CyprIoT: framework for modelling and controlling network-based IoT applications
Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Jean-Claude Royer, Massimo Tisi

To cite this version:
Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Jean-Claude Royer, Massimo Tisi. CyprIoT: framework for modelling and controlling network-based IoT applications. the 34th ACM/SIGAPP Symposium, Apr 2019, Limassol, Cyprus. 10.1145/3297280 . hal-02333578v2

HAL Id: hal-02333578
https://hal.archives-ouvertes.fr/hal-02333578v2
Submitted on 28 Oct 2019

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

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
CyprIoT : Framework for Modelling and Controlling Network-Based IoT Applications

Imad Berrouyne1,3, Mehdi Adda3, Jean-Marie Mottu2, Jean-Claude Royer1, Massimo Tisi1
1Naomod Team, IMT Atlantique, LS2N (UMR CNRS 6004)
2Naomod Team, Université de Nantes, LS2N (UMR CNRS 6004)
Nantes, France
3Mathematics, Computer Science and Engineering Dep. University of Quebec At Rimouski
Rimouski, QC G5L 3A1, Canada

ABSTRACT
Model-Driven Engineering (MDE) is a paradigm that favors using models to address software engineering problems. Very few attempts have been made to apply this paradigm to the Internet of Things (IoT). Most of the existing MDE approaches focus on abstracting the heterogeneity of IoT things while neglecting network communication heterogeneity. In fact, few attempts target network-based IoT applications. In this paper, we propose a framework, called CyprIoT, to model and control network-based IoT applications using MDE techniques. Our approach relies on 1) Networking Language, to model a network of IoT things 2) Rule-Based Policy Language, to control and supervise the behavior of the modeled network 3) Code Generator, to interpret the model and generate deployable network artifacts and 4) Plug-in System, to customize, enhance or implement expert knowledge into the generated artifacts.

KEYWORDS
Internet of Things, Model-Driven Engineering, Domain-Specific Language, Code Generation

ACM Reference Format:

1 INTRODUCTION
The IoT is a modern paradigm disrupting the way how objects and people communicate. Prominent applications such as smarthomes [16] make the IoT more and more visible in our everyday life. It is expected to see even more IoT applications in the near future. According to Gartner, more than 8 billion connected IoT devices are already in use, and it is expected that this number will grow to 20.4 billion by 2020 [10].

Even though the IoT generates a lot of hype, few engineering models have been proposed to meet its requirements. The IoT brings about new engineering challenges due to a multifaceted heterogeneity in the involved technologies. In a typical IoT application, many computing platforms, languages and communication protocols may be used. In addition, everyday a new IoT device emerges, with often non-standard features. Although this low-level heterogeneity seems problematic, it constitutes the differentiating factor between the IoT and the traditional internet. Indeed, many research studies [15, 34] suggest that it is even necessary to connect IoT things from different ranges.

Understandably, this heterogeneity requires more human resources and expertise at different levels. Most companies, often with limited human resources fail to consider those aspects in their IoT applications. This often results into flawed applications that may lead to large-scale network attacks such as Mirai and Persirai [31, 33] targeting numerous IoT things. As a matter of fact, for security experts [27], existing engineering models based on a quick-fix approach, have shown their limits w.r.t security. Moreover, the SANS Institute reports that almost 90% of security professionals affirm that changes to security controls are required when it comes to the IoT [25].

MDE is a promising paradigm having the potential to overcome such issues (i.e. platforms, languages and communication heterogeneity and control mechanisms implementation). Using abstractions, MDE eases and automates software engineering. It can help in designing robust and reliable IoT systems by abstracting features such as communication means and by providing model-based mechanisms for control. Using MDE, an engineer can design a complete IoT application in a unified manner using abstract concepts, thus glossing over the low-level details. Afterwards, a code generation procedure can interpret the model and generate the deployable code.

All the more so that recently MDE has successfully been applied to adaptive and distributed systems, by the Model @Runtime approach [6] as well as in model-driven security [3]. Moreover, it can also enable reasoning formally on large IoT models for various purposes such as security analysis and threat assessment [22].

In this paper we introduce a framework, called CyprIoT, for modelling and controlling a network-based IoT application [2]. By Modelling we refer to the ability to describe a
network of connected IoT things beside its concrete implementation, while by Controlling, in the other hand, we refer to the ability to determine and supervise the behavior of the modeled network and devices beside their implementations. The proposed framework favors the separation of concerns by providing two separate languages, one for modelling a network of IoT things and the second to control it using Rule-Based Policies. It also provides a Plug-in System to offer a convenient means of implementing expert knowledge in the generated artifacts. Finally, a modular Code Generator is meant to generate the deployable code. The source code and the Xtext grammar \(^1\) of the concrete textual syntax of the languages are available on Github \(^2\). To the best of our knowledge this is the first open-source model-based framework to target generic network-based IoT applications.

In this paper we propose to investigate the following research questions (RQs):

- **RQ1**: Can MDE help to design a Network-Based IoT application?
- **RQ2**: Can MDE help an IoT engineer to abstract low-level details and expertises?
- **RQ3**: Is MDE suitable to control a Network-Based IoT application using realistic scenarios?

This paper is structured as follows. Section 2 presents a typical IoT usecase that will be used throughout the paper to illustrate our solutions. Section 3 & 4 provides our modelling solution based on a Domain-Specific Language (DSL) composed of a Networking Language and a Rule-Based Policy Language, Section 5 introduces our modular Code Generator that interprets a network model to produce deployable network artifacts. Section 6 provides an overview of the existing approaches. Finally, Sections 7 & 9 present the discussions and conclusion.

## 2 RUNNING CASE

Figure 1 depicts a typical Smarthome (SH). This is a common network-based IoT application that will be used throughout the paper to illustrate the utility of our framework. Notice that heterogeneous computing platforms (e.g., C, Java, Python) and communication protocols (e.g., MQTT, UPnP, Zigbee, HTTP, CoAP) are used.

The gateway is running a Python program on top of a Raspberry Pi equipped with a Zigbee chip [35]. The latter establishes a Point to Point (PtP) communication with other Zigbee-ready IoT things. The Temperature Sensor (TS) and the Smart Fridge (SF) are programmed using a standards-compliant C program. The Smart Lock (SL) is running on top of Arduino and the Smart Heating and Ventilation System (SHVS) is programmed using a Java program. All these IoT things are equipped with a Zigbee chip to communicate wirelessly with the gateway.

![Figure 1: Smarthome, usecase of an IoT application](image)

Bob, the owner of the SH, has an Android phone that is running a Java app. Alice, his mother, is an elderly person. She suffers from a Parkinson disease and has to take medications every day on a regular basis.

The gateway also handles dynamic connections using Universal Plug and Play (UPnP). When Bob is at home, his phone communication switches dynamically to connect locally to the gateway using UPnP. An authorized doctor can also control the gateway to access the SH and treat Bob’s mother when Bob is away.

Additionally, the city requires a battery powered Fire and Smoke Sensor (FSS) in all SHs. Given its constrained resources, the FSS uses a standards-compliant C program and an energy-efficient communication on top of Constrained Application Protocol (CoAP). Furthermore, for security reasons, it should not be connected to the local network, so that in case of an outage in the local network, it can still notify the Fire Station (FS). The cityPolicy requires that when the sensed temperature reaches certain threshold, the system should automatically inform the local authorities.

To benefit from the SH manufacturer maintenance service, the gateway should post the telemetry data using Hypertext Transfer Protocol (HTTP), in case an intervention is needed. The manufacturerPolicy requires that in case of a malfunction in the network, the gateway should automatically notify the manufacturer via a dedicated channel. We presume that the gateway holds a property containing the status of the network.

The gateway sends the sensors data to a private Publish and Subscribe (PubSub) messaging middleware accepting Message Queuing Telemetry Transport (MQTT). The gateway translates the Zigbee data into MQTT or HTTP depending on the receiver. Using his phone Bob can then control the SH remotely and monitor the activity of his mother.

---

1. [https://www.eclipse.org/Xtext/](https://www.eclipse.org/Xtext/)
2. [https://github.com/atlanmod/CyprIoT](https://github.com/atlanmod/CyprIoT)
To make our application "smart", we can draw the following simple or conditioned scenarios:

Scenario 1: Bob needs to monitor his mother remotely.
Scenario 2: An outage in the network requires the intervention of the manufacturer.
Scenario 3: The SH is on fire, local authorities need to be notified.
Scenario 4 (conditioned): The smoke sensor notifies the firefighters when the temperature sensor is greater than a given value.
Scenario 5 (conditioned): If an external phone tries to connect to the gateway using UPnP between 22:00 and 09:00, then reject its requests.
Scenario 6 (conditioned): If the fridge contains less than 2 milk packs, then notify the food store to deliver new packs.
Scenario 7 (conditioned): If Bob’s mother forgets to take her medications before 16:00, then email the doctor.
Scenario 8 (conditioned): The food store does not deliver food as usual, even when the fridge is empty, this is because the presence sensor did not detect any activity for a while in the house, nobody is at home to take the delivery.

Finally, to provide meaningful insights for Bob about the activity of its SH, all the sensed data is sent to a web-based analytics platform designed with Javascript.

In fact, a SH, as many network-based IoT applications, consists of 1) a network of heterogeneous things connected using heterogeneous protocols and 2) a set of smart scenarios and policies, defined by some entities, to determine or supervise the behavior of the network.

Further, the SH application along with these scenarios and policies will be modeled within CyprIoT framework. It is worth noting that, in such usecase authentication, privacy and trustworthiness are essential, however they are beyond the scope of this paper.

3 NETWORKING LANGUAGE

In this section we present our first main contribution of the paper, a Networking Language which consists of connecting various IoT things. Figure 2 depicts the metamodel of the proposed abstractions of the language. On another note, to clearly position our work w.r.t the state of the art, we attached the core concepts of the existing work (White) in this area. We also created a link with the Policy Language (Light Gray) that will be the subject of the next section. Moreover, Listings 2 and 3 present the network model of the SH application local communications.

3.1 Thing

We assume that an IoT thing behavior is modeled using ThingML [17]. The latter provides a DSL to model an IoT thing behavior using a statechart. Each state has a transition specifying its next state. In addition, a thing can communicate with the outside world using a port. For instance, as shown in Figure 3, the TS uses a port in the SendTemperature state to disseminate its sensed data.

```
Listing 1: Declaration of things

```// User declaration
1 user Bob
2 user Alice
3
4 // Role declaration
5 role sensor
6 role actuator
7
8 // Import of ThingML models
9 thing homeGateway import 'gateway.thingml' assigned sensor
10 thing temperatureSensor import 'temperature.thingml' assigned sensor
11 thing smartFridge import 'fridge.thingml' assigned sensor, actuator
12 thing smartHeater import 'heater.thingml' assigned actuator
13 thing bobPhone import 'phone.thingml' assigned sensor, actuator
14 thing smokeDetector import 'smokeSensor.thingml' assigned sensor
15 thing medicationBottle import 'medication.thingml' assigned sensor
16 thing lightSwitch import UNKNOWN assigned actuator
17
18 // Import of Arduino code
19 thing smartLock import 'lockmodel.ino' assigned sensor, actuator
```

The Networking Language offers a means of designing a network in a readable fashion using high-level abstractions. Listing 1 depicts the declaration of the things in the SH usecase. For instance, the line `thing temperatureSensor import "temperature.thingml" assigned sensor;` imports the model (i.e ThingML statechart) of the TS.

Notice that the smartLock sensor is sourced from a concrete code (i.e Arduino code), while all other models are sourced from ThingML models. In fact, there are two ways to source the behavior of an IoT thing. The first way consists of loading a ThingML model describing the behavior in the form of a statechart. The second way consists of sourcing the behavior from a concrete source code (e.g., C, JAVA). The latter requires a step further that consists of converting the imported code into a ThingML model (See Section 5.2). That is because our framework needs to work only with ThingML models.

For the sake of interoperability, it is also possible to declare an IoT thing without sourcing it with a behavior, as it is the case for the lightSwitch actuator that is using the keyword UNKNOWN, meaning that the behavior is not provided. If a communication is specified in the network model, ports of other things are configured to communicate with it. For instance, lightSwitch thing uses an unknown program that sends a message to the lightTopic. Thus, if another thing is configured to retrieve the information from the lightSwitch sensor, then it will be configured to subscribe to the lightTopic.

3.2 Channel

The utility of a channel concept arises because of the need to assemble various IoT things without concerns about the concrete details of their communications such as the protocol or the message format. In addition, this also helps easing things collaboration.

We performed a bottom-up analysis on IoT data protocols. Then, we have drawn their commonalities that we reify in the concept of channel. Indeed, it is the medium that exists between two IoT things enabling a communication. We identified two main channel types: channel:pubsub channels and
channel:ptp. Other channels may also be found in some IoT applications, but they are quite little used.

The former type is often used when IoT things have to communicate in a decoupled manner and does not necessarily need to know each other, as it is the case with bobPhone and the gateway that are communicating through MQTT. Those things only need to know the information about the broker that acts as an intermediary between them.

The latter type, in the other hand, is used when the IoT thing is accessible via a public interface, such as an IP address or a Uniform Resource Locator (URL) or visible in a local network, so that another IoT thing can connect to it and get its data. In effect, this is the case for the smartFridge and the gateway, that are communicating through a channel:ptp, namely Zigbee.

As depicted in Figures 2 and 3, ThingML uses ports for external communications. Typically, two IoT things can communicate only if they have compatible ports. In other words, they can understand each other only if they exchange the same type of messages. For instance, creating a bind from the temperatureSensor to the smartLock will not work, as the smartLock is not expected to understand the temperature messages, while connecting it to the smartHeater should work as it has a port that accepts this type of messages.

In our Networking Language, the channel can be typed, so that we can ensure ports compatibility. A warning is fired in the user interface whenever a channel is connecting incompatible ports, that way an engineer can prevent communication incompatibility bugs early in the engineering process. For instance, in Listing 2, presume that the temperatureSensor sends a message of type temperatureMessage to the smartHeater that has a port accepting the temperatureMessage, then to ensure a compatible communication, the intermediary topic needs to be typed with a temperatureMessage (Line 25 in Listing 2).
3.3 Network

After declaring things and channels, a network can then be configured using the keyword `network`. Listing 3 shows the configuration of the SH network. The network has a domain that is unique and serves as a global identifier for the network. For instance, in our running case we use the domain in the topic structure as the root topic of the channel. The network also contains the instantiation of IoT things and channels as well as their bindings. The platform running an instance can also be declared. For instance, as stated in Line 7, the gateway thing is running on top of PYTHON, while the private broker channel, in Line 15, is running on top of MQTT. A thing instance can have an owner, that is the user holding all the privileges over the thing. This may be used by the Policy Language for control purposes (See Section 4). A `bind` declaration connects a thing’s port to a PubSub or PTP channel, respectively through a `topic` or a `connectionPoint`.

In many network-based IoT applications, data have to pass through an intermediary IoT thing before reaching its final destination. This is the case with the gateway, that forwards the sensed data to the private broker to be received by Bob via the channel `pubsub`. For this specific case, we propose the concept of bridge. It can forward an existing communication (i.e bind) to a topic or a connectionPoint. Line 42-46 create a bridge for each of the sensed data that Bob should receive. On another note, it is also possible to enforce a bridge as a rule. This is discussed in Section 4.5.

4 POLICY LANGUAGE

The second main contribution is a Policy Language which enables to control the modeled network. Figure 4 depicts the metamodel of the proposed abstractions. This section describes how a policy can be expressed. In Section 5 we will treat how it can be enforced.

4.1 Policy

A policy contains a set of rules that can be enforced by the Code Generator (CG). We presume that policies are similar to contracts. They ensure that the IoT application is behaving as expected from the perspective of a given entity such as the government, the SH owner or the manufacturer. Within
our DSL, we presume that the policy can access the declared IoT thing’s internal behavior, modeled a priori.

Moreover, we permit the enforcement of many policies in the same network. For instance, in our running case, as shown in Listing 3 we enforce a cityPolicy, a homePolicy and a manufacturerPolicy. As of today, rule conflict management has not been investigated, although it is contemplated in our future work.

4.2 Rule Structure
Listing 4 depicts the syntax of a rule. It is composed of 5 elements - Subject, Effect, Action, Object and Condition(s). The latter is optional. Table 1 depicts different attributes of those elements and how they can be combined with each other.

```
1 rule <Subject> <Effect>:<Action> <Object> when <Condition>
```

Listing 4: Rule syntax

4.3 Trigger
To implement some of the SH scenarios, we may need to trigger some actions under some conditions. This if often referred as the “smart” dimension of the IoT. Our Policy Language allows defining such rules.

A trigger rule can activate two actions Transition that performs the transition of a given state on a thing and GoToState that instruct the thing to go to a specific state. Those actions can be conditioned by the CurrentState of a subject thing, i.e. to be able to activate an action when CurrentState of a subject thing is of a given value, or by NextState, i.e. to be able to activate the action if its next state is of a given value.

For instance, in Line 15 of Listing 5 the medicationBottle triggers GoToState in gateway to email the doctor when the medication has not been taken. We presume that the email state in the gateway is a routine handling sending the email.

4.4 Permission
By default, we presume that unless a rule is allowing IoT things to communicate, all the communications are denied. A permission allow/deny a communication between two entities. When the subject is a user, it applies to all the things owned by this user. While when the subject is a role, it applies to all things being assigned this role.

In the SH application, we apply a basic role-based policy that allows sensors to send only, and actuators to receive only, as these are the normal communication actions they are supposed to achieve. Also, in Line 16 we deny communicating any information to the httpChannel channel, i.e the food store, typically for asking for a new milk delivery, when the presenceSensor does not detect any activity for more than 1 day.

4.5 Bridge
A bridge rule enforces a behavior similar to the one described previously (See Section 3.3). However, it is not intended to forward an existing communication in the network. Rather, it applies a forward globally on types such as a topic or a connectionPoint. Then, if those types are used within the network, the bridge will be enforced. For instance, a bridge rule can specify that any information received by a given thing or topic, has to be received by a thing, or that any information received by a thing to be received by another thing too.

4.6 Control Types
The rules permit to apply two types of control:

- **Communication Control**: This consists of a Deny/Allow of sending or receiving messages between two entities. The message content may be used for dynamic control, i.e controlling the message flow while the network application is running. For instance, we may deny sending a message when it takes certain value, in a PubSub communication this is known as content-based PubSub [28].
• Thing Behavior Control As stated earlier, the behavior of a thing is modeled using a statechart. In that respect, those rules aim at controlling the behavior specified in this statechart. For instance, a rule can trigger that a given thing’s CurrentState is B.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Effect</th>
<th>Action</th>
<th>Object</th>
<th>Conditions</th>
<th>Control Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thing Instance</td>
<td>Trigger</td>
<td>Transition GoToState</td>
<td>Thing</td>
<td>CurrentState NextState</td>
<td>Thing Behavior</td>
</tr>
<tr>
<td>Thing Instance</td>
<td>Allow</td>
<td>Send</td>
<td>Domain</td>
<td>Thing</td>
<td>Role User</td>
</tr>
<tr>
<td>Thing Instance</td>
<td>Bridge</td>
<td>From</td>
<td>To</td>
<td>Thing Instance</td>
<td>Topic</td>
</tr>
</tbody>
</table>

### Listing 5: Definition of policies

```java
policy roleBasedPolicy { // Role-Based policy
  rule sensor allow.send channel gateway.Channel, broker
  rule actuator allow.receive channel gateway.Channel, broker
}
policy cityPolicy { // Scenarios 3 & 4 (In order)
  rule temperatureSensor trigger goToState
    smokeDetector.notifyState when
    property: currentTemperature > 25
}
policy homePolicy { // Scenarios 5, 6, 7, 8 (In order)
  rule homeGateway.UpnpPort deny.send bobPhone when
    property: (bobPhone.id == bobIDXXXX || bobPhone.id != aliceIDXXXX) and
    property: (homeGateway.currentHour >= 22 && homeGateway.currentHour < 9)
  rule homeGateway.trigger goToState homeGateway,
    notifyMilkState when homeGateway.milkPack.size < 2
  rule medicationBottle.trigger goToState homeGateway.
    emailState when medicationBottle.medicationTaken == false and
    property: homeGateway.currentHour > 16
  rule homeGateway.askMilk deny send channel: http when
    message: homeGateway.milkPack.size < 2 and
    messages: homeGateway.presenceSensor == false and
    property: homeGateway.presenceDay == 1
}
policy manufacturerPolicy { // Scenario 2
  rule homeGateway.trigger goToState homeGateway.
    notifyState when
    property: isNetworkWorking == false
}
```

### 5 CODE GENERATION

Our last main contribution is a CG along with a plug-in system to leverage the network model designed with the DSL. It consists of generating deployable code implementing the network configuration as well as the enforced policies. In this section, we explain its main building blocks, how the network artifacts are generated and how it can be extended with expert knowledge.

5.1 Architecture

Figure 5 depicts the main building blocks of CyprIoT framework. Our proposed DSL, composed of the Networking and Policy Languages, serves to express with high-level abstractions the model of a network-based IoT application. This model is then interpreted by the CG that is responsible for producing deployable network artifacts.

The core of the CG (CG-Core) is composed of three modules: Network Generator (CG-NG), Plug-in Loader (CG-PL) and a Command Line Interface (CG-CLI).

The CG-NG processes the network model following a sequence of phases executed in order. Those phases are executed according to this specific order: Initialize, Load, Validate, Transform, Generate. Each phase has a responsibility that is depicted in Table 2. The CG-NG module renders transformed ThingML models implementing what was specified in the network model. Then, using ThingML multi-platform code generator as a library, deployable code can be generated for any platform (e.g., Java, C). As of today, we concentrated our efforts into the five former phases, in the future we plan to add two more phases: Verify to test and certify the conformity of the generated artifacts w.r.t the network model and Deploy, to automatically deploy the generated artifacts into a running network.

The CG-PL is responsible of loading plug-ins that are hooked into CG-Core. In order to be recognized, a plug-in needs to be declared in a configuration file. This module is discussed in Section 5.3.

The CG-CLI offers a means to use our CG as a standalone application. It expects a declaration of the input network model along with some optional arguments. Then, based on those elements, the CG is executed to produce the IoT network artifacts.

This design choice (i.e Modular Design) is motivated by the need to ease extensibility an separation of concerns. Thus, our CG does not implement expert knowledge but offer mechanisms to implement it by experts using plug-ins.

5.2 Model Loading

Our CG uses ThingML 3, which is also an open source tool, to load and process the behavior of a thing.

---

3https://github.com/TelluIoT/ThingML
Table 2: Code generator extension points

<table>
<thead>
<tr>
<th>Interface</th>
<th>Input</th>
<th>Output</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Void</td>
<td>Void</td>
<td>Initialize the application and the plugins</td>
</tr>
<tr>
<td>Load</td>
<td>Network Model</td>
<td>ThingML Models</td>
<td>Source a thing with a behavior, then convert into a ThingML model if necessary</td>
</tr>
<tr>
<td>Validate</td>
<td>ThingML Models</td>
<td>Boolean</td>
<td>Validate the conformity of the ThingML models w.r.t some requirements</td>
</tr>
<tr>
<td>Transform</td>
<td>Network Model</td>
<td>ThingML Models</td>
<td>Transform the ThingML models to conform with the network model</td>
</tr>
<tr>
<td>Generate</td>
<td>ThingML Models</td>
<td>Network artifacts</td>
<td>Generate network artifacts</td>
</tr>
</tbody>
</table>

It is possible to load a behavior from a concrete code, as long as its can be transformed into a ThingML format. A plug-in may be needed for that purpose.

As a proof-of-concept, we developed a simple plug-in that we hook to the Load phase. It loads an Arduino file, finds its external communication interfaces (e.g., MQTT Publish/Subscribe commands), then renders a ThingML file encapsulating the behavior intended in the concrete code, and abstracting its communication interfaces into ports so that they can be bound to any channel within the Networking Language, as it is the case for a model-based thing behavior. As of now, we can only identify the interfaces written in a certain format. This feature may enable interoperability with traditional approaches and will be extended in future work.

5.3 Plug-in System

Table 2 shows the available interfaces for plug-ins. Each interface corresponds to a phase, it accomplishes a specific task, and impose a specific input and output. A typical network-based IoT application may involve several expertise (e.g., Safety, Access Control, Data Consistency), the plug-in system aims at providing experts a way to create a plug-in implementing their concerns in the network artifacts. For instance, as a proof-of-concept, we implemented a simple plug-in to generate access control rules for a Mosquitto broker. We hook the plug-in to the Generate phase that provides us with the network model as well as the things’ internal behavior in a ThingML format, we leverage those inputs to produce a consistent access rules to be enforced in the broker.

6 RELATED WORK

ThingML [17] proposes a methodology for the IoT using established MDE techniques [24]. The language has shown its efficiency at abstracting hardware and programming languages [23, 32]. The approach provides a DSL to design the IoT things’ internal behavior using statecharts and an extendible multi-platform code generation framework. The latter also provides a plug-in system to add a network client to IoT things. However, abstractions w.r.t communication are rather minimal in the DSL, simply consisting in declaring the used protocol and its attributes. In other words, the proposed language does not offer abstractions capturing network aspects such the communication channel that can be used between IoT things.

Salihbegovic et. al [26] present a Visual Domain-Specific modelling Language (VDSML) based on a JavaScript editor. It aims at giving an IoT engineer a user interface to virtually design an IoT system. Only a set of predefined IoT things are available to use within the editor. The tool is able to generate a configuration file for IoT platforms, namely OpenHab [21]. However, the formal specification of the language such as the metamodel is missing. The language is not enough generic as only a limited set of IoT things can be modeled. By using a statechart-based solution in our framework, we can theoretically model any IoT thing behavior.

Node-red [19] is a flow-based visual tool that aims to connect various interfaces of IoT things. The tool focuses only on the connection of already-deployed IoT nodes. Its basic idea is to map the output of an IoT thing to the input of another, this mapping is made inside the platform running Node-red. Such method is useful only to create a “mashup” [5, 18, 20] of existing services. The approach does not provide a model of the network. Controlling the behavior of an IoT thing as well as code generation are not provided. While our framework not only provides a model of the running network, but it is also capable of controlling the network using a Policy Language.

Fuch et. al [12] propose to program an IoT thing using a UML2 Activity Diagram (UAD). The behavior of an IoT thing is designed in the form of an activity. The latter is transformed into a script in order to be executed by an interpreter running on the IoT thing. Activities can communicate between each with their input/output interfaces via a prototypical communication protocol. However, the approach focuses on the advantages of UADs to ease collaboration between IoT things and does not discuss the heterogeneity of their communications. Moreover, only one IoT thing platform (SUN Spot [29]) has been considered in their study.

Amrani et. al [1] introduce a DSL to design a network of IoT things. The language allows to declare the possible actions of an IoT thing. Those actions need to be mapped to concrete events in the target platform. The DSL is accompanied by a rule-based policy language to trigger actions when certain conditions are met. The communication between IoT things is not conceptualized in the DSL metamodel, it consists of specifying a connection of an IoT thing to a specific protocol, which creates a hard coupling between the abstract representation and the concrete representation. In our DSL, we aim at bypassing communication heterogeneity by abstracting commonalities of prominent IoT protocols in the concept of channel. Moreover, code generation is not discussed in their work.

Bertran et. al [4] present a tool based on the Sense/Compute/Control (SCC) paradigm [8]. It consists of a DSL, a generator of Java code, a simulator and a deployment framework. Although, the DSL abstracts the specification of an IoT thing, we still need to implement its behavior in Java after code generation. The framework assumes that IoT things are
Abstracting the heterogeneity of the communication channels

According to our empiric experimentations, the approach has
 Policy Language.
 control over the communication by leveraging the rule-based
 enables better collaboration of IoT things and offers better
 level and hard skills to build a network-based IoT application.
 a plug-in to enhance the security of the network.
 easily added. For instance a security layer may be added as
 result of an automatic process.
 network is less error-prone and more robust as they are the
 using measuring instruments. We remark that the generated
 network of IoT things, however we still need to evaluate this
 proven that we may need less time to generate a working

7 DISCUSSIONS

According to our empiric experimentations, the approach has
 proven that we may need less time to generate a working
 network of IoT things, however we still need to evaluate this
 using measuring instruments. We remark that the generated
 network is less error-prone and more robust as they are the
 result of an automatic process.

The plug-in system eases the extensibility of the framework,
 features that are not provided by the core of our CG can be
 easily added. For instance a security layer may be added as
 a plug-in to enhance the security of the network.

Using CyprIoT, an IoT engineer needs to learn less low-
level and hard skills to build a network-based IoT application.
 Abstracting the heterogeneity of the communication channels
 enables better collaboration of IoT things and offers better
 control over the communication by leveraging the rule-based
 Policy Language.

<table>
<thead>
<tr>
<th>Table 3: Comparison of existing approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>[17]</td>
</tr>
<tr>
<td>[26]</td>
</tr>
<tr>
<td>[19]</td>
</tr>
<tr>
<td>[12]</td>
</tr>
<tr>
<td>[11]</td>
</tr>
<tr>
<td>CyprIoT</td>
</tr>
</tbody>
</table>

N/A : Not Applicable — N/P : Not Provided — (*) Limited

Besides the usecase presented in this paper. Our approach
may have other applications, for instance, it could be lever-
aged for security in collaborative systems such as implement-
ing an advanced access control [30] within the proposed Policy
Language.

Some limitations may disprove our approach. In some
specific cases we still need to look at the low-level details to
understand how we can establish a reliable communication
between two IoT things. We tested our approach only for
few communication channels. This means that our approach
is still not systematic. Also, only ThingML models were
considered as source models, this was the most advanced
DSL we could find to model a thing’s internal generically.
Moreover, as of today, IoT things sourced from a concrete
code may still need to have their interfaces written in a
specific format in order to be used in the network model,
which limits the possibilities of bindings.

8 CONCLUSION

An IoT network application involves heterogeneous comput-
ing platforms and communication protocols. Commonly, each
protocol is designed to fit a specific range of IoT things. We
conducted this study to find means to connect heterogeneous
IoT things as well as control mechanisms to design smart
and realistic IoT applications in a unified manner. Thus,
exempting IoT engineers from learning transversal skills and
focus only on the business logic of their network-based IoT
application.

MDE is a promising paradigm to tackle the ubiquitous
heterogeneity in the IoT. In this paper, we showed that
by abstracting the common concepts of similar technologies
and separating the engineering knowledge from the technical
knowledge, we exempt the developer from the need to look
at the low-level details, that are time-consuming and provide
less value compared with the logic of the whole network-based
IoT application.
CyprIoT, the framework introduced in this paper, provides MDE instruments to develop network-based IoT applications. It consists of a readable Networking Language to enable modelling the IoT application globally, a Rule-Based Policy Language to control the modeled application and a modular and extensible Code Generator to generate deployable network artifacts. The code generation process is divided into phases. Each phase has a specific responsibility to ease extensibility and separate concerns. In addition, a plug-in system is conceptualized to allow experts to implement their knowledge in the generated artifacts.

In future work, we aim to make the DSL more readable, expressive, and interoperable. We will also work on the verification of the generated artifacts and we envision easing their deployment. Finally, we will continue improving the overall architecture of the framework with better modularity.

9 ACKNOWLEDGEMENTS

We acknowledge the support of Institut Mines-Télécom Atlantique and the Natural Sciences and Engineering Research Council of Canada (NSERC), 06351.

This research has been financed by the Institut Mines-Télécom Atlantique and the Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), 06351.

REFERENCES

[33] Woolf, N.: Ddos attack that disrupted internet was largest of its kind in history; experts say. The Guardian 26 (2016)