0bject thing: 101 LoC

to contain the problem without scarifying the freedom to use the technology that works best for the thing.

7 Conclusion

The interoperability problem of the IoT causes poor collaboration between things, because of difficulties in networking and smart scenarios between heterogeneous things. Heterogeneity is intrinsic to the IoT. There is a need to embrace it to solve the interoperability problem. In that sense, we proposed a model-based engineering methodology that abstracts the low-level networking concepts (source of heterogeneity) into more inclusive concepts based on models, thus freeing up the IoT application from heterogeneity, and consequently leading to seamless interoperability at the model-level. The methodology makes the engineering more rigorous, prevents bugs earlier and saves times.

In the future, we plan to add more abstractions to the CY-DSL (e.g., spacio-temporal features), improve the T-PROCESS to interpret more complex networking scenarios, develop mechanisms to prevent common distributed systems issues at low-level (e.g. deadlock, synchronization) and support the generation of textual artifacts (e.g., documentation, configuration file).

8 Acknowledgements

We acknowledge the support of Institut Mines-Télécom Atlantique and the Natural Sciences and Engineering Research Council of Canada (NSERC), 06351.
Cette recherche a été financée par l'Institut Mines-Télécom Atlantique et le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), 06351.

References

1. Mohab Aly, Foutse Khomh, Yann-Gaël Guéhéneuc, Hironori Washizaki, and Soumaya Yacout. Is fragmentation a threat to the success of the internet of things? *IEEE Internet of Things Journal*, 6(1):472–487, 2018.
2. Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.
3. July Katherine Díaz Barriga, Christian David Gómez Romero, and José Ignacio Rodríguez Molano. Proposal of a standard architecture of iot for smart cities. In *International Workshop on Learning Technology for Education Challenges*, pages 77–89. Springer, 2016.
4. Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Jean-Claude Royer, and Massimo Tisi. CyprIoT: framework for modelling and controlling network-based iot applications. In *Proceedings of the 34th ACM/SIGAPP SAC*, 2019.

5. Arne Bröring, Stefan Schmid, Corina-Kim Schindhelm, Abdelmajid Khelil, Sebastian Käbisch, Denis Kramer, Danh Le Phuoc, Jelena Mitic, Darko Anicic, and Ernest Teniente. Enabling iot ecosystems through platform interoperability. *IEEE software*, 34(1):54–61, 2017.
6. Valentina Casola, Luca DOnofrio, Giusy Di Lorenzo, and Nicola Mazzocca. A service-based architecture for the interoperability of heterogeneous sensor data: A case study on early warning. In *Geographic Information and Cartography for Risk and Crisis Management*, pages 249–263. Springer, 2010.
7. Soumya Kanti Datta, Rui Pedro Ferreira Da Costa, Christian Bonnet, and Jérôme Härri. onem2m architecture based iot framework for mobile crowd sensing in smart cities. In *2016 European conference on networks and communications (EuCNC)*. IEEE, 2016.
8. Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009.
9. Giancarlo Fortino, Claudio Savaglio, Carlos E Palau, Jara Suarez de Puga, Maria Ganzha, Marcin Paprzycki, Miguel Montesinos, Antonio Liotta, and Miguel Llop. Towards multilayer interoperability of heterogeneous iot platforms: The inter-iot approach. In *Integration, interconnection, and interoperability of IoT systems*, pages 199–232. Springer, 2018.
10. Ivan Gojmerac, Peter Reichl, Ivana Podnar Žarko, and Sergios Soursos. Bridging iot islands: the symbiote project. *e & i Elektrotechnik und Informationstechnik*, 133(7):315–318, 2016.
11. Eclipse IoT Working Group et al. IEEE, Agile-IoT EU, and IoT Council. 2018. IoT Developer Survey 2018.(2018).
12. Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. Thingml: a language and code generation framework for heterogeneous targets. In *Proceedings of the ACM/IEEE 19th MODELS Conference*, 2016.
13. IBM Emerging Technologies. Node-RED. A visual tool for wiring the IoT, 2016.
14. Raphael Kim and Stefan Poslad. The thing with e. coli: Highlighting opportunities and challenges of integrating bacteria in iot and hci. *arXiv:1910.01974*, 2019.
15. Young Gon Kim, Hyoung Seok Hong, Doo-Hwan Bae, and Sung Deok Cha. Test cases generation from uml state diagrams. *IEE Proceedings-Software*, 146(4):187–192, 1999.
16. K Kreuzer et al. Openhab-empowering the smart home. *Openhab. org, Tech. Rep.*, 2013.
17. James Manyika, Michael Chui, Peter Bisson, Jonathan Woetzel, Richard Dobbs, Jacques Bughin, and Dan Aharon. Unlocking the potential of the internet of things. *McKinsey Global Institute*, 2015.
18. Iftikhar Azim Niaz and Jiro Tanaka. Code generation from uml statecharts. In *Proc. 7 th IASTED International Conf. on Software Engineering and Application (SEA 2003), Marina Del Rey*, pages 315–321, 2003.
19. Maria Rita Palattella, Mischa Dohler, Alfredo Grieco, Gianluca Rizzo, Johan Torsner, Thomas Engel, and Latif Ladid. Internet of things in the 5g era: Enablers, architecture, and business models. *IEEE Journal on Selected Areas in Communications*, 34(3):510–527, 2016.
20. Keyur K Patel, Sunil M Patel, et al. Internet of things-iot: definition, characteristics, architecture, enabling technologies, application & future challenges. *International journal of engineering science and computing*, 6(5), 2016.
21. Stefan Schmid, Arne Bröring, Denis Kramer, Sebastian Käbisch, Achille Zappa, Martin Lorenz, Yong Wang, Andreas Rausch, and Luca Gioppo. An architecture for interoperable iot ecosystems. In *International Workshop on Interoperability and Open-Source Solutions*, pages 39–55. Springer, 2016.
22. Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE software*, 20(5):42–45, 2003.
23. Peter Wegner. Interoperability. *ACM Comput. Surv.*, 28(1):285–287, March 1996.
24. Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *12th USENIX OSDI proceedings*, 2016.
25. Ping Zhang, Arjan Durresi, and Leonard Barolli. Policy-based mobility in heterogeneous networks. *Journal of Ambient Intelligence and Humanized Computing*, 4(3):331–338, 2013.