A model-driven methodology to unify software engineering in the internet of things

Imad Berrouyne

Thèse de doctorat de

L'ÉCOLE NATIONALE SUPÉRIEURE
MINES-TÉLÉCOM ATLANTIQUE BRETAGNE
PAYS-DE-LA-LOIRE - IMT ATLANTIQUE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : *Informatique*

Par
Imad BERROUYNE

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

Thèse présentée et soutenue à Saguenay, le jeudi 4 février 2021
Thèse N° : 2021IMTA0233
Unité de recherche : IMT Atlantique Bretagne-Pays de la Loire, Ecole Mines-Télécom
Laboratoire des Sciences du Numérique de Nantes – LS2N (CNRS, UMR 6004)

Rapporteurs avant soutenance :

Yann-Gaël GUÉHÉNEUC	Professeur – Université Concordia, Montréal (Canada)
Davide DI RUSCIO	Maître de conférences (HDR) – Université de L'Aquila, L'Aquila (Italie)

Composition du Jury :

Président :	Yann-Gaël GUÉHÉNEUC	Professeur – Université Concordia, Montréal (Canada)
Examinateurs :	Abdenour BOUZOUANE	Professeur – Université du Québec à Chicoutimi, Saguenay (Canada)
	Jean-Claude ROYER	Professeur – IMT Atlantique, Nantes (France)
	Mehdi ADDA	Professeur – Université du Québec à Rimouski, Rimouski (Canada)
	Luigi LOGRIPPO	Professeur – Université du Québec en Outaouais, Gatineau (Canada)
	Davide DI RUSCIO	Maître de conférences (HDR) – Université de L'Aquila, L'Aquila (Italie)
Directeur de thèse :	Jean-Claude ROYER	Professeur – IMT Atlantique, Nantes (France)
Co-encadrant de thèse :	Massimo TISI	Maître assistant – IMT Atlantique, Nantes (France)
Co-encadrant de thèse :	Jean-Marie MOTTU	Maitre de conférences – Université de Nantes, Nantes (France)

Abstract

The Internet of Things (IoT) aims for connecting Anything, Anywhere, Anytime (AAA). This assumption brings about a good deal of software engineering challenges. These challenges constitute a serious obstacle to its wider adoption. The main feature of the Internet of Things (IoT) is genericity, i.e., enabling things to connect seamlessly regardless of their technologies.

Model-Driven Engineering (MDE) is a paradigm that advocates using models to address software engineering problems. MDE could help to meet the genericity of the IoT from a software engineering perspective. In that sense, the IoT could be a requirement provider on the one hand and MDE its solution provider on the other. Existing MDE approaches focus on modeling the behavior of things. But, little attention has been paid to network-related modeling.

The present thesis presents a methodology to create smart networks of things based on MDE. It aims to cover and leverage the network-related aspects of an IoT application compared to the existing work. The principle we use consists of avoiding the intrinsic heterogeneity of the IoT by separating the specification of the network, i.e., the things, the communication scheme, and the constraints, from their concrete implementation, i.e., the low-level artifacts (e.g., source code). Technically, the methodology relies on a model-based Domain-Specific Language (DSL) and a code generator. The former enables the specification of the network, and the latter provides a procedure to generate the low-level artifacts from this specification. The adoption of this methodology permits making software engineering of IoT applications more deterministic and saving a significant amount of lines of code compared to the state of practice.

Keywords : Internet of Things, Software Engineering, Model-Driven Engineering, Model Transformation, Policy Enforcement, Code Generation

RÉSUMÉ

L'Internet des objets (IdO) vise à connecter tout objet, partout, en tout temps (TTT). Cette hypothèse entraîne de nombreux défis en matière de génie logiciel. Ces défis constituent un sérieux obstacle à son adoption à grande échelle. L'une des principales caractéristiques de l'IdO est la généricité, c'est-à-dire permettre aux objets de se connecter de manière transparente, quelles que soient la technologie qu'ils utilisent.

L'Ingénierie Dirigée par les Modèles (IDM) est un paradigme qui préconise l'utilisation de modèles pour résoudre les problèmes de génie logiciel. L'IDM pourrait aider à répondre au besoin de généricité de l'IdO du point de vue du génie logiciel. Les approches d'IDM existantes se focalisent essentiellement sur la modélisation du comportement des objets. Peu d'attention a été accordée à la modélisation liée à leur réseautage.

La présente thèse présente une méthodologie pour l'IdO basée sur l'IDM. Fondamentalement, elle fournit une solution pour créer des réseaux intelligents d'objets. Le principe que nous utilisons consiste à contourner l'hétérogénéité intrinsèque de l'IdO en séparant la spécification du réseau, c'est-à-dire les objets, le schéma de communication et les contraintes, de son implémentation concrète, c'est-à-dire les artefacts logiciels de bas niveau (par exemple, le code source). Techniquement, la méthodologie repose sur un langage dédié basé sur les modèles pour la spécification du réseau et une procédure pour la génération du code des artefacts de bas niveau à partir de cette spécification. L'adoption de cette méthodologie rend l'ingénierie logicielle des applications d'IdO plus rigoureuse, permet de prévenir les bogues plus tôt et de gagner du temps.

Mots clés : Internet des objets, Génie logiciel, Ingénierie dirigée par les modèles, Transformation des modèles, Politiques de contrôle, Génération de code

To Mina, for her incredible patience and determination.
This manuscript is her diploma.

To Mustapha, for the outstanding conditions.

This work is entirely dedicated to them.

Life has this strange thing of being brutal
but endearing...

Acknowledgements

I want to express my gratitude to my supervisors for their exceptional support and devotion. I thank Jean-Claude Royer, who directed my thesis on the IMT Atlantique side and offered a deep insight into the study, Abdenour Bouzouane, who led my thesis on the UQAC side and provided valuable guidance. My thanks equally go to Massimo Tisi, who co-directed this work and whose friendly advice has greatly enlightened my understanding of model transformation, Mehdi Adda, who co-directed this work and whose insightful and convivial discussions have substantially contributed to my knowledge on control and policies enforcement and Jean-Marie Mottu who co-directed my work and constantly ensured that my thesis was progressing smoothly.

I wish to acknowledge the insightful recommendations provided by Soichiro Hidaka and Noël Plouzeau as members of the monitoring committee. I want to thank my colleagues in the NaoMod team and the jury members, Yann-Gaël Guéhéneuc, Davide Di Ruscio, and Luigi Logrippo, for their precious feedback and advice.

I would like to extend my special thanks to Laboratoire des Sciences du Numérique de Nantes (LS2N), IMT Atlantique, Université du Québec à Chicoutimi (UQAC), Université du Québec à Rimouski (UQAR), Université du Québec en Outaouais (UQO) and the National Research Council Canada (CNRC) for providing us with the proper environment to succeed and for their financial support. The assistance supplied by Delphine Turlier (IMT Atlantique) and Anie Tremblay (UQAC) for accomplishing the administrative duties was greatly appreciated.

My warmest thanks go to my parents, my sisters, and my brother for their unconditional support during this journey.

Table of Contents

Abstract **3**

Résumé **5**

Dedication **7**

Acknowledgements **11**

List of Tables **17**

List of Figures **19**

List of Abbreviations **21**

Synthèse en Français **25**

Introduction **33**

1 State of the Art & Context **37**

1.1 From Network of Computers to Network of Things 37

1.1.1 The Conventional Internet 37

1.1.2 Towards the IoT 38

1.1.3 Software Engineering for the IoT 39

1.2 Model-Driven Engineering 42

1.2.1 Modeling 42

1.2.2 Model Transformation 43

1.2.3 MDE and IoT 47

1.3 Summary 54

2 Modeling a Network of Things **59**

2.1 Reification of IoT concepts 59

2.1.1 Things 61

2.1.2 Channels 65
2.1.3 Users and Roles 67
2.1.4 Network 68
2.1.5 Forwarding 69
2.2 Every “thing” is a model 71
2.2.1 Metamodel 71
2.2.2 Network Model 73
2.2.3 Usability of the Model 74
2.3 Domain-Specific Language 74
2.3.1 Integrated Development Environment 75
2.3.2 Readability and Maintainability 75
2.4 Summary 76

3 Controlling a Network of Things 79
3.1 Model-based Control 79
3.1.1 Policy 79
3.1.2 Rule 81
3.1.3 Control Types 82
3.2 Communication Control Rules 83
3.2.1 Structure 84
3.2.2 Potential Applications 84
3.2.3 Ports Communication Control 85
3.2.4 Things Communication Control 85
3.2.5 Users Communication Control 87
3.2.6 Communication Control Based on Roles 87
3.2.7 Combinations 88
3.3 Smart Rules 89
3.3.1 Structure 90
3.3.2 Potential Applications 90
3.3.3 Behavioral factors 91
3.3.4 Temporal factors 91
3.4 Conflict detection and resolution 92
3.4.1 Early Detection 93
3.4.2 Conflict Detection Algorithms 93

3.4.3 Resolution Strategies at Enforcement 96

3.5 Summary 97

4 Generation of the Network Artifacts 99

4.1 Code Generator 99

4.1.1 Core Architecture 100

4.1.2 Model Loading 100

4.1.3 Extensibility 101

4.2 Model Transformation 102

4.2.1 Model-to-Model Transformation 103

4.2.2 Model-to-Text Transformation 106

4.3 Enforcement Strategies 107

4.3.1 Enforcement of Communication Control Rules 107

4.3.2 Enforcement of Smart Rules 112

4.4 Summary 114

5 Assessment 115

5.1 Methodology 115

5.2 Evaluation 117

5.2.1 Quantitative Evaluation 117

5.2.2 Qualitative Evaluation 119

5.3 Case Study: Smart home 120

5.3.1 Description 121

5.3.2 Implementation 122

5.4 Potential Applications 123

5.4.1 Smart City 123

5.4.2 Industry 4.0 124

5.5 Discussion 124

5.5.1 Research Considerations 124

5.5.2 Threats to validity 126

6 Conclusion & Perspectives 133

6.1 Contributions of the thesis 134

6.1.1 Contribution 1: Software Engineering Methodology 134

6.1.2 Contribution 2: Model-Based Network Abstractions 135

6.1.3 Contribution 3: Model-Based Control Abstractions 135
6.1.4 Contribution 4: Code Generator 135
6.2 Perspectives 135
6.2.1 Model-Driven Reverse Engineering 136
6.2.2 Better Artifacts Generation 136
6.2.3 Simulation 137

Bibliography **139**

Appendices **155**

A Xtext Grammar for the Networking Language **155**

B Xtext Grammar for the Policy Language **159**

C ATL Rules for Networking and Forwarding Transformations **162**

D ATL Rules for Communication Control Transformations **173**

E ATL Rules for Smart Rules Transformations **186**

List of Tables

1.1 Summary of the main existing MDE approaches for the IoT 56

3.1 The combination of rule entities. 82

4.1 Code generator plugin interfaces 101

5.1 Comparison of the required lines of code with CY-DSL, C, Java
ăand Arduino 118

List of Figures

1.1 The four-layers architecture of MDA . . . 43
1.2 The two types of model transformation . . . 44
1.3 The four-layers architecture of MDA applied to IoT . . . 48

2.1 The mapping of the information theory model concepts to the IoT concepts 60
2.2 A simplified version of the ThingML metamodel . . . 62
2.3 Statechart-based behavior of a temperature sensor . . . 63
2.4 A graph-based example of a network of things . . . 66
2.5 Networking language metamodel . . . 72
2.6 An auto-completion example to suggest the possible paths . . . 75
2.7 The early detection of inconsistencies in the editor . . . 76
2.8 Modeling the network: the first step of the methodology . . . 77

3.1 Policy language metamodel . . . 80
3.2 A conflict detection error in the editor of opposite rules . . . 94
3.3 A conflict detection error in the editor between coarse and fine grained rules 94
3.4 Controlling the network: the second step of the methodology . . . 97

4.1 The core components of the code generator . . . 99
4.2 Generation of network artifacts using the T-PROCESS . . . 103
4.3 GTRs for adding the communication interface and forwarding . . . 104
4.4 Generation of access control rules for Mosquitto and RabbitMQ . . . 107
4.5 Enforcement checkpoints . . . 108
4.6 Enforcement at the “on receive” checkpoint . . . 109
4.7 Enforcement at the “on send” checkpoint . . . 109
4.8 Enforcement of a user-based rule . . . 110
4.9 Enforcement of a role-based rule . . . 110
4.10 Enforcement of a Best-Effort strategy . . . 111
4.11 GTR for applying the trigger:executeFunction rule . . . 112
4.12 GTR for applying the trigger:goToState rule . . . 113

4.13 Generation of the network artifacts: the last step of the methodology . . . 114

5.1 BPMN diagram of the methodology 116
5.2 CY-CGEN execution time according to the number of things 119
5.3 A smart home example 120

List of Abbreviations

EMF	Eclipse Modeling Framework (EMF) is an established framework developed and maintained by Eclipse to create metamodels. Multiple model-based tools rely on this framework as a basis. Xtext uses EMF to define the abstract syntax of a DSL. As ThingML-DSL and CY-DSL are built using Xtext, they are both relying on EMF. The role of EMF in the methodology is to offer a common modeling basis for our model-based tools.
TH-DSL	ThingML-DSL (TH-DSL) is the domain-specific language introduced by Harrand et. al [90] within the ThingML toolset. TH-DSL offers a means to specify the behavior of a thing in the form of a statechart. This specification is an EMF model. The role of TH-DSL in the methodology is to enable writing the behavior of a thing.
TH-Model	ThingML-Model (TH-Model) is an EMF model encapsulating the behavior of a thing deemed to be written using TH-DSL. By having an EMF model, we are able to reuse the existing EMF-based solutions (e.g., ATL, Acceleo) in our methodology.
TH-CGEN	ThingML-Code Generator (TH-CGEN) is the code generator introduced by Harrand et. al [90] to generate the low-level code from an EMF model encapsulating a statechart-based behavior of a thing. The role of TH-CGEN in the methodology is to generate the low-level code from the TH-Model.
CY-DSL	CyprIoT-DSL (CY-DSL) is the domain-specific language we introduce within the CyprIoT toolset. CY-DSL offers a means to specify a network of things and optionally to specify a control policy enforced in the network. This specification is an EMF model. The role of CY-DSL in the methodology is to enable the wiring of the things specified using TH-DSL.
CY-Model	CyprIoT-Model (CY-Model) is an EMF model encapsulating the specification of the network and the policies deemed to be written using CY-DSL. By having an EMF model, we can reuse the existing EMF-based solutions

	(e.g., TH-Model, ATL, Acceleo) in our methodology.
CY-GEN	CyprIoT-Code Generator (CY-GEN) is the code generator we introduce within the CyprIoT toolset. CY-GEN takes as input the model specified in CY-DSL, then generates the network of things in the low-level code and any textual artifact derived from this input model (e.g., documentation, configuration file).
Mod-Load	Model Loading (Mod-Load) is the procedure we propose in CY-GEN that is responsible for loading the models (i.e., CY-Models and TH-Models) required by the transformation process.
M2MT	M2MT (Model-to-Model Transformation) is a type of model transformation aiming to produce a target model artifact from the input model. For instance, this mechanism allows us to adapt the model of a thing according to some properties (e.g., topology) of the network model.
M2TT	M2TT (Model-to-Text Transformation) is a type of model transformation aiming to produce a target text artifact from the input model. For instance, this mechanism allows us to generate some textual artifacts that are not part of a thing's internal behavior (e.g., access control rules, configuration file, documentation). In general, the information necessary to make these artifacts exists in the network model, yet it needs to be written in the syntax of these artifacts.
T-PROCESS	The Transformation Process (T-PROCESS) is the transformation work that occurs within CY-GEN. It includes both model-to-model transformations and model-to-text transformations. In this process, we implemented the required code to transform TH-models according to the CY-Model specification and the code necessary to generate the textual artifacts.
ATL	The ATLAS Transformation Language (ATL) is a model-to-model transformation language based on EMF, i.e., it takes an EMF input model and produces an EMF output model. ATL allows us to adapt the model of a thing written in TH-DSL according to some properties (e.g., topology) of the network model written in CY-DSL.
GTR	The Graph Transformation Rule (GTR) is a graphical formalism that we use for readability purposes to present the ATL rules as the reader may be unfamiliar with ATL syntax. Still, the ATL rules are provided in the

	appendices. The GTR consists of two sides, the Left-Hand Side (LHS) and the Right-Hand Side (RHS). The LHS presents the model before it undergoes the transformation or the input model, while the RHS presents the model after transformation. The GTR has no direct role in the methodology; instead, it makes ATL rules more presentable.
MQTT	MQTT (Message Queuing Telemetry Transport) is open OASIS and ISO standard (ISO/IEC 20922) publish and subscribe protocol used generally for a decoupled communication between resource-constrained devices. This protocol is widely used in the IoT. A publish and subscribe communication consists of a sender, a receiver, and a broker. The sender and the receiver communicate with the broker. The broker serves as an intermediate between the things. The advantage of having a broker is that it can store the message published by a thing in a topic and send it to the things that subscribed to this topic in a decoupled manner. We rely on MQTT in our methodology as a communication protocol between things.
Mosquitto	Mosquitto is an open-source MQTT broker that is largely used in the IoT. Mosquitto requires some access control rules to ensure secure access to its topics. Within our methodology, we generate these access control rules automatically.
RabbitMQ	RabbitMQ is another open-source MQTT broker that is largely used in the IoT. In the same way as Mosquitto, it also requires some access control rules to ensure secure access to its topics. We automatically generate its access control rules within our methodology to emphasize its ability to generate multiple textual artifacts from a unique model.

Synthèse en Français

Introduction

L'Internet des Objets (IdO) est un paradigme qui vise à connecter Tout objet, parTout, en Tout temps (TTT) [15]. En particulier, ce paradigme permet de connecter des objets de tailles diverses allant des bactéries [16] aux superordinateurs. Manifestement, chaque objet a ses propres exigences. Par exemple, un capteur de petite taille peut nécessiter un client utilisant le Protocole pour Applications Contraintes (CoAP) pour se connecter au web, en raison de ses ressources limitées, alors qu'un ordinateur portable nécessite un client utilisant le Protocole de Transfert Hypertexte (HTTP). Les prémisses de l'IdO sont que (1) tout ce qui a une puissance de calcul (2) peut se connecter à l'Internet. Alors que la première prémisse (1) est liée au matériel, la deuxième (2) pourrait être abordée en utilisant une approche appropriée de génie logiciel.

L'ingénierie logicielle pour l'IdO est complexe. D'une part, de nombreuses parties prenantes sont impliquées (par exemple, des responsables de sécurité, du réseau ou du commerce), chacune utilisant ses propres outils et méthodes [19]. D'autre part, l'hétérogénéité (par exemple, des langages de programmation et des protocoles) [20] cause, entre autres, un problème d'interopérabilité en entravant la communication entre les objets. En général, une application d'IdO typique contient plusieurs plateformes, langages et protocoles. De plus, chaque jour, un nouvel objet connecté émerge, avec des fonctionnalités souvent non standards.

Néanmoins, bien qu'elle soit problématique, l'hétérogénéité constitue le facteur différenciateur entre l'IdO et l'Internet classique. En effet, de nombreuses études [21, 22, 23, 24, 25] suggèrent qu'il est nécessaire de connecter des objets de gammes différentes au moyen de différents protocoles.

Peu d'approches de génie logiciel ont été proposées pour répondre aux exigences de l'IdO. Aujourdhui, les approches existantes sont souvent chronophages, exigent beaucoup d'expertise et conduisent à des applications d'IdO insuffisamment testées et peu sûres [18, 27]. En effet, l'hétérogénéité exige d'impliquer plus de ressources humaines et d'expertise que d'habitude. C'est pourquoi, la plupart des entreprises, souvent dotées de ressources humaines inappropriées ou limitées, ne répondent pas à ces exigences de manière adéquate. Ainsi, il peut en résulter des applications défectueuses qui peuvent être à l'origine d'attaques à grande échelle telles que Mirai et Persirai [28, 29].

En fait, pour les experts en sécurité, les approches de génie logiciel existantes ont montré leurs limites en matière de sécurité des réseaux. L'institut SANS rapporte que près de 90 % des professionnels de la sécurité affirment que des changements aux contrôles de sécurité sont nécessaires dans le domaine de la sécurité pour l'IdO [30].

Il est clair que l'IdO a besoin d'une nouvelle approche de génie logiciel. L'Ingénierie Dirigée par les Modèles (IDM) est un paradigme qui a le potentiel de surmonter certaines des problématiques de l'IdO. Il peut permettre de concevoir des applications d'IdO robustes et fiables en séparant la spécification (source d'intention) de l'implémentation (source d'hétérogénéité). En particulier, en permettant la conception d'une application d'IdO complète de manière unifiée à l'aide de modèles d'une part, et l'interprétation de ces modèles à l'aide d'un générateur de code d'autre part. Par exemple, le Langage de Modélisation Unifié (UML) [31] est un langage de modélisation générique permettant de concevoir, à l'aide de modèles, toute application Orientée Objet (OO). Les modèles UML sont utilisés à des fins d'illustration, de génération de code [32] ou de génération de cas de test [33] pour n'en citer que quelques-uns. Dans cette thèse, nous nous dirigeons vers le même objectif, mais pour l'IdO.

Par ailleurs, l'IDM a été mise en uvre avec succès aux systèmes adaptatifs et distribués, à savoir l'approche Model@Runtime [34] et dans la sécurité dirigée par les modèles [116]. De plus, un modèle peut être utilisé à diverses autres fins telles que l'analyse de la sécurité du réseau et l'évaluation de ses vulnérabilités [35], entre autres.

Dans cette thèse, nous proposons d'étudier dans quelle mesure l'IDM peut fournir des outils de génie logiciel cohérents pour concevoir un réseau d'objets connectés et hétérogènes. Nous proposons une méthodologie de génie logiciel basée sur les modèles en ce sens. Celle-ci comprend trois volets. Premièrement, la modélisation d'un réseau d'objets, deuxièmement le contrôle d'un réseau d'objets, troisièmement la génération du code du réseau d'objets. Nos résultats montrent que cette méthodologie présente les avantages suivants : une économie en termes de ligne de code, une séparation des responsabilités et une meilleure communication des parties prenantes.

Modélisation d'un réseau d'objets

Dans ce premier volet, nous introduisons des concepts pour la modélisation d'un réseau d'objets. La modélisation consiste en deux parties, notamment la partie théorique basée sur l'IDM et la partie d'interface, c.-à-d. l'ensemble des outils permettant d'exploiter la partie théorique, basé sur un Langage Dédié (LD) équipé d'un système de détection des anomalies dans le modèle.

Dans la partie théorique, nous réifions les principaux concepts nécessaires pour la modélisation d'un réseau d'objets. Les concepts introduits (par exemple, canal, réseau, utilisateur, rôle) reposent sur la théorie de l'information de Shannon [134] et s'intègre dans la couche M2 de l'Architecture Dirigée par les Modèles (MDA), initié par le consortium Object Management Group (OMG) [68].

Nous utilisons certains outils d'IDM déjà établis, à savoir ThingML, pour modéliser le comportement d'un objet et Xtext, pour créer la syntaxe du LD. Nous concentrons notre thèse sur la modélisation des concepts primitifs pour connecter les objets. La modélisation, ayant lieu à la couche M1, utilise les concepts réifiés dans la couche M2, pour créer le modèle. Les modèles permettent d'éviter les détails techniques, qui sont source d'hétérogénéité. En effet, grâce aux modèles, un ingénieur logiciel peut créer un modèle spécifiant un réseau d'objets en utilisant uniquement les concepts primitifs réifiés. Ainsi, seuls les aspects nécessaires pour créer la logique commerciale du réseau sont attendus. La génération des artéfacts de bas niveau, permettant de passer de la couche M1 à la couche M0, consiste en une procédure spécifique interprétant ce modèle, selon le langage de programmation cible.

Techniquement, nous supposons que le comportement d'un objet est basé sur ThingML-DSL (TH-DSL) [90]. Celui-ci permet de spécifier un comportement à l'aide d'un diagramme états-transitions, avec la possibilité d'inclure des fonctions et des propriétés qui peuvent être appelées depuis un état. TH-DSL fournit une syntaxe basée sur la grammaire Xtext, représentant la forme textuelle d'un modèle Eclipse Modeling Framework (EMF), appelée ThingML-Model (TH-Model). Nous avons créé un LD appelé CyprIoT-DSL (CY-DSL), dédié exclusivement à la modélisation du réseau. Il fournit notamment les concepts primitifs pour lier les objets spécifiés avec TH-DSL. En effet, CY-DSL est basé également sur Xtext, qui est donc une représentation textuelle d'un modèle EMF. EMF propose un mécanisme, appelé référencement inter-modèles, qui permet d'intégrer d'autres modèles EMF. Nous utilisons ce mécanisme pour importer le TH-Model à l'intérieur du modèle du réseau, appelé CyprIoT-Model (CY-Model).

CY-DSL est intégré dans l'Environnement de Développement Intégré (EDI) Eclipse. Il bénéficie donc des avantages d'un EDI (par exemple, coloration syntaxique, rapport d'erreurs, auto-complétion, débogueur, refactoring) [144], lui permettant d'augmenter la productivité des ingénieurs logiciels.

CY-DSL applique des validateurs syntaxiques et sémantiques pour assurer que la modélisation du réseau est correcte. En effet, ces validateurs aident à prévenir certaines incohérences (par exemple, communication utilisant des formats de message incompatibles, port et chemins incompatibles) en les signalant sous forme d'erreurs ou d'avertissements dans la console de l'EDI. Ce mécanisme permet de gagner du temps en aidant à anticiper les bugs lors du déploiement du réseau [145, 146].

Contrôle d'un réseau d'objets

Le volet précédent montre comment créer un modèle du réseau à l'aide de concepts unifiés au niveau du modèle. Dans ce deuxième volet, nous montrons comment utiliser ce modèle pour contrôler le comportement du réseau à l'aide de politiques déclaratives. Le contrôle correspond à la faculté d'injecter des moniteurs permettant soit de restreindre la communication soit de déclencher des actions.

Le contrôle permet une spécification plus précise du comportement attendu du réseau, moyennant des politiques. L'unification des concepts de réseautage au niveau du modèle facilite cette spécification et évite les problèmes de mise en uvre de bas niveau. En effet, il est plus difficile de spécifier de telles contraintes lorsqu'on intervient directement sur le code (hétérogène) de bas niveau.

Le réseau applique les politiques, qui contiennent des règles. Ces règles ont une structure spécifique et appliquent des actions en fonction des éléments du réseau spécifié, par exemple, l'objet, le canal, l'utilisateur. Une politique vise à garantir que l'application d'IdO se comporte comme prévu du point de vue d'une partie prenante telle que, par exemple, un agent de sécurité, un gouvernement ou le propriétaire du réseau.

La structure de règles offre deux types de contrôle d'un point de vue théorique, à savoir un contrôle de la communication et un contrôle du comportement des objets. Cette structure peut aussi servir à divers objectifs de contrôle à l'avenir, jusqu'à présent, nous avons choisi de mettre en uvre les deux types de contrôle suivants :

— **Contrôle de la communication [150] :** consistant à *interdire* ou *autoriser* l'envoi ou la réception de messages entre deux entités. L'entité peut être, entre autres, un objet, un utilisateur ou un canal. Par exemple, nous pouvons spécifier une règle pour refuser ou autoriser l'échange de messages entre deux objets, deux utilisateurs, deux rôles ou la combinaison de certains d'entre eux.

— **Contrôle du comportement de l'objet [140] :** consistant à appliquer une action sur l'objet en fonction de l'état actuel d'un autre objet. En effet, comme le comportement de l'objet consiste en un diagramme états-transitions, le contrôle vise donc à modifier le comportement encapsulé dans ce diagramme, de manière à satisfaire la règle. Par exemple, une règle peut spécifier qu'un objet t_x doit passer à l'état s_i lorsque l'état actuel d'un objet t_y est s_j.

Génération du code d'un réseau d'objets

Dans les deux précédents volets, nous introduisons les concepts pour créer une spécification rigoureuse d'un réseau d'objets. Ce dernier volet présente le générateur de code, appelé CyprIoT Code Generator (CY-CGEN), qui a la fonction de générer le code de bas niveau implémentant la spécification du réseau et des politiques, tous deux écrits en CY-DSL.

La génération de code se fait en trois étapes : a) charge le modèle du réseau contenant les TH-Models et les politiques, b) transforme les TH-Models pour les connecter selon la spécification du réseau et applique les politiques c) génère les artéfacts de bas niveau (par exemple, le code, la documentation, les règles de contrôle d'accès).

Le CY-CGEN se base sur la Transformation de Modèles (TM). La TM [78] est un processus basé sur des règles de transformation qui prend un ou plusieurs modèles en entrée pour produire un artéfact logiciel en sortie. Il existe deux types de transformations : Modèle à Modèle (MàM) et Modèle à Texte (MàT). Le premier type produit un modèle en sortie tandis que le second produit du texte en sortie, à partir du modèle d'entrée. Nous utilisons les deux types pour l'interprétation du modèle du réseau, encapsulé dans le CY-Model.

Une transformation MàM nous permet d'adapter le TH-Model (c'est-à-dire le comportement de l'objet) selon le CY-Model (c'est-à-dire la spécification du réseau) au niveau du modèle. Ce mécanisme prend les informations du CY-Model et ajoute *uniquement ce qui est nécessaire* au TH-Model, afin d'être conforme à la spécification du réseau. Comme ce processus se déroule au niveau du modèle (en utilisant des concepts unifiés), l'interopérabilité des objets est préservée. Les TH-Models transformés sont ensuite utilisés pour générer leur équivalent dans le code de bas niveau à l'aide du générateur de code de ThingML, appelé ThingML Code Generator (TH-CGEN).

Une transformation MàT nous permet de générer des artéfacts connexes qui ne font pas partie du comportement interne d'un objet (par exemple, règles de contrôle d'accès, fichier de configuration, documentation) et qui sont nécessaires au bon fonctionnement du réseau. Les informations nécessaires à la production de ces artéfacts se trouvent généralement dans la spécification du réseau, mais pour ces artéfacts elles doivent être écrites dans le bon format.

Par ailleurs, nous avons opté pour une architecture modulaire, favorisant l'extensibilité et la séparation des responsabilités. Le noyau du CY-CGEN fournit certaines caractéristiques de base nécessaires à la mise en uvre d'un réseau de base, c'est-à-dire connecter les objets et appliquer les politiques. Ce noyau ne met pas en uvre ce que nous appelons les connaissances d'expert, mais propose de faire cela à l'aide de plugins. Par connaissances d'expert, nous entendons la capacité à utiliser les informations du modèle du réseau pour extraire des connaissances plus avancées, éventuellement à l'aide d'une procédure plus spécifique. Cela peut conduire à la génération automatique d'artéfacts logiciels, qui nécessite normalement une certaine expertise, ou à l'introduction d'une expertise dans le processus de génération du code.

Résultats

Nous avons testé notre méthodologie sur des réseaux allant de 1 à 25 objets. Nous utilisons les méthodes de génie logiciel traditionnelles comme base de référence, avec des objets basés sur C, Java et Arduino. Nous avons comparé le nombre de lignes de code nécessaire entre notre base de référence et la méthodologie proposée dans cette thèse.

Nos résultats montrent que notre méthodologie permet d'économiser une quantité significative de lignes de code comparée aux méthodes traditionnelles d'ingénierie logicielle pour l'IdO. Le taux de gain de lignes de code varie entre 64.29% et 97.89% pour plusieurs cas d'utilisation. Ce gain correspond à la partie générée automatiquement par CY-CGEN.

Conclusion

La présente thèse plaide en faveur d'une méthodologie intégrée de génie logiciel basée sur des modèles pour concevoir et déployer des réseaux d'objets. L'IDM est un paradigme prometteur pour répondre à certaines exigences de l'IdO, notamment les problèmes d'hétérogénéité et d'interopérabilité. Notre méthodologie rend le cycle de développement d'une application d'IdO plus rigoureux et permet de définir précisément les responsabilités. La première contribution de cette thèse est une nouvelle approche de conception des applications d'IdO. La deuxième contribution consiste en un formalisme nouveau pour modéliser un réseau d'objets hétérogènes. La troisième contribution consiste en un mécanisme de contrôle pour permettre de mettre en uvre des contraintes au réseau. La quatrième contribution consiste en un générateur de code pour interpréter un modèle abstrait du réseau et générer ses artéfacts logiciels de bas niveau. À notre connaissance, il n'existe dans la littérature aucune méthodologie fondée sur les modèles qui fournisse des outils aussi intégrés pour créer un réseau d'objets, mettre en uvre des politiques et générer le code de bas niveau.

INTRODUCTION

The Internet of Things (IoT) is reshaping our society's relationship with technology. Prominent applications such as Smart Homes [1], Wearables [11], and Smart Cities [12] make the IoT more and more visible in our everyday life. It is expected to see more IoT applications in the near future [13]. Gartner reports that more than 14 billion connected devices are already in use, and forecasts that this number will grow to 25 billion by 2021 [14].

The IoT aims for connecting Anything, Anywhere, Anytime (AAA) [15]. In particular, connecting things of different sizes ranging from bacterias [16] to supercomputers. Manifestly, each thing has its requirements. For instance, a tiny sensor may require a Constrained Application Protocol (CoAP) [17] client to connect to the World Wide Web (WWW), because of its limited resources, while a laptop requires a standard Hypertext Transfer Protocol (HTTP) client. 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 [18]. On the one hand, many stakeholders are involved (e.g., Security, Business, Network), each using its own tools and methods [19]. On the other hand, heterogeneity of software technologies (e.g., languages, protocols) [20] causes, inter alia, an interoperability problem by hindering communication between things. Commonly, a typical IoT application contains multiple computing platforms, languages, and protocols from various ranges. Besides, every day a new thing emerges, 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 [21, 22, 23, 24, 25] suggest that it 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 expertise, lack separation of concerns [26], and lead to poorly tested and insecure

IoT applications [18, 27]. 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 [28, 29].

As a matter of fact, for security experts, existing software engineering models based on a quick-fix approach, i.e., fixing the bug when it arises at runtime, have shown their limits w.r.t. security. The SANS Institute reports that almost 90% of security professionals affirm that changes to security controls are required in the IoT [30].

Clearly, the IoT needs a novel software engineering approach adequate to its requirements. MDE is a promising paradigm having the potential to overcome 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 the interpretation of these models using an automatic code generator on the other. For instance, the Unified Modeling Language (UML) [31] is a generic modeling language to design, using models, any Object-Oriented (OO) application. UML models are used for illustration purposes, code generation [32] or test cases generation [33] to cite a few. In the present thesis, 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.

MDE has successfully been applied to some adaptive and distributed systems, namely the Model@Runtime approach [34] and in model-driven security [116]. The latter finds its use in various purposes, such as security analysis and threat assessment [35], to name just a few.

In this thesis, we propose to investigate the ability of MDE to provide consistent software engineering tools to design an IoT application. Concretely, we offer a model-based software engineering methodology for a) the modeling of a network of things, b) the control of this network with a policy, and c) code generation. The network enables to connect things according to a topology, where each thing has a statechart-based behavior, the policy relies on the elements of the network (e.g., state of a thing, time) to force a runtime behavior, and the code generator interprets the two to generate the low-level software artifacts. We use lines of code as our evaluation criteria to compare the proposed approach with the current state of practice.

This manuscript is structured as follows. Chapter 1 gives an overview of state of art.

Chapter 2 and Chapter 3 present the modeling abstractions to specify and control a network of things. Chapter 4 shows how model transformation bridges the gap between the model-based specification of the network and its concrete implementation. Chapter 5 embodies the whole methodology, provides an evaluation, depicts a concrete case study, and discusses the benefits and limitations. Finally, we present the conclusion and point out some future research directions.

Research Questions

We propose to investigate the following research questions (RQs):

RQ1: What software engineering process should be followed to design a network of heterogeneous things?

RQ2: How can MDE enable interoperability between heterogeneous things?

RQ3: How can MDE provide effective control mechanisms to regulate the behavior of a network of heterogeneous things?

RQ4: What are the qualitative and quantitative benefits of using a MDE methodology to design and implement a network of heterogeneous things?

Chapter 1

State of the Art & Context

1.1 From Network of Computers to Network of Things

From an evolutionary perspective, the connection is a natural human need [36]. Before the existence of the Internet, people used to connect physically either face to face, by postal mail, by telegram (1875), or by phone (1876). The emergence of digital electronics in the late 1950s enabled the proliferation of computers among the vast majority of people. While the connection was previously tied physically to space and time, the Internet enabled a virtual connection between people, no matter when and where.

1.1.1 The Conventional Internet

In 1960, the Advanced Research Projects Agency Network (ARPANET) was the first major attempt to interconnect computers; it relied on the Internet Protocol Suite (known as TCP/IP) to enable a standard communication and used the concept of packet switching which is the basis for data transfer on the Internet today. In 1985, based on the advancement of the ARPANET and the contribution of the European Organization for Nuclear Research (CERN), The National Science Foundation Network (NSFNET) [37] created a larger system of interconnected university-based computers and grew into what is called nowadays the Internet.

The emergence of personal computers, brought computing and connectivity closer to individuals, leading to the adoption of the Internet at large scale. The technology has since then evolved. As it was predicted by Moore's law [38], the number of transistors in an integrated circuit doubles every two years. Consequently, bringing computing power to smaller devices. This factor paved the road to the Internet of Things (IoT).

1.1.2 Towards the IoT

The term IoT was coined for the first time by Kevin Ashton [39] in 1999 while working at MIT Auto-ID Labs. According to him, the conventional Internet is centered primarily around human beings as the only providers of data. As human beings have limited time, attention, and accuracy, his data are also limited. Whereas, things have the potential to generate unlimited and accurate data (e.g., sensing the environment) faster and cheaper. This new requirement challenges the TCP/IP architecture, as heterogeneous sources of information could be involved.

Remark

In the scope of the present thesis, we define a thing as any device regardless of its size that has computing power and can connect to a network, regardless of the communication mean.

The IoT is a generalization of the conventional Internet [40]. Indeed, while the data of the conventional Internet has been mainly generated directly by input devices such as a keyboard or mouse, the IoT is a paradigm meant to generate data from pre-programmed things. For instance, collecting data from sensors and producing actions by actuators, thus naturally expanding the scope of possibilities for an ambient intelligence [41], i.e., the ability for things to be more useful for people.

The IoT makes the computing ubiquitous. Indeed, while the desktop and personal computers were so far the norms, tiny devices such as sensors and actuators are expected to be also more present, enabling connectivity in a distributed and decoupled manner. Ultimately, the IoT aims for connecting AAA [15].

Today, no consensus has been yet reached on the exact definition of the IoT. Atzori et al. [15] view the IoT as the idea of connecting things or objects in a way that they can accomplish common goals. Miorandi et al. [42] see it as an umbrella keyword for the extension of the conventional Internet to the physical world by incorporating sensor and actuator devices. Gubbi et al. [135] consider that the integration of tiny sensors and actuators to the conventional Internet, defines and justifies the creation of the IoT as a paradigm. Al-Fuqaha et al. [43] describe the IoT as the idea that transforms traditional objects into more useful objects by exploiting emerging technologies such as pervasive computing, embedded devices, communication technologies, and sensor networks.

The IoT still faces several obstacles related to its engineering, scalability, and deployment. For instance, software-engineering-wise, the IoT lacks a set of best practices to build scalable, robust, and secure applications [44]. Deployment-wise, the intrinsic heterogeneity of the IoT creates an interoperability problem between things, restricting the ability to achieve seamless smart scenarios [45].

Remark

The concept of smart scenario [46] 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)

1.1.3 Software Engineering for the IoT

As stated by Atzori et al. [15], the IoT is a paradigm with many visions. This observation partly explains the lack of a widely accepted software engineering approach. Still, numerous authors agree on the one hand that a) it aims to connect AAA [47, 48, 49, 50, 51], and on the other that b) all software engineering approaches aim to leverage the smart side of interconnected objects [52, 53, 54, 55]. There is, therefore, a pressing need for a unifying software engineering approach to meet these two crucial requirements. In this thesis, we introduce a methodology based on models that is heading towards this objective.

Presently, software engineering for the IoT consists of programming each thing to fit the needs of the application [56, 57, 58, 59]. The existing approaches require a significant amount of time, as many skills are required, from networking and security to programming. 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. Such an approach is less likely to scale, typically in very large networks involving Wireless Sensors and/or Actuators Network (WSAN), as it lacks a trace and relies on programming each thing separately, thus making engineering indeterministic, time-consuming, repetitive and prone to bugs.

In most scenarios, the IoT will rely on the conventional Internet's backbone [135] for communication unless the conditions are inconvenient, namely because of resources or the requirements of the application. The layers lower than the data layer in the Open Systems Interconnection (OSI) model receives a consensus w.r.t. the standards introduced for the IoT [136]. The current work aims to contribute to the data layer, one level higher.

Several protocols emerged to address the communication requirements at the data level. Message Queuing Telemetry Transport (MQTT) [139] is a publish and subscribe protocol used generally for a decoupled communication between resource-constrained devices. A publish and subscribe communication consists of a sender, a receiver, and a broker. The sender and the receiver communicate as an intermediate between them. The advantage of using a broker is that it can store the message published by a thing in a topic, and send it to the things that subscribed to this topic in a decoupled manner. Z-Wave [10] is a radio protocol dedicated to smathomes to send data between things; it enables the communication between Z-Wave equipped products and supports mesh networks. Zigbee [8] is a protocol dedicated to personal area networks (PAN), i.e., interconnecting devices centered on an individual person's workspace. It supports multiple types of networks: point-to-point, tree or mesh networks. These protocols often target a specific range of things, and provide little or no interoperability between them.

Furthermore, the previous challenges were owned by the IoT too, mainly due to its distributed and decoupled nature. In fact, the conventional Internet was not meant for scalable interoperability between nodes. Currently, the applications running on top of the Internet (e.g., WWW, chat, peer-to-peer, audio/video-conference) are based on standards (e.g., HTTP, MQTT, BitTorrent, VoIP) and function as silos. Each silo runs independently from the other. During the creation of the Internet, interoperability between these silos was not an urgent need. Thus, standards were a good compromise to bring connectivity to these quite homogeneous nodes. At that point, the IoT is different; interoperability is instead crucial to enable smart scenarios. Arguably, the next milestone is to bring interoperability among these silos.

On another note, the fifth generation of the cellular network (5G) covers the connectivity of resource-constrained devices [60], specifically for Low Power Wide Area (LPWA), but still requires using different protocols, namely NB-IoT or eMTC, that fit best the requirements of the thing. Not to mention that standardization takes time and has high upfront costs to be adopted at a large scale by providers. Indeed, heterogeneity will undeniably persist, hence the need to embrace it instead of containing it in a standard.

Remark

Heterogeneity refers to the various software technologies required at a low-level to create a network of things. It includes primarily heterogeneity of programming languages (e.g., C, Java, Arduino) and protocols (HTTP, MQTT, Zigbee).

Some approaches tend to contain this interoperability problem with standardization (e.g., standard architecture, standard platform) [61, 62, 63, 64, 65]. Standards lack inclusiveness, as certain things could not afford their cost (e.g., computing resources required, cost of adoption) [16]. We believe that this is not a scalable strategy for the IoT to reach its ultimate goal, i.e., connecting AAA. In the present thesis, we tackle this problem at the software engineering phase using MDE. Thus, embracing heterogeneity as an intrinsic feature of the IoT rather than trying to eliminate it with standards.

1.2 Model-Driven Engineering

Model-Driven Engineering (MDE) is a software engineering paradigm where *"everything is a model"* [66]. Indeed, every software artifact (e.g., specification, test suites, source code) can be seen as a model of a lower-level artifact or system. The objective of such formalism is to reuse, maintain, update, and test software artifacts using automatic tools.

1.2.1 Modeling

MDE is an umbrella keyword for the paradigm advocating the use of models as the main unit in designing software [67]. By representing everything as a model, MDE offers machine-ready access to the process of software engineering. Thanks to that, several steps of this process can be automated.

The Model-Driven Architecture (MDA) initiative is a set of guidelines proposed by the Object Management Group (OMG) in 2001 as a standard approach for model-based software engineering [68]. Basically, the MDA consists of separating the functional specification from its implementation using a four-layers architecture. It focuses on means to produce concrete artifacts from abstract diagrams using a technique called Model Transformation (MT) (cf. Section 1.2.2). Generating code (concrete artifact) automatically from its abstract specification (abstract diagram) is an example of MT.

The MDA provides several layers (cf. Figure 1.1), each corresponds to a technical space and fulfills a specific duty on the whole architecture. Figure 1.1 shows the four-layers architecture. The highest layer, i.e., the meta-metamodel (M3-Layer), defines the language to specify the metamodel. The meta-metamodel, being the highest level of abstraction, has been sufficiently tackled in the literature, and it is out of the scope of this work. Its two common implementations are Meta-Object Facility (MOF) [69] and Ecore, a metametamodeling solution offered by the Eclipse Modeling Framework (EMF) [70]. The EMF is a state-of-art standard framework, maintained by the Eclipse foundation [1], to design models. Further, we use EMF in the implementation of the methodology proposed in the present thesis (cf. Section 2). The next layer, i.e., the metamodel (M2-Layer), defines the concepts of a specific domain to specify its model at a lower level (M1-Layer).

The most used meta-modeling language in MDA is the UML. The UML provides metamodels (M2-Layer) to specify, in the form of diagrams, various aspects of an OO application. These diagrams represent the UML model. So, UML can be used to cre-

1. https://www.eclipse.org/

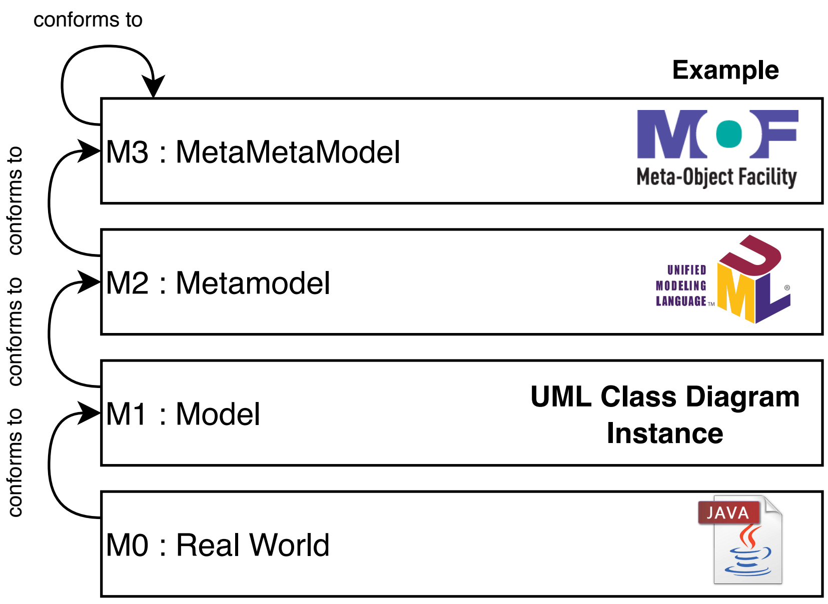


Figure 1.1 – The four-layers architecture of MDA

ate a model of a Java application consisting, e.g., of a class diagram [71], an activity diagram [72], and a communication diagram [73]. Because the model is designed with a standard formalism, it can be interpreted automatically by some MDE tools [74] for several purposes, such as the generation of low-level code (e.g., Java, C++ or C#), the generation of the documentation or the test cases [33].

One common way to specify a model consists of using a Domain-Specific Language (DSL). A DSL, in contrast with a General-Purpose Language (GPL) (e.g., C, Java, Go), provides a specific solution to a particular field, such as the IoT in our case. In general, the creation of a DSL requires two essential ingredients: a parser for the syntax and a semantic analyzer to validate the meaning of the expressions. Several tools exist to facilitate the creation of a DSL. Among them, we can cite Xtext [75], MPS [76] and Spoofax [77]. Xtext, in particular, relies on EMF for parsing; it converts any parsed file into an EMF model. Thus, an Xtext-based DSL offers a way to create a model in a textual form. Further, we use Xtext for the implementation of our DSL solution.

1.2.2 Model Transformation

Model Transformation (MT) [78] is the process that takes one or more input models to produce a target artifact (either a model or text) by means of transformation languages.

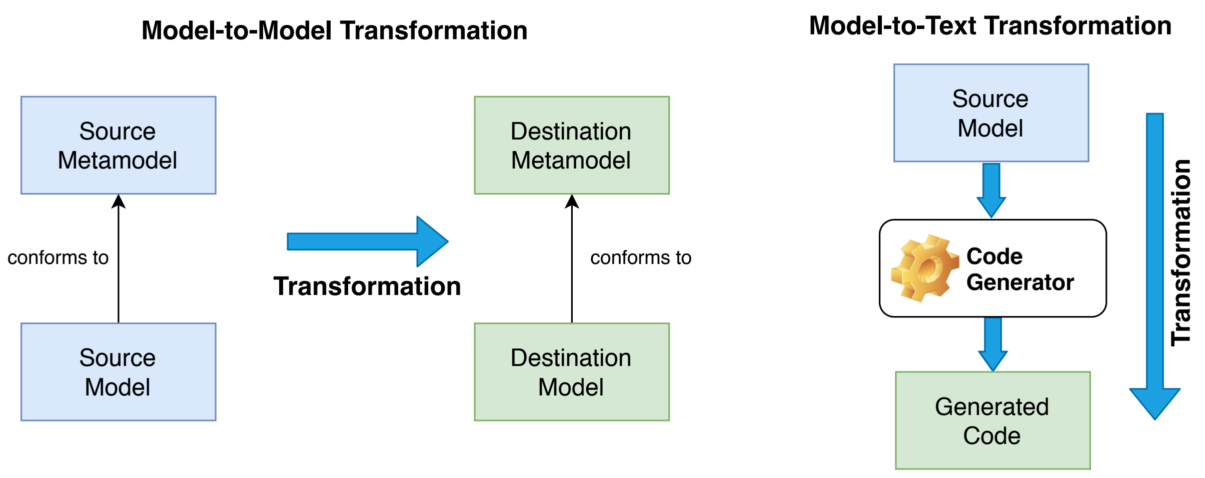


Figure 1.2 – The two types of model transformation

In this part, experts are expected to map the abstract concepts to some other concepts, either at the same level of abstraction or at a lower level. In the present thesis, we use MT to transform the specification of a network, containing things and policies, into the network's concrete software artifacts.

Figure 1.2 illustrates how MT occurs. There are two types of transformations: Model-to-Model Transformation (M2MT) and Model-to-Text Transformation (M2TT).

Model-To-Model Transformation

M2MT aims to produce a *target model* artifact from the input model. For instance, this mechanism allows us to adapt the model of a thing according to some properties (e.g., topology) of the network model.

Among the main transformation tools dedicated to M2MT, we can cite Query/View/-Transformation (QVT) (standard) [83], Tefkat [79], Kermeta [80] and ATLAS Transformation Language (ATL) [81].

QVT is an OMG standard that defines a set of languages to express M2MT. It consists of two main languages: QVT Relations (QVT-R), a declarative language, and QVT Operational (QVT-O), an imperative language. To date, the literature lacks a complete implementation of the standard, Eclipse MMT [82] is considered as its most advanced implementation attempt. Some languages such as ATL and Tefkat implements several concepts introduced by QVT.

Tefkat is a declarative transformation language with a very expressive and readable syntax. However, the language has no support for imperativeness. This limitation complicates its adoption for some complex transformations, as this complexity can be hard to express merely using a declarative style [84].

Kermetta is a platform for the creation of a rich development environment based on metamodels. It provides an imperative language dedicated to executable meta-modeling, i.e., a way of adding operational semantics to the metamodel. Kermetta is not a transformation language, but offers a means to write a M2MT in an imperative way within this environment. The verbosity and the lack of readability (due to its imperative style) make writing complex transformations difficult.

ATL is mostly a declarative language, based on EMF and a readable syntax. It supports imperative instructions; however, as recommended by the authors, this must be used only if necessary. ATL permits to achieve transformations across different EMF-based models using transformation rules. We use ATL for M2MT in the present thesis as it is easy to use, readable, well documented, and provides both declarative and imperative styles. Listing 1.1 depicts an example of a basic rule in ATL. This rule transforms the ElementX of an input model conforming to the metamodel MetamodelA, to a target model conforming to the metamodel MetamodelB, by filling its attributeB with attributeA from the input model. The same principle can be used to extract any information from the input model (e.g., model of the network) and adds it to the target model (e.g., model of the thing).

```
rule basicRule {
    from
        i: MetamodelA!ElementX
    to
        t: MetamodelB!ElementY (
            attributeB <- i.attributeA -- Replacing with attributeA from the input model
        )
}
```

Listing 1.1 – A basic transformation rule in ATL

Model-To-Text Transformation

M2TT aims to produce a *target text* artifact from the input model. For instance, this mechanism allows us to generate some textual artifacts that are not part of the internal behavior of a thing (e.g., access control rules, configuration file, documentation). In general, the information necessary to make these artifacts exists in the model of the network, yet it needs to be written in the syntax of these artifacts.

Among the main M2TT languages, we can cite Meta Object Facility Script (MOFScript) [85] and Acceleo [86].

MOFScript is an imperative M2TT language that permits the generation of the text output. It also relies on an OMG standard, named MOFM2T, that complements QVT. MOFScript is based on EMF and uses templates and rules for the generation of the text output. It comes as an Eclipse plugin.

```
[template network2documentation(n : Network)]
[file (n.name.concat('.md'), false)]

# Information about the network
* Name : [n.name/]
* Domain : [n.domain\/]

# Available things
[for (th : Thing | n−>getThings())]
* [th.name/]
[/for]
[/file]
[/template]
```

Listing 1.2 – A basic Acceleo template to generate a documentation in Markdown syntax

Acceleo is a recent M2TT language that supports EMF models. It uses templates with a readable syntax and some built-in features such as advanced string manipulation, error detection, code completion, refactoring, and a tractability mechanism that allows tracking down the source of an element in the text output. Unlike MOFScript, Acceleo has a feature called *Incremental Generation* to modify a generated code and protect it against future generations. We use Acceleo for M2TT in the present thesis because of these benefits. Listing 1.2 shows a simple template to generate some documentation from a model to a markdown syntax output. Acceleo permits operations ranging from basic ones such as simply writing a string (e.g., writing the name of the network with n.name) to more advanced ones such as a for loop (e.g., Lines 9 to 11). The same principle can be used to extract any information from an input model (e.g., model of the network) and write it in various textual output artifacts (e.g., access control rules, configuration file).

1.2.3 MDE and IoT

Generally speaking, software engineering consists of five phases: understanding and specifying requirements, design, implementation, testing and deployment [87]. Without a transparent separation of concerns, multiple factors can undermine this software engineering cycle; communication issues, lack of skills, and misalignment of business needs and resources are some of these factors [88]. The methodology presented in this thesis contributes to the design and implementation phases and helps separate concerns.

MDE can dissect the complexity of the IoT through the four-layers architecture as shown in Figure 1.3. Indeed, we can create metamodel for the IoT at the M2-Layer to model an IoT application based on unified concepts at the M1-Layer, consequently avoiding heterogeneity at the M0-Layer. This heterogeneity can be delegated to an automatic code generation process where we map abstract concepts to their low-level equivalent for each target platform.

So far, we have seen that the IoT can play the role of the requirement provider. Ideally, MDE, on the other hand, can play the role of the solution provider. In reality, the two paradigms share an interesting commonality, they both seek genericity; the IoT seeks to connect generic things, while MDE seeks to unify and automate the development of generic software artifacts [89].

The use of MDE as a software engineering approach for the IoT has been tackled in the literature. We explore in what follows the most relevant approaches. From our literature review, two groups emerged. The approaches that use MDE for modeling the things and those for modeling the network. The former group corresponds to the keyword *things* of the Internet of Things (IoT), and the latter to the keyword *internet*.

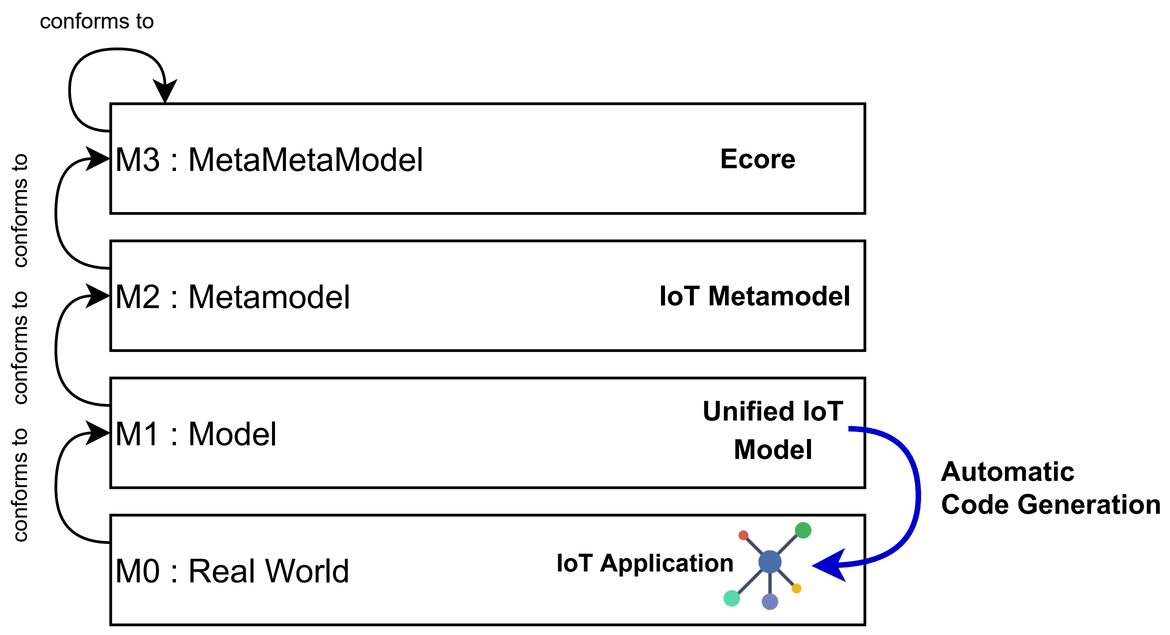


Figure 1.3 – The four-layers architecture of MDA applied to IoT

Modeling the thing

The modeling of a thing consists of using abstract concepts to describe its internal behavior. We can distinguish two ways; the first consists of mapping the behavior into some established formalism such as a statechart or a workflow chart, and the second consists of defining the behavior with an unconventional formalism. The former benefits from interoperability with the established tools, while the latter generally reflects better the reality using some intuitive concepts.

Harrand et al. propose ThingML [90], an approach based on EMF consisting of two underlying components: ThingML-DSL (TH-DSL), a DSL to specify a model of a thing, and ThingML Code Generator (TH-CGEN), a code generator that produces the low-level code from this model. TH-DSL abstracts the behavior into a statechart [91, 92] on the one hand, and the code generator reproduces this statechart using the low-level code concepts (e.g., C, Java, Javascript, Go) on the other. TH-DSL supports the embedding of low-level code (i.e., inserting chunks of low-level code inside the model) and provides a means to connect a single thing to a network via a protocol. Nevertheless, ThingML lacks abstractions to create a network, i.e., making things collaborate towards a common goal according to a defined network scheme.

Yakindu [93] is another statechart-based tool to design the behavior of a thing inside a visual editor. It provides a code generator for C/C++, Java, and Python. Contrary to ThingML, Yakindo lacks the means to connect a thing to a network and the possibility to embed the low-level code.

Eclipse Vorto [94] uses an unconventional formalism to abstract the thing's capabilities. Its underlying concepts are: information model, function, operation, and attribute. A function consists of a set of attributes and a set of operations. The functions are grouped inside the information model. Eclipse Vorto proposes to use code generators to produce code by interpreting the information model. The solution also offers a repository to share and reuse information models and code generators. Compared to ThingML, modeling the behavior is rather limited; only a few operations are achievable, and networking is not covered.

Fuch et al. [95] propose to program a thing using a UML2 Activity Diagram (UAD). The behavior is designed in the form of activity. The latter is transformed into a script to be executed by an interpreter running on the thing. Activities can communicate between each with their input/output interfaces via a prototypical communication protocol suggested by the authors. However, the approach focuses merely on the advantages of UADs to ease collaboration between things. Also, only one type of thing (namely SUN Spot [96]) has been considered in their study.

Modeling the network

Modeling the network consists of *wiring* things through their external interfaces, to form a network of things. Indeed, the model of the network contains the behavior of things and the interactions between them. Because of the intrinsic heterogeneity of the IoT, i.e., presuming that we must be able to connect things regardless of their programming languages or protocols, the wiring of things is difficult using their low-level heterogeneous concepts. Hence, the need for higher abstractions free from the technical considerations to enable seamless wiring.

Remark

The wiring consists of linking the ports of things via channels.

Remark

A channel is an abstract concept defining the medium that is enabling a communication between two things.

Remark

A port is a logical address within a thing that serves to exchange messages with the outside.

The existing approaches for network modeling are disparate; some approaches target Wireless Sensor Networks (WSN) (a network of many tiny sensors dedicated to collecting data) [97], others target the web of things (the IoT using existing web technologies) [98] and others target a specific category of IoT applications. According to our literature review, we noticed a lack of a full-scale, documented, and open-source approach for modeling a network of things. In what follows, we discuss the existing approaches extensively under this strand, as a significant part of the present thesis contributes to this area.

Ciccozzi et al. [99] presents a conceptual model of Mission-Critical Internet of Things (MC-IoT) systems. These systems are characterized by a significant need for dependability, safety, security, and availability due to their intolerance to failure [26]. The conceptual model depicts the relationships between the various entities of a typical MC-IoT system. The authors merely point out some directions on applying MDE to these types of IoT systems.

Dietterle et al. [100] present a way to map the Specification and Description Language (SDL) [101] with TinyOS component models [102] to enable a formal description of communication protocols. TinyOS is an open-source operating system designed for wireless sensor networks. Then, a general scheme for creating code from these models is proposed. This approach focuses only on using TinyOS at runtime, making it less generic than our methodology.

Salihbegovic et al. [103] present a Visual Domain-Specific Modeling Language (VDSML) based on a JavaScript editor. It aims at giving an IoT engineer a user interface to design an IoT system virtually. Only a set of predefined things is available to use within their editor. The tool offers a code generator for the configuration file of OpenHab [104]. However, the formal specification of the language, such as the metamodel, is not provided.

Therefore, it is difficult to assess the scope of their contribution and evaluate whether the language is extensible beyond the limited set of things they mention.

Amrani et al. [105] introduce a DSL to design a network of things. The language allows to declare the possible actions of a thing. Those actions must 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 things is not conceptualized in the DSL metamodel. The code generation is not discussed.

Bertran et al. [106] present a tool based on the Sense/Compute/Control (SCC) paradigm [107]. It consists of a DSL, a generator of Java code interfaces, a simulator, and a deployment framework. Although the DSL can abstract the specification of a thing, the engineer must implement its behavior in Java after code generation. Also, the framework assumes that things are capable of running Java, while this may not be the ideal programming language choice for certain ranges of things.

Glombitza et al. [108] provide an approach to model a thing as a web service so that it can communicate with the established web services in the WWW. They permit to compose these web services using a state-machine-based DSL. The DSL comes with a code generator that generates the C++ code. The authors assume that the communication between these web services is based on their own novel protocol [109].

Node-red [110] is a flow-based visual tool allowing to connect things, APIs, and online services using a browser-based editor. It presumes that the things are already deployed. Basically, it enables to map the output of a thing to another's input; this mapping is made from their editor. Node-red is centralized and useful only to create a *"mashup"* [111, 112, 113] of existing applications or services; it does not cover the creation of a network from scratch.

Einarsson et al. [114] propose a DSL dedicated to the modeling of smart home applications. The DSL can describe the interaction of a thing with a cloud platform. The authors assume that all things communicate homogeneously, i.e., using one single protocol. A model-to-text transformation is applied to the smart home model to generate the code. The approach covers only the interactions with a cloud platform; local home networks are out of the scope of their approach, although this may be added in the future.

Furthermore, we also found some open source ecosystems such as OpenHAB [2], Domoticz [3] and Home Assistant [4] that targets exclusively smart home applications.

2. https://www.openhab.org
3. https://www.domoticz.com/
4. https://www.home-assistant.io

In the present thesis, we provide a way to specify a network of things in the form of a model. Our approach aims to be inclusive enough to support any IoT application based on a network.

Modeling the control

The model of the network provides a global view of the application. This model can be exploited for various purposes. In the present thesis, we exploit this model for control purposes based on declarative policies. The function of these controls is to make sure that the network behaves as expected from the perspective of the policy maker (e.g., security officer, administrator, privacy officer, automation maker).

Remark

A declarative policy is a tool to constrain or influence the network's behavior according to some conditions.

We specify some constraints inside a policy using the attributes of the network. Then, we enforce these constraints depending on the context [115]. For instance, if our infrastructure permits it, we may write the control either directly in the thing or in the form of access control rules if the control must be handled by an external component. The ability to control the network is another significant part of the present thesis; we analyze some relevant approaches in this field.

Control can be seen as a generic word containing security, privacy, or automation. We explore in what follows the potential of applying MDE for control purposes. Some of the presented approaches use MDE, namely the Model-Driven Security (MDS) approaches, and some try to apply their own control approach and formalism.

MDS consists of using MDE for security purposes. Basin et al. [116] present a comprehensive overview of MDS studies. Most of them consist of modeling security requirements at the model level, then generating the security mechanisms at a low-level using a dedicated process. For illustration, Basin et al. show, using a concrete example, how the specification of a security policy is transformed by a code generation tool to control the behavior of a Graphical User Interface (GUI). These studies are designed to experiment specific security features of standalone applications and are not meant to cover distributed systems.

Lang et al. [117] propose a MDS approach to specify and enforce declarative policies, based on Attribute-Based Access Control (ABAC) [118]. ABAC is an access control paradigm that defines a policy using characteristics (known as attributes) of the system. Their approach permits to generate machine-enforceable access and logging rules. The authors illustrate their approach with a hypothetical intelligent transportation system.

Loddersted et al. [119] proposes an extension of UML to include security concepts, targeting distributed systems. Their approach relies on Role-Based Access Control (RBAC) [120]. RBAC is an access control paradigm where access decisions are based on the role assigned to a user. They extend the UML with some security concepts in the form of a UML profile. As a proof-of-concept, they generate an Enterprise JavaBeans (EJB) application enforcing the policy. The authors claim that the approach can improve productivity during the software engineering cycle. However, no quantitative study is included in their paper, while we propose to measure the benefits of our methodology using quantitative and qualitative evaluations.

Martínez et al. [121] propose an approach to obtain a Platform-Independent Model (PIM) of the global access control policy in a network. A PIM is a model encapsulating the domain structure and behavior, free from the technological platform used to implement it. Their approach uses the firewall configuration files in the system to extract all access control rules. Then, these rules are transformed into PIMs for each firewall and merged into a global access control model. A eXtensible Access Control Markup Language (XACML) policy can be easily derived from this model.

Neisse et al. [122] propose an open-source model-based security toolkit, to specify and enforce security policies. The enforcement consists of creating a monitor directly in the thing. To illustrate their approach, the authors apply it to a smart home case study.

Sicari et al. [123] proposes a policy framework for the IoT. The framework contains a specification language based on Extensible Markup Language (XML) and an enforcement engine targeting smart health applications.

Mavropoulos et al. [35] suggest a metamodel to describe IoT applications along with their security requirements. They use a DSL to specify hardware, software, social, and security elements. The approach is not meant for code generation, but rather for security analysis and visualization.

Furthermore, the OASIS consortium provides a framework to express and enforce policies [124]. It defines a declarative language called XACML to express an ABAC policy in XML. The framework decouples the access control logic from the low-level code. It relies on a request-response model where access control decisions are taken dynamically at run-time. It also defines the mechanisms to process this policy. The security framework needs a centralized Policy Decision Point (PDP) to evaluate access requests vis-a-vis the policy. La Marra et al. [125] proposes a solution to enforce a XACML policy in a decentralized fashion. Next Generation Access Control (NGAC) [126] proposes another approach to ABAC that decouples the access control logic from the low-level code. It is similar to XACML but differs on the expression of policies, the treatment of attributes, the computation of decisions, and the enforcement. Moreover, we could also find a study on access control languages that compared the benefits of XACML, XACL [127], APPEL [128], P3P [129] and EPAL [130]. The study concluded that XACML was the most suitable access control language for the IoT among the studied languages [131].

In the present thesis, we provide a way to specify policies at the model-level using the attributes specified in the network model. We then provide a code generation process that interprets these policies to enforce them in the low-level code.

1.3 Summary

The IoT faces several software engineering challenges to build scalable, robust, and secure applications. The intrinsic heterogeneity of the IoT creates an interoperability problem between things, restricting the ability to make smart scenarios. Presently, most of the existing software engineering approaches for the IoT consist of programming each thing to fit the application's needs. These approaches require a significant amount of time, as many skills are required, from networking and security to programming.

Recently, several model-based approaches emerged to tackle heterogeneity in the IoT. In this respect, the literature distinguishes 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; the first consists of mapping the behavior into some established formalism such as a statechart or a workflow chart, and the second consists of defining the behavior with an unconventional formalism. The former benefits from 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 ThingML. This work constitutes our baseline.

The modeling of a network consists of wiring things through their external interfaces to form a network of things. This modeling offers a broader perspective on the IoT application that can be employed to control the network. Because of the presumed heterogeneity of the IoT, the wiring is complicated at the code level. Hence, the need for higher abstractions free from the technical considerations to enable seamless wiring. The existing approaches for network modeling are disparate; some approaches target WSN (a network of many tiny sensors dedicated to collect data), others target the web of things(i.e., the IoT using existing web technologies) and others target a specific category of IoT applications such as smarthomes or TinyOS-based things. According to our literature review, we noticed a lack of a full-scale, documented, and open-source approach for modeling and controlling a network of things generically. Moreover, only a few approaches 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.

Table 1.1 presents a summary of the main existing MDE approaches for the IoT. The first column cites the reference paper. The second column corresponds to the scope of their modeling solution. The third column corresponds to its target usage, and the fourth column indicates whether it is open source. Many research studies focus either on creating a model of the behavior of the thing or adding controls for some specific usecases. The literature lacks a generic approach to model a network of things, constrain it and generate its code.

The Problem

By and large, the existing software engineering approaches in the literature still suffer from heterogeneity at low-level, interoperability issues, redundant tasks, difficulty to control a network, and overlapping concerns. The root cause of these issues is often difficult to capture during a traditional software engineering experience.

The Proposed Solution

We propose to dissect this problem using various MDE, primarily by separating the specification (using unified concepts at the model-level) from the implementation of the network, thus making these issues more visible, hence easier to capture and solve with an appropriate engineering tool.

Table 1.1 – Summary of the main existing MDE approaches for the IoT

Reference	**Design Scope**	**Target Usage**	**Source code**
[90]	Thing-level	Multi-platform Code Generation	Open Source
[93]	Thing-level	Multi-platform Code Generation	Open Source
[94]	Thing-level	Multi-platform Code Generation	Open Source
[95]	Thing-level	Sun SPOT [96] Code Generation	N/P
[103]	Network-level	OpenHab Configuration Generation	N/P
[110]	Network-level	Things Mashup	Open Source
[105]	Network-level	Modeling	N/P
[106]	Network-level	Simulation Java Framework	Open Source
[108]	Network-level	iSense [132] Code Generation	N/P
[114]	Cloud-level	IoT platforms APIs	N/P
[124] (*)	Control-level	ABAC and RBAC	Open source
[35]	Control-level	Security analysis and visualization	Open source
[122]	Control-level	Adding a monitor in the thing	Open source
[123]	Control-level	Smart health applications	N/P

(*) : Not based on MDE but decouples the specification from the implementation
ăN/P : Not Provided

What to expect?

The following chapters introduce a model-based software engineering methodology to design a network of (presumably heterogeneous) things. The proposed approach target static networks where things are known before deployment. We plan to cover dynamic networks, i.e., where thing may be added at runtime, in our future iteration. The present work complements the literature w.r.t. MDE and MDS for the IoT.

The underlying building blocks of the methodology are:

1. A DSL, to model a network of things (cf. Chapter 2).
2. A policy language, to control the network (cf. Chapter 3).
3. A code generation process based on model transformation, to wire the things and enforce the policies (cf. Chapter 4).

Chapter 5 presents an algorithm depicting the software engineering process that we advocate. It shows step by step, thanks to a Business Process Model and Notation (BPMN) diagram, the various responsibilities, and the process to design a network of things with our methodology. We apply this process to a case study in Section 5.3. Finally, we answer the research questions, wrap up the contributions and present the future work in Chapter 6.

Chapter 2

Modeling a Network of Things

The literature contains approaches to model a thing's behavior, yet it lacks a comprehensive modeling solution to specify a network of things and generate its concrete low-level code from its specification. This chapter presents the concepts for modeling a network of things, the theory (based on MDE), and the user interface (based on a DSL). It provides some means to *wire* things.Although the theory is reproducible with any other modeling formalism offering the possibility to create and use a metamodel, we tried to use some state-of-art MDE tools, namely ThingML [90], to model the behavior of a thing and Xtext [133], to create the syntax of the DSL. To illustrate these concepts, we show a few small examples throughout the chapter.

2.1 Reification of IoT concepts

Problem 1. *The heterogeneity creates a barrier for seamless communication between things.* ***The proposed solution*** *consists of reifying primitive networking concepts at the model level, to wire things irrespective of their programming language or protocol.*

The reification of primitive networking concepts takes place at the M2-Layer. We use these concepts in the M1-Layer to specify the network. Then, we move from M1-Layer to M0-Layer (i.e., the concrete) using automatic code generation, as depicted by the layers' architecture in Figure 1.3. The reification was guided by the Shannon information theory [134]. Shannon explains how communications occur between two parties, the sender and the receiver, in our case, the sending thing and the receiving thing.

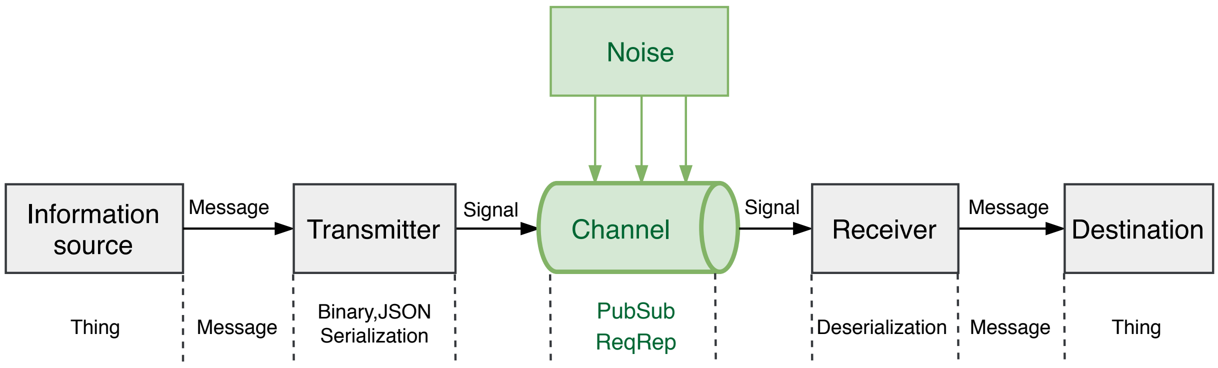


Figure 2.1 – The mapping of the information theory model concepts to the IoT concepts

Remark

Reification refers to the process of creating a concrete concept from something intangible/abstract. The low-level IoT concepts are challenging to unify due to heterogeneity. Their reification requires focusing on their commonalities necessary to wire things.

Figure 2.1 depicts a Shannon diagram of communication between two parties. Below each box, we provide the potential mappings to some IoT concepts. The things are the source and destination of the information. They encode a message on send using serialization and decode it on receive using deserialization. The message is sent or received via a channel (e.g., publish-subscribe, point-to-point).

Finding the right concepts that can unify heterogeneous low-level concepts is difficult. We focused on their commonalities and identified a few primitive relations. Still, if something more specific is needed, it can be tackled during the interpretation of the model (cf. Chapter 4). As we base our concepts on the information theory model, we must find these concepts on any thing regardless of its characteristics. We also aim at making these concepts readable to avoid the necessity of learning any new (time-consuming) skill.

Models allow avoiding the technical details, 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 in the metamodel. The process of code generation that allows moving from the M1-Layer to M0-Layer interprets this model depending on the target low-level programming language.

In the following sections, we discuss in detail these concepts. Within our methodology, we define the following responsibilities:

— **Thing Designer:** one responsible for writing the behavior of the thing.

— **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).

We provide a detailed BPMN diagram about the relationships between these responsibilities in Section 5.1. Our methodology's focus is on designing a network of things that is eventually enforcing a policy. As shown in the BPMN of Figure 5.1, the **Thing Designer** (upper row) provides the behavior of a thing in the form of a model. Then, the **Network Designer** (middle row) uses these models to create the network. Finally, if a policy is enforced (optional), the **Policy Designer** (bottom row) writes it and provides it to the **Network Designer**.

2.1.1 Things

We present in this section ThingML, authored by Harrand et al. [90], as it is the main requirement of our methodology. We presume that the model of a thing, called ThingML-Model (TH-Model), is specified using ThingML-DSL (TH-DSL) [1]. TH-DSL permits specifying a statechart-based behavior along with functions and properties that can be called within any state. Figure 2.2 depicts a simplified metamodel of TH-DSL and TH-Model. The central concept in this diagram is the concept of **Thing**. We can start reading this diagram from here. A **Thing** may posse a **Property**, a **Statechart**, a **Function**, a **Port** and a **Message**. While the diagram shows the big picture of what can be achieved with TH-DSL, the most useful concepts to our methodology are, from the one hand, that the **Statechart** may have a **State** with a **Transition**, from the other hand that this **Thing** may have a **Port** that accept a specific **Message**. TH-DSL provides a syntax based on the Xtext [2] grammar to write the model in a textual form.

1. https://github.com/TelluIoT/ThingML
2. https://www.eclipse.org/Xtext

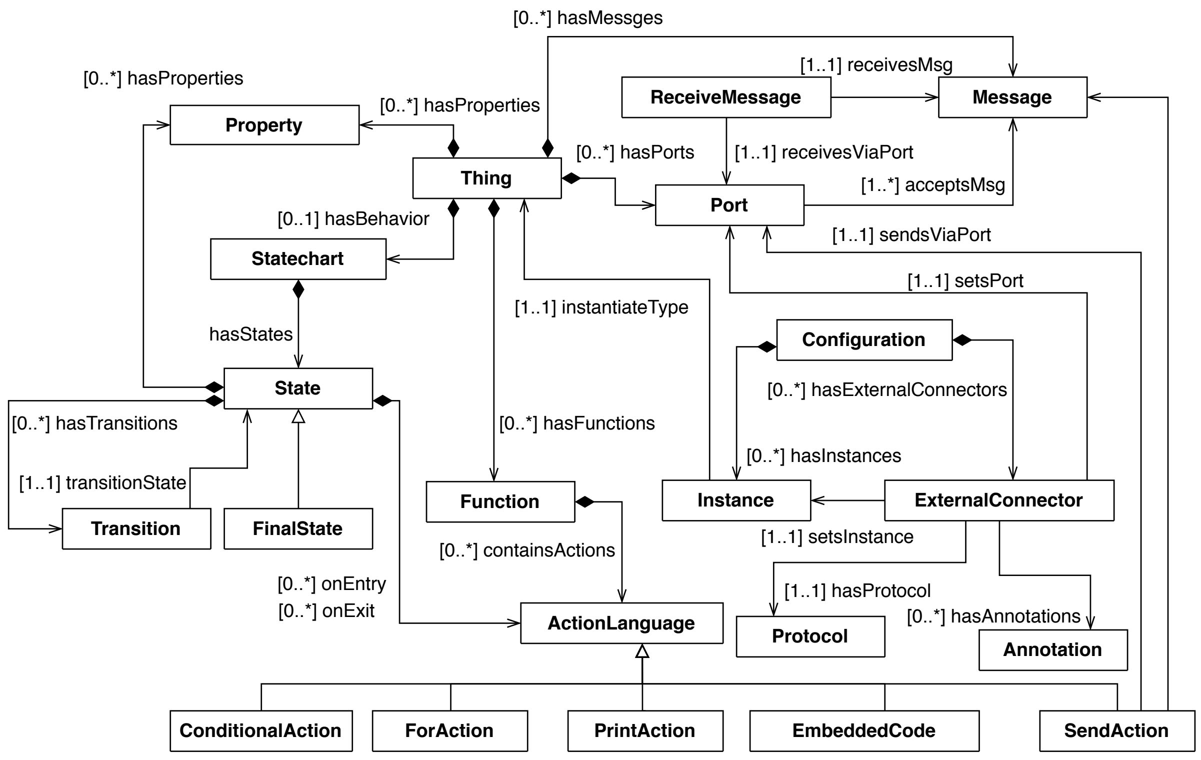


Figure 2.2 – A simplified version of the ThingML metamodel

Figure 2.3 depicts an example of the behavior of a temperature sensor based on a statechart and Listing 2.1 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 41. The port provides a means to route some data within the statechart to a specific internal address, identified by the port name. This statechart needs to be specified by a **Thing Designer**.

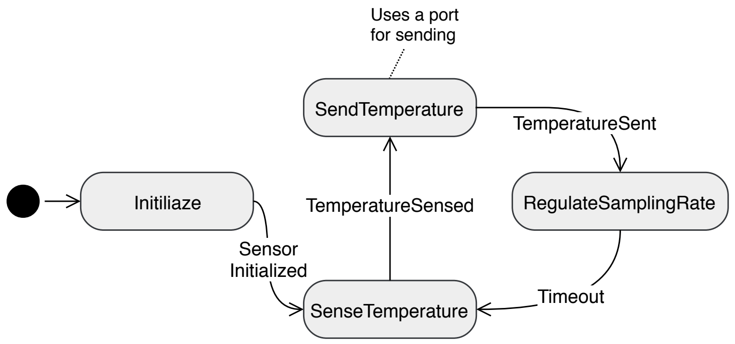


Figure 2.3 – Statechart-based behavior of a temperature sensor

```
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 // variable storing the current 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    provided port sendingTemperaturePort {
10     sends temperatureMessage // the sending port
11   }
12   function sense() do
13     'sensor.read11('&sensorPin&')' // embedding arduino code to read the temperature; &sensorPin&
           sets the value 8 in the low-level code
14     currentTemperature = 'sensor.temperature' // assiging the temperature value to the
           currentTemperature property
15   end
16   statechart temperatureSensorBehavior init initialize {
17     state initialize {
18       on entry do
```

```
            println "initialize"
            `sensor.begin();`
         end
         transition -> senseTemperature
      }
      state senseTemperature {
         on entry do
            println "senseTemperature"
            sense()
         end
         transition -> regulateSampling
      }
      state regulateSampling {
         on entry do
            println "regulateSampling"
            `delay(`&samplingRate&`)` // setting the sampling rate
         end
         transition -> sendTemperature
      }
      state sendTemperature {
         on entry do
            println "sendTemperature"
            sendingTemperaturePort!temperatureMessage(currentTemperature) // sending the current
                temperature via the sending port
         end
         transition -> senseTemperature
      }
   }
}
datatype UInt8<1> // Mapping the types UInt8 into its low−level equivalents
   @type_checker "Integer"
   @c_type "uint8_t"
   @java_type "byte"
   @js_type "byte"
```

Listing 2.1 – The behavior of the temperature sensor in ThingML-DSL; the syntax for embedding code is: ‘<EMBEDDED CODE>‘.

As shown in the metamodel of Figure 2.2, TH-DSL offers the concept of *external* *connector*. This concept enables us to wire the port with the outside by specifying the protocol and the serialization format. The 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).

We created a DSL called CyprIoT-DSL (CY-DSL), dedicated exclusively to networking; its metamodel is presented in Section 2.2.1. We show its underlying usage in Section 2.3. Listing 2.2 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. More details about roles in Section 2.1.3). It consists of a name as an identifier (i.e., TemperatureSensor) and the relative path in disk of the TH-Model (i.e., "temperatureSensor.thingml").

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 14 and 34 of Listing 2.1).

In summary, ThingML is useful for us to specify a thing's behavior in the form of 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.

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

Listing 2.2 – Declaration of a thing

2.1.2 Channels

As shown in Figures 2.1 and 2.4, the channel constitutes the medium between the interface of the sender and the receiver. 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.

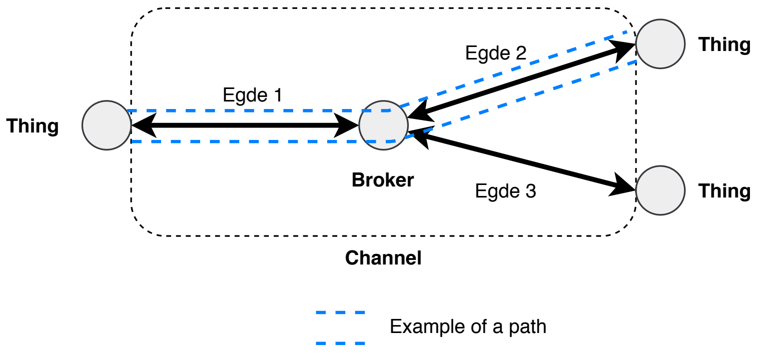


Figure 2.4 – A graph-based example of a network of things; the channel is the medium between things; The combination: Edge 1, Edge 2 is a path; the combination: Edge 1, Edge 3 is another path.

We performed a bottom-up analysis on IoT data protocols. Then, we factorized their commonalities in the concept of **channel** (e.g., the ability to route a message via a distinct path is a commonality). The Enterprise Integration Patterns book [138] references ten message channels on message-oriented applications; namely Point-to-Point Channel, Publish-Subscribe Channel, Datatype Channel, Invalid Message Channel, Dead Letter Channel, Guaranteed Delivery, Channel Adapter, Messaging Bridge, Message Bus. We focus our study on two largely used types of channels in the IoT: Publish-Subscribe and Point-to-Point. The former type is often used when things (e.g., MQTT-based thing) must communicate in a decoupled manner without the need to know each other [137]. These things only need to know the information about the broker that acts as an intermediary between them. The latter type is used when things are accessible via a public interface, such as an IP address or a Uniform Resource Locator (URL) or visible in a local network (e.g., Zigbee-based thing), so that another thing can reach it.

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

Listing 2.3 – Declaring channel, path and fork (Line 1 to 6)

The concept of **channel** decouples the communication of things from their programming languages, thus enabling seamless networking at the model-level using only abstract concepts. Listing 2.3 shows an example of a **channel**, containing two paths, where the second is a fork of the first, i.e., the path temperaturePath is contained in the path indoor. 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 [138]. It is inspired from existing protocols, for instance, in MQTT [139] a message is exchanged via a topic, while in HTTP via a URL. MQTT is a publish-subscribe protocol that enables many-to-many communication between things using hierarchical topics. Both topics and URLs can be unified under the concept of **path**.

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 2.3, a **path** consists of an identifier (temperatureTopic), a declaration of the accepted message (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 must be accepted (i.e., understood) by the sending port, the receiving port and the path that links them, to reach its destination correctly. For instance, the sendingTemperaturePort is compatible with the path temperatureTopic as both accepts the message temperatureMessage.

2.1.3 Users and Roles

We also reify the concepts of user and role. Lines 2-3 of Listing 2.4 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 (cf. Section 3.2.5). 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.

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. Chapter 3 for more details). Declaring a **role** has a similar syntax than a **user**. Lines 5-6 of Listing 2.4 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 2.2. Also, a thing can have simultaneously several roles.

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

Listing 2.4 – Declaration of users and roles

2.1.4 Network

First, the Network Designer declares things, channels, roles, and users, then, uses them to specify the **network**. The things and channels must be instantiated to be used inside the **network**. Several instances may be derived from the same declaration. The concept of 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 2.5, a **network** has an identifier (mySimpleNetwork) and a **domain** (org.atlanmod.mynetwork). A **domain** is supposed 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. 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**); this information provides the code generator with the target protocol to be implemented in the low-level code. Finally, we can specify to **bind** a port of an **instance** of thing to one or multiple paths of any of the available instances of channels (e.g., Line 10 of Listing 2.5).

A **network** can also enforce a policy. Multiple policies can be enforced. For instance, in Listing 2.5, both roleBasedPolicy as well as smartpolicy are enforced. Policies and control strategies are discussed in detail in the next chapters.

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 4.2.1).

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

Listing 2.5 – Specification of a network; sending (=>) and receiving (<=) messages via a path

2.1.5 Forwarding

Usually, in the IoT, because of limited resources, a thing may need to pass through a more powerful intermediary thing before reaching its final destination. 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.

The concept of forwarding enables to forward an *existing binding* (i.e., **bind**) to another **path**. For instance, in Line 14 of Listing 4.1, we **forward** the temperature received via **ZIGBEE** by myGW to temperatureTopic, a path of mqttBroker that is using **MQTT** as a protocol. We assume in this case 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. The code generator handles this heterogeneity through an automatic process (cf. Chapter 4).

In fact, due to variation in sizes of things, an interoperability issue is a common trait of connectivity of things in the IoT [140]. The ability to forward an existing binding at the model-level allows navigating freely between the various ranges of things. The implementation of these forwarding is kept for the code generation phase. During code generation, some specific procedures interpret these model-based forwarding and reproduce their equivalent in the target programming language of the involved things (cf. Section 4.2.1).

```
network mySimpleNetwork {
  domain org.atlanmod.mynetwork
  enforce roleBasedPolicy, smartpolicy
  // Instance of things
  instance myRD : RemoteDisplay platform JAVASCRIPT owner Bob
  ...
  // Instance of a channel
  instance zigbeeChannel:brokerChannel protocol ZIGBEE
  instance mqttBroker:brokerChannel protocol MQTT(server="mqtt.atlanmod.org:1883")
  ...
  // Binding : We add an identifier (i.e., tempBindGw) to the bind
  bind tempBindGw : myGW.TempDataPortRec <= zigbeeChannel{temperaturePath}
  // Forwarding
  forward tempBindGw to mqttBroker{temperatureTopic}
  // Binding : Receiving the forwarded temperature message
  bind myRD.receivingTemperaturePort <= mqttBroker{temperatureTopic}
}
```

Listing 2.6 – Forwarding of an existing binding

2.2 Every "thing" is a model

Problem 2. *The reusability of software artifacts by machine is crucial for large scale networks.* ***The proposed solution*** *is to use models, so that the software engineering cycle can benefit from MDE tools.*

The metamodel (M2-Layer) is a defining piece of our methodology. It contains the networking concepts and how these relate to each other. The metamodel allows us to create the network model, i.e., the CyprIoT-Model (CY-Model) (M2-Layer).

We present in the next section an EMF-based metamodel for our networking language. Everything is a model in the lower layer, the behavior of a thing is a model (i.e., TH-Model) and the network that is wiring them is also a model (i.e., CY-Model). These models naturally represent a real-world IoT application (M0-Layer).

2.2.1 Metamodel

The proposed metamodel consists of two parts: the networking part and the policy part (cf. Chapter 3). Figure 2.5 depicts the metamodel for the networking part. It shows how the reified concepts depend on each other. This formalism provides us with the language to express our CY-Model. Also, to position our work w.r.t. state of the art, we highlighted in white and bold the concepts of ThingML. Otherwise, we created a link with the policy concepts (Light Gray) that will be the next chapter's subject.

In Figure 2.5, the network class possesses the highest number of relationships, making it a central concept of our formalism. The network can have multiple instances, either instances of things or instances of channels. The former enables instantiating TH-Models and using its constituents (e.g., Port, Message, Property). In contrast, the latter allows instantiating a channel by setting the communication protocol (e.g., MQTT, HTTP, COAP, ZIGBEE) and using its paths. The network enables us to bind the ports of these TH-Models to any path of the instances of channels.

Typically, two ports can communicate if they are compatible, i.e., they accept both the same type of message; otherwise, this results in an invalid model. For instance, in Listing 2.5 as the port `TempDataPortRec` of `myGW` consumes the messages sent by `TempDataPortSend` of `myTempSensor`, we presume that both things have compatible ports, otherwise an error will be reported in the editor (cf. Section 2.3).

In summary, by using this metamodel, we end up with two kinds of models, (1) TH-Models, containing the behavior of each thing, and (2) the network model (i.e.,

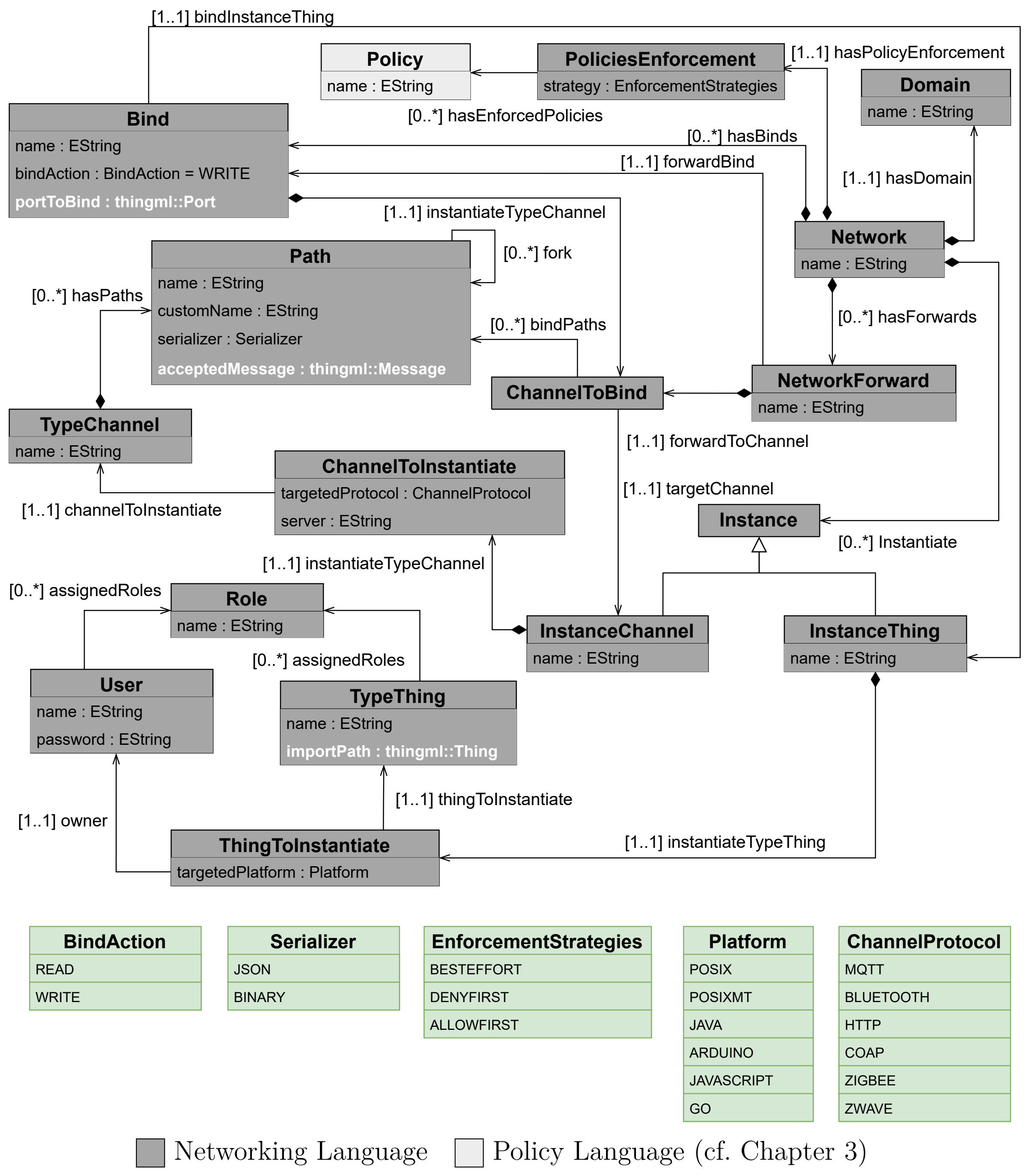


Figure 2.5 – Networking language metamodel; boxes in green represent enumerations

CY-Model), including these TH-Models and specifying how they interact between each other via their ports.

2.2.2 Network Model

The network model (i.e., CY-Model) is the unique primary input in our methodology. It conforms to the metamodel (cf. Figure 2.5) and can serve to generate various outputs depending on the use. We focus on code generation (cf. Chapter 4).

The network model provides stakeholders (e.g., software engineer, business manager, security officer) a common artifact to work together instead of working in silos. It enables to work based on a unified view of the network in place of dealing with different artifacts (e.g., written document, databases, email, drawing). From this view, we can derive various valuable artifacts using an adequate MDE tool or a manual procedure.

A unique input offers the opportunity for tracing every step of the software engineering process. Using some MDE tools, we could follow the evolution of its attributes throughout the code generation process, from the input to the final artifact. Moreover, it helps to identify the parts that can be automated by an existing MDE tool, thus, benefiting from a rigorous trace to determine the parts that have to be done manually or need the intervention of an expert.

The Network Designer fills each attribute of the metamodel with a concrete value, depending on the intended scenario. This value corresponds to a real-world element. In this particular example, we presume that Bob owns a temperature sensor in the real world; `Bob` is the value for the user, and the instance represents the temperature sensor. The interpretation of such information depends on the needs of the engineer. For instance, it can be interpreted to write access control rules w.r.t. Bob, draw a diagram, or document the application in a plain text file.

2.2.3 Usability of the Model

The model relies on rigorous formalism, as shown in Section 2.2.1. Thus, to be valid, a set of rules defined by the metamodel (i.e., the model conforms to the metamodel) and CY-DSL (i.e., checking the semantic—cf. Section 2.3.1) has to be respected. These rules enable us to check syntactically and semantically the correctness of the network at the design stage.

Also, the model can be used for various other purposes and we aim to use it as an input for any automatable software engineering task for our IoT application. To highlight few examples, we can cite two of its primary uses: a) Model Transformation [141], consisting of transforming the behavior of the TH-Models according to the specification of the network in the CY-Model, or to generate some text such as a user manual (cf. Chapter 4) and b) Model Views [142], which enables to explore various facets of the network model, such as a visual report of its resistance to security threats [35]. We focus on the present thesis on model transformation. Further, we detail how we use it for control and code generation purposes.

Model transformation allows generating automatically the network artifacts from a rigorous and valid network model. This process reduces the quantity of manually written code, thus limiting the surface of human-induced bugs.

2.3 Domain-Specific Language

Problem 3. *The unfamiliarity of model-based approaches to software engineers is an obstacle to its adoption.* ***The proposed solution*** *is a DSL based on a readable syntax.*

We created a DSL based on the metamodel of Figure 2.5, named CY-DSL[3] [143]. It is also based on the Xtext grammar, and is therefore a textual representation of an EMF model. 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. The full grammar can be found in Appendix A.

CY-DSL relies on the same concepts shown in the previous section, but in a textual form. Hence, while the model represents the back-end, CY-DSL represents its front-end. It is a practical option to create models from an Integrated Development Environment (IDE).

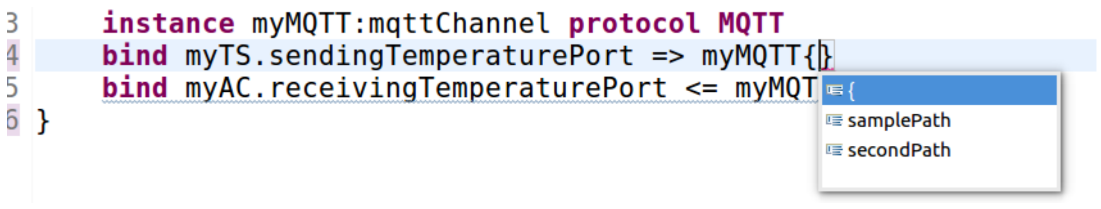
2.3.1 Integrated Development Environment

The CY-DSL benefits from the advantages of an IDE (e.g., syntax highlighting, error reporting, auto-completion—cf. Figure 2.6) [144] same as popular programming languages. An IDE consists of a set of tools (e.g., editor, debugger, refactoring) to increase the productivity of software engineers. By default, CY-DSL is integrated into Eclipse IDE.

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

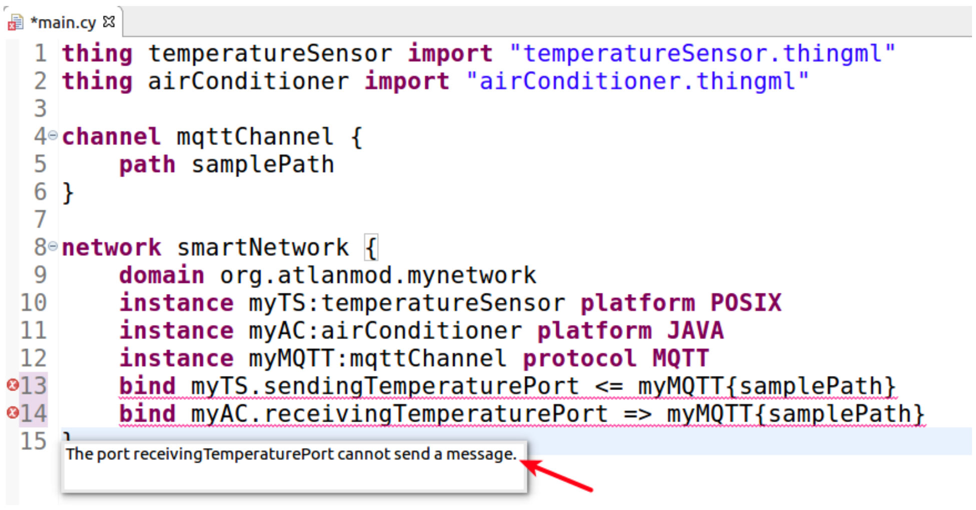
CY-DSL is backed by some syntactic and semantic validators to ensure the correctness of the network. Indeed, these validators help prevent some inconsistencies (e.g., preventing a communication using incompatible message formats, or incompatible port and path) by reporting them instantly in the form of errors, warnings, or notices as shown in Figures 2.7. This mechanism saves time by helping to anticipate early runtime bugs [145, 146].

One can still write a CY-DSL file using any editor (e.g., vim, nano, notepad++), i.e., without relying necessarily on Eclipse. We also offer a command-line interface that plays the same role. It takes a CY-DSL file as input and shows the inconsistencies after execution.



```
    instance myMQTT:mqttChannel protocol MQTT
    bind myTS.sendingTemperaturePort => myMQTT{}
    bind myAC.receivingTemperaturePort <= myMQT
}
```

Figure 2.6 – An auto-completion example to suggest the possible paths



```
thing temperatureSensor import "temperatureSensor.thingml"
thing airConditioner import "airConditioner.thingml"

channel mqttChannel {
    path samplePath
}

network smartNetwork {
    domain org.atlanmod.mynetwork
    instance myTS:temperatureSensor platform POSIX
    instance myAC:airConditioner platform JAVA
    instance myMQTT:mqttChannel protocol MQTT
    bind myTS.sendingTemperaturePort <= myMQTT{samplePath}
    bind myAC.receivingTemperaturePort => myMQTT{samplePath}
}
```

The port receivingTemperaturePort cannot send a message.

Figure 2.7 – The early detection of inconsistencies in the editor; the usecase of an incompatible port and path

2.3.2 Readability and Maintainability

Readability corresponds to the degree whereby a piece of code is easy or difficult to grasp by a software engineer or, in some cases, a machine. Multiple factors can influence readability (e.g., keywords, line length, indentation, space). CY-DSL is a declarative language, making it easier to learn. The syntax in Appendix A, shows that we use intuitive and comprehensible keywords (e.g., **thing**, **channel**, **path**, **instance**, **policy**, **rule**).

The syntax shows the intended scenario. This point is particularly helpful in improving communication between stakeholders. Indeed, communication skills are crucial in software engineering [147], a high-level textual description of the network provides a central source of information, thus offering a unified platform to discuss various aspects of the application.

The model also provides a broader picture of the network, leading to a more participatory decision making w.r.t. its features. This comprehensive picture allows stakeholders to move easily towards a common goal [148]. Indeed, each stakeholder can discuss a particular aspect of the application before moving to more advanced steps, making the process quickly iterative. It is, therefore, convenient to maintain the network. For instance, one can keep several versions of the CY-Model depending on the discussions between stakeholders.

2.4 Summary

By reifying these network concepts, we offer software engineers a unified environment to specify a network of heterogeneous things. The CY-DSL provides a way to specify the model in a textual form. It can also control its syntax and semantics, resulting in a correct model and fewer bugs in the network at runtime. Figure 2.8 presents a straightforward illustration to summarize what has been achieved in this chapter. As shown in this figure, at this stage, our methodology consists essentially of a CY-DSL file, backed by the CY-Model. The CY-Model imports the TH-Models to use them in the specification of the network. Further, we will update this figure to position each contribution, w.r.t. the overall methodology.

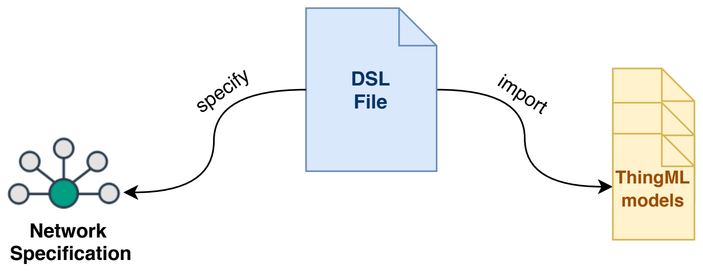


Figure 2.8 – Modeling the network: the first step of the methodology

Chapter 3

Controlling a Network of Things

The previous chapter shows how to create a model of the network based on unified concepts. This chapter 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 to either restrict communication or trigger actions according to some conditions. We will discuss solely the controls' specification in this chapter; its enforcement is developed in the next one.

3.1 Model-based Control

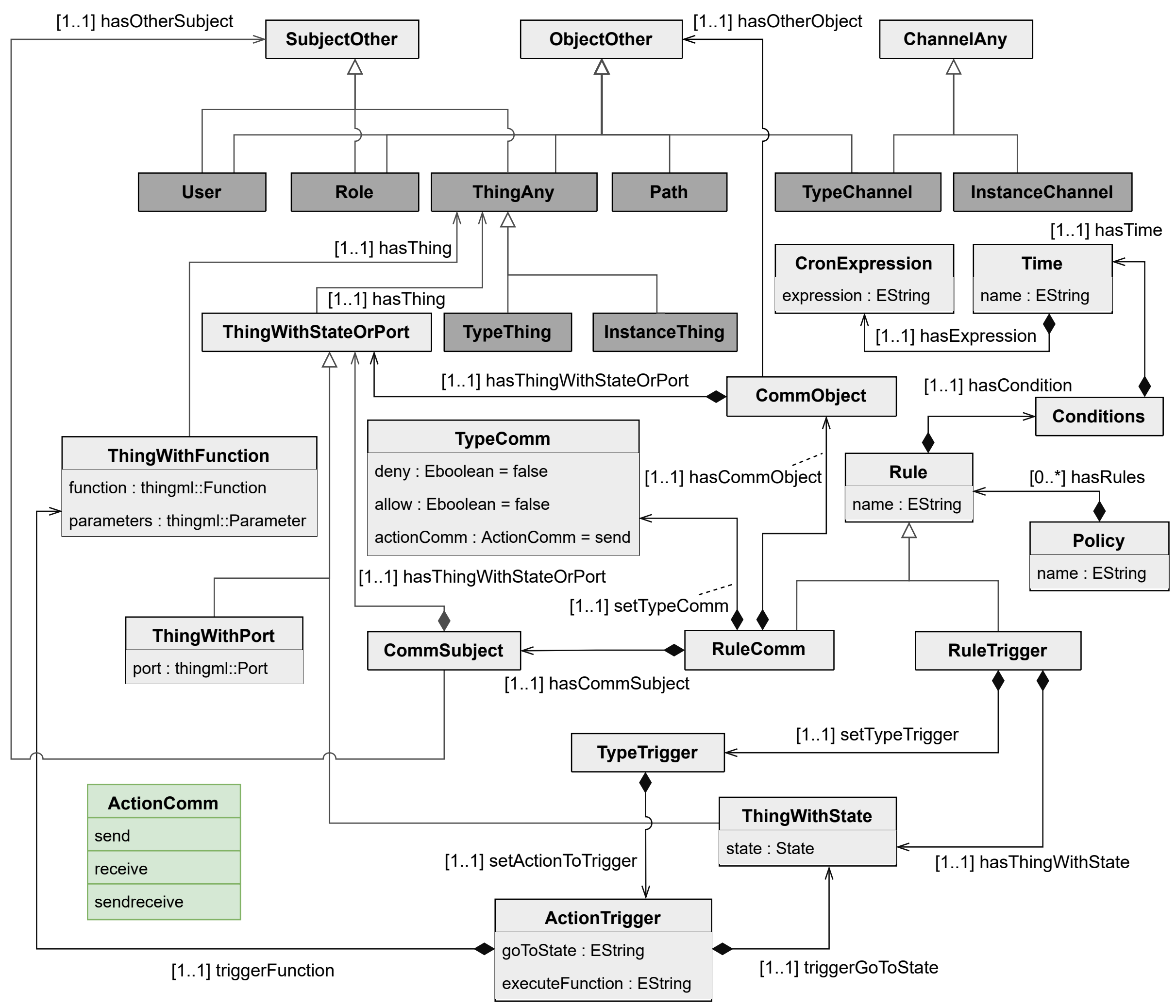
Problem 4. *The heterogeneity creates a barrier for crosscutting control.* ***The proposed solution*** *consists of specifying the controls at the model-level where the network concepts are unified.*

The model offers a unified description of the network. The control allows for a more specific description by adding constraints on its behavior. Specifying such global constraints is hard when dealing with heterogeneous low-level code.

Figure 3.1 depicts the metamodel of the control abstractions. These abstractions also have a textual version, based on Xtext (cf. Appendix B for the full grammar). It is the continuation of the network metamodel of Figure 2.5. The network enforces policies that contain rules. These rules have a specific structure and apply actions based on the elements of the specified network (e.g., thing, channel, user). This section describes the rule-based control system we offer.

3.1.1 Policy

Generally speaking, a policy can serve various purposes [149] (e.g., communication control, administrative goal). In this thesis, we focus on two main aspects, communication control (cf. Section 3.2) and smart scenarios (cf. Section 3.3). A policy ensures that the



▨ Networking Language (cf. Chapter 2) ☐ Policy Language

Figure 3.1 – Policy language metamodel; continuation of Figure 2.5; boxes in green represent enumerations.

IoT application is behaving as expected from a stakeholder's perspective, such as 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 2.5, one or more policies can be enforced in the network; in this particular example, the network enforces roleBasedPolicy and smartpolicy.

Listing 3.1 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 2.5).

The policy's specification is readable and relieved from the low-level technical details. It needs to be written by a **Policy Designer**. The Network Designer chooses what policy to enforce. The enforcement inside the low-level code is the concern of the code generator (cf. Chapter 4), assumed by experts. Experts are responsible for developing the enforcement strategies in the code generator and mapping the model's abstract concepts into their concrete equivalents at a low-level.

3.1.2 Rule

A rule specifies the conditions necessary for an action on an object. Listing 3.2 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. The ability to use model-based abstractions permits the dissociation of the network's safety from its concrete implementation (source of heterogeneity).

```
policy smartPolicy {
  rule myTempSensor—>state:isLow trigger:goToState myAC—>state:isOn
  rule myTempSensor—>state:isLow trigger:executeFunction myAC—>function:setTemperature(25)
}
```

Listing 3.1 – An example of a policy (Discussed in Section 3.3.3)

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

Listing 3.2 – Rule syntax

Remark

An entity is any distinct concept within the network that can be uniquely identified, such as a thing, a user, a role, a path, or a channel. In the diagrams of Figures 2.5 and 3.1, the entities are the class having the attribute **name**.

Table 3.1 depicts the possible entities of a rule for each of its parts and how they can be combined. The rule in Listing 3.2 has to be used alongside this table. The first column contains the subject's supported entities and the fourth column, the object's entities.

Table 3.1 – The combination of rule entities.

Subject	**Action Type**	**Action**	**Object**	**Control Type**
Port ăInstance of thing Thing User ăRole	Permission	Send ăReceive ăSend-Receive	Port ăInstance of thing Thing User ăRole Path Channel	Communication
State	Trigger	goToState executeFunction	State Function	Thing Behavior

3.1.3 Control Types

The rule structure previously mentioned offers two types of control, namely a communication control and a 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 [150]:** 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.

— **Thing Behavior Control [140]:** 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 so that it satisfies the intent of the rule. 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.

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

3.2 Communication Control Rules

Problem 5. *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.*

The network encompasses a communication flow analogous to the representation of Figure 2.1. Communication control rules specify the constraints on this flowmany 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 message that can be sent or received by the thing s/he owns.

By default, we presume that all the communications are denied 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.

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.

3.2.1 Structure

From the rule structure of Listing 3.2, we inherit the structure of Listing 3.3 for the communication control rules. We use the keywords **deny** or **allow** to control the actions of **send**, **receive** or **send–receive**. Further, we detail all the possibilities of such rule.

```
rule <Subject> <deny | allow>:<send | receive | send−receive> <Object>
```

Listing 3.3 – Communication control rule syntax

3.2.2 Potential Applications

The communication control type has various applications in the real world, such as access control [151], content-based control [152] or privacy control. For instance, ABAC and RBAC are two established access control paradigms. The specification and implementation of these paradigms vary. Our communication control approach is singular and aims to be generic, yet, we partially cover the specification of ABAC and RBAC from an access control perspective. 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.

This thesis focuses on the software engineering aspects of the IoT; however, we believe that extending our methodology to cover the specification and implementation of ABAC and RBAC is a promising avenue of research. All the more that access control is an important milestone for the IoT [153, 154]. The separation of the specification from the implementation should ease the enforcement of such access control models.

3.2.3 Ports Communication Control

The port decides to accept a new message, or prevent it from leaving. It is an essential notion from a communication control perspective. It is one of the few checkpoints of the network, where we can apply a control concretely, and the most fine-grained communication control we offer. A rule can specify that two ports are denied or allowed to send or receive messages. We separate the specification of control w.r.t. sending and receiving. A port may enable sending but deny receiving, and vice versa.

Listing 3.4 shows an example of such a rule. It denies the port TempDataPortSend of the instance myTempSensor from sending messages to the port TempDataPortRec of the instance myGW. Depending on the enforcement strategy and the network specification, there may be slightly different interpretations of this rule during code generation. We provide details w.r.t. the enforcement strategies in Section 4.3. Nevertheless, generally, we assume that whenever the port TempDataPortSend tries to send a message, where TempDataPortRec can potentially receive it, sending the message should be denied from sending at TempDataPortSend level by the code generator.

The control at the port level matters because it offers the ability to deny or allow only a specific interaction type. This control may be useful to define what facet of thing is accessible. For instance, an untrusted thing may be denied to communicate with a port that handles authentication.

```
rule myTempSensor—>port:TempDataPortSend deny:send myGW—>port:TempDataPortRec
```

Listing 3.4 – Denying communication between ports

3.2.4 Things Communication Control

There are two levels of granularity on the thing level: the control at the instance level, and at the type level. As a reminder, a type of thing must be instantiated to be used in the network.

Instances of Things Communication Control

Controlling communication between instances of things is the second level of granularity we offer. It applies to all ports of the instance.

Listing 3.5 shows an example of such a rule. It denies the instance named myTempSensor from sending messages to the myGW regardless of the sending port. This rule assumes that whenever myTempSensor tries to send a message, where myGW can potentially receive it, sending this message should be denied from leaving myTempSensor.

The control at this level is useful to regulate the interaction of the *smallest unit* (i.e., an instance of thing) of the network with any other entity.

```
rule myTempSensor deny:send myGW
```

Listing 3.5 – Denying communication between instances of things

Thing Types Communication Control

Controlling communication between types of things is the third level of granularity of control. It applies to all the instances of the type. Listing 3.6 shows an example of such rule. This rule denies all instances of type TemperatureSensor from sending messages to any instance of type Gateway.

```
  rule TemperatureSensor deny:send Gateway
```

Listing 3.6 – Denying communication between thing types

The control on the thing type-level enables to set the communication rules globally, before the configuration of the network. This control may be useful to prevent some im-proper network configurations or conflicts of interest. For instance, for privacy reasons, we can restrict any sensitive type of thing, such as a *Private Security Camera* to communicate with any type presumably communicating publicly, such as a *Web Server* or URL (equivalent to a **path**).

3.2.5 Users Communication Control

A user can own one or more instances of things. The user-based control applies to all the instances of things the user owns. Listing 3.7 shows an example of such a rule. It denies the instances owned by the user Bob from sending messages to instances owned by the user Alice. The control at this level enables a person-oriented regulation.

```
rule Bob deny:send Alice
```

Listing 3.7 – Denying communication between users

3.2.6 Communication Control Based on Roles

A role can be assigned to multiple things. One major difference with a user from a control perspective is that a thing can have multiple roles. All the permissions given to a role will be applied to all things with that role. This concept is useful to create hierarchies of things with specific responsibilities. Listing 3.8 shows an example of such rule. It denies the instances with the role secretThingRole from sending messages to instances with the role publicThingRole.

```
rule secretThingRole deny:send publicThingRole
```

Listing 3.8 – Denying communication between roles

3.2.7 Combinations

The previous sections showed communication rules based on a subject and an object of the same type (i.e., both are of type thing, or user or role). In this section, we show how we can combine different types of subjects and objects.

Combinations of entities

Combining the entities cited in the previous sections offers a more flexible way to constrain the network. Indeed, we can control communication between two different entities. We can specify a rule controlling whether a port can communicate with an instance, a type of instance, a user, or a role. The action applies respectively to all ports of the instance, all instances of the type, all instances owned by the user, and all instances with that role.

The same principle is also valid for the combination of other entities. Another example, we can specify a rule controlling whether an instance can communicate with a port, a type of instance, a user or a role; the action applies respectively to the port, all instances of the type, all instances owned by the user, and all instances with that role.

The subject and object can be of different granularity; the rule applies to all elements of these two levels. For instance, if the subject is a port, the object is a user, and the action is to deny sending, then the control will deny the port from sending to any port of any thing owned by this user. Listing 3.9 shows an example of such rule. All the instances of type Gateway are denied sending to any instance that has been assigned the role publicThingRole.

```
rule Gateway deny:send publicThingRole
```

Listing 3.9 – Denying communication between a type of thing and a role

Combining entities with channels and paths

Unlike the object, the subject can also be of type channel or path, as shown in Tables 3.1. This syntax permits to control the communication between the entities and the channel. A channel contains a set of paths. A path is another essential notion from a control perspective; it is a checkpoint that enforces a control concretely. Indeed, depending on the policy, it can accept or refuse to transmit a message from one end to another.

On the one hand, we can specify a rule with an entity as a subject and a path as an object, resulting in applying the action between this entity and this path. On the other hand, we can also specify a channel as an object, resulting in applying the action to all its paths. Listing 3.10 shows an example of such a rule. All the instances of type Gateway are denied to send to the path myRestrictedPath (Line 1) and the channel myRestrictedChannel (Line 2).

```
1  rule Gateway deny:send path:myRestrictedPath
2  rule Gateway deny:send channel:myRestrictedChannel
```

Listing 3.10 – Denying communication between a type of thing and a path or channel

All the rules presented so far aim to regulate the communication flow. This type of control is useful to set the boundaries between the elements of the network. These boundaries may prevent some unexpected behaviors on the network, such a data leakage or a breach of trust.

3.3 Smart Rules

Problem 6. *The interoperability of things to achieve smart scenarios suffers from heterogeneous concepts at a low-level.* ***The proposed solution*** *consists of specifying these smart scenarios at the model-level, and implementing them by a code generator using a dedicated procedure.*

The previous section shows how we can specify the control of communication flow; in this section, we present how we can control the network's behavior according to contextual factors, such as the state of things and the 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 [54]. The difficulty to implement smart scenarios is often caused by heterogeneity of things at a low-level. In our methodology, we avoid this heterogeneity at design time by relying on unified concepts at the model-level and delegate it to the automatic code generation process.

3.3.1 Structure

From the rule structure of Listing 3.2, we inherit the structure of Listing 3.11 for smart rules. We use the keyword **trigger** to activate two actions: **goToState** or **executeFunction**. The former triggers a thing to go to a particular state, and the latter triggers a function within a thing.

```
rule <thingID>−>state:<stateID> trigger:<goToState | executeFunction>
                                   <thingID>−><state: | function:><stateID | functionID>
```

Listing 3.11 – Smart rule syntax

3.3.2 Potential Applications

By lacking a common representation of the behavior at a low-level with traditional software engineering, it is challenging 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 interoperability of these heterogeneous things, that 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 specify that when a thing t_x is on the state s_i, the thing t_y must be in 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.

3.3.3 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 3.12 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 (sequence of instructions), there are cases where a function must be executed depending on the state of another thing. For instance, the rule in Line 3 specifies that the state isHigh of myTempSensor, triggers the function setTemperature(25) of mySAC (thus, setting the temperature to 25°C when it is warm).

3.3.4 Temporal factors

The ability to depend on the physical time is one of the defining features of IoT applications [42]. For this reason, a rule can optionally require the satisfaction of some temporal factors to be enforced. 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

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

Listing 3.12 – Go to a state (Line 2) and execute a function (Line 3) according to another thing's state

triggering an action only at a specific time of the day.

As shown in Listing 3.13, the syntax consists of adding the keyword **when** and specifying either a specific time or a period using the keyword **time**:. Listing 3.14 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.

```
rule <thingID>−>state:<stateID> trigger:<goToState | executeFunction>
                                  <thingID>−><state: | function:><stateID | functionID>
                                     when time:<start>−<end>
```

Listing 3.13 – Smart rule syntax including the temporal factors; for readability reasons, this syntax is divided in three lines

The specification of these temporal factors is straightforward, but their implementation may be difficult in a decoupled system as there is no standard way to define the physical time (cf. Chapter 4). Logical time is unsuitable in this case, as it is not a problem of ordering. Instead, things depend on the physical world, incidentally one fundamental promise of the IoT.

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

3.4 Conflict detection and resolution

Problem 7. *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.

We enable the detection of most conflicts between any rule in the long-run. So far, we only offer some conflict detection and resolution mechanisms w.r.t. the communication 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 and ask the software engineer to resolve them.

3.4.1 Early Detection

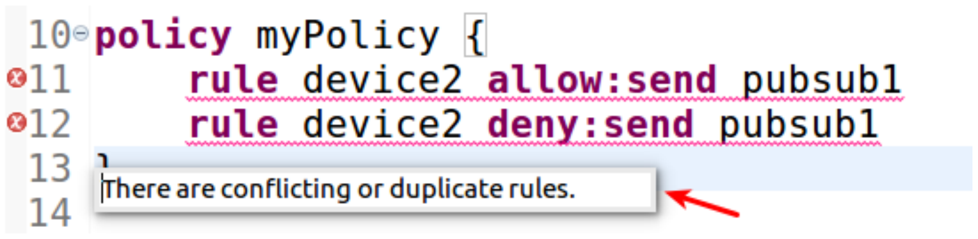
The conflicts are displayed in real-time in Eclipse IDE (cf. Example in Figure 3.2). The model is deemed invalid until the conflict is resolved. This mechanism prevents the engineer from introducing bugs by detecting inconsistencies in the model earlier. Figures 3.2 and 3.3 show examples of the errors that may be displayed. Figure 3.2 reports an inconsistency of the model due to two opposite rules; namely, the first rule allows device2 to send to pubsub1, while the second states the opposite. Figure 3.3 depicts two rules that specifies opposite constraints at different levels of granularity; namely, the first rule deny temperatureSensor from sending to the actuator airConditioner, while the second allows temperatureSensor to send to actuators.

3.4.2 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 their scalability, did not receive any detailed treatment and will be addressed as a future work. These algorithms are executed in the editor in

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

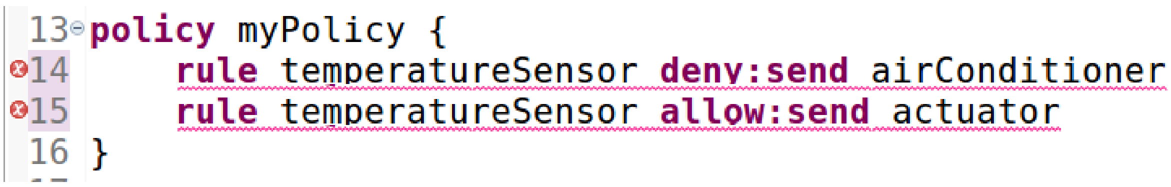
Listing 3.14 – Go to a state (Line 2) and execute a function (Line 3) according to another thing



```
policy myPolicy {
    rule device2 allow:send pubsub1
    rule device2 deny:send pubsub1
}
```

There are conflicting or duplicate rules.

Figure 3.2 – A conflict detection error in the editor of opposite rules



```
policy myPolicy {
    rule temperatureSensor deny:send airConditioner
    rule temperatureSensor allow:send actuator
}
```

Figure 3.3 – A conflict detection error in the editor between coarse and fine grained rules; *airConditioner* has been assigned the role *actuator* earlier.

real-time. When a conflict is detected, an error is shown, 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 implemented three algorithms: **(Algorithm 1)** Detection of opposite rules **(Algorithm 2)** Detection of a conflict between a group (i.e., a role or a user) and a thing and **(Algorithm 3)** Detection of a conflict between a role and a user.

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

Algorithm 1: Detecting opposite rules

Algorithm 1 is a basic procedure detecting opposite rules. Two rules are opposite if they enforce opposite actions for the same subject and object. The algorithm returns the collection of opposite rules, w.r.t. an input rule (the rule having the focus in the editor).

It iterates over all rules to check whether the subject and the object are equal on the one hand and whether the actions are opposed, i.e., when a rule denies an action (e.g., **deny**:**send**) the other allows it (e.g., **allow**:**send**) and vice versa. If so, it adds the rule to the *Contradictory* collection and returns it.

```
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 $Conflicting$;
```

Algorithm 2: Detecting a conflict between a thing and a group

Algorithm 2 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.

```
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 $Conflicting$;
```

Algorithm 3: Detecting conflict between two groups

Algorithm 3 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 the actions are opposed. If these two conditions are met, it adds the rule to the *Conflicting* collection and returns it.

3.4.3 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 case of conflict during code generation. The software engineer must specify the resolution strategy for each enforced policy. We propose a few resolution strategies, namely: Best-Effort, Deny-First, Allow-First. We discuss some of these strategies in the next chapter (cf. Section 4.3.1). Line 3 of Listing 3.15 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 3.15 – Specification of the resolution strategy

3.5 Summary

A model-based control allows for a more specific description of the network. We offer two main types of rules, communication control rules, and smart rules. While the former controls the communication flow, the latter provides means to specify smart scenarios based on behavioral and temporal factors. A mechanism of conflict detection and resolution is proposed to limit bugs as early as possible. Figure 3.4 updates the global view of our methodology with what has been achieved in this chapter. A policy is now enforced in the specification of the network.

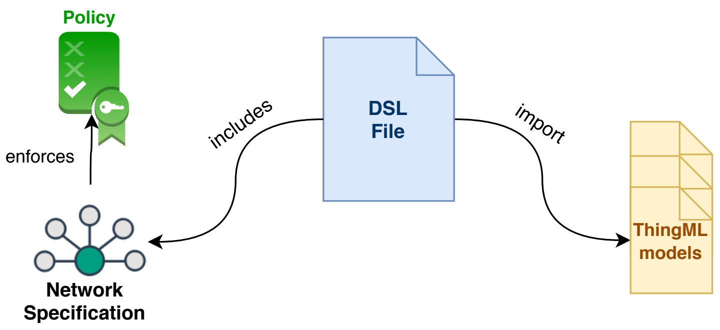


Figure 3.4 – Controlling the network: the second step of the methodology

Chapter 4

Generation of the Network Artifacts

So far, we saw how to write a specification of a network along with the policy. This chapter presents the code generator that generates the low-level code implementing the network and the policies, both written in CY-DSL. We explore its main building blocks, i.e., the procedures for wiring things, enforcing policies, generating textual artifacts, and extending the overall process with plugins.

4.1 Code Generator

CY-DSL provides the networking and the policy language and serves to express with high-level abstractions the CY-Model. The next step consists of interpreting this model to produce the application artifacts. We describe the functioning of the code generator. Subsection 4.1.1 presents its core architecture, Subsection 4.1.2 explains how it loads the models and Subsection 4.1.3 shows how it can be extended.

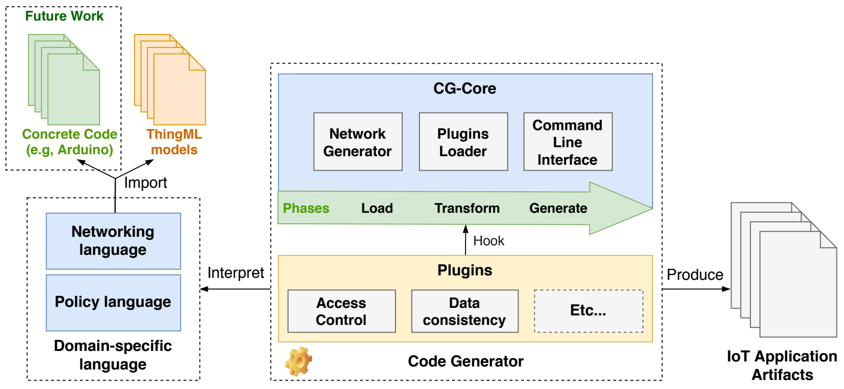


Figure 4.1 – The core components of the code generator

4.1.1 Core Architecture

The code generator, named CyprIoT Code Generator (CY-CGEN), interprets the CY-Model and generates the corresponding low-level code of each imported thing. The core components are depicted in Figure 4.1. It takes as input the CY-Model, based on CY-DSL, and follows three steps: a) it loads the CY-Model and the TH-Models it contains, b) it transforms the TH-Models to wire them as specified by the CY-Model and enforces the policies, and c) it generates the low-level application artifacts (e.g., code, documentation, access control rules).

The core of the CY-CGEN (CG-Core) is composed of three modules: Network Generator (CG-NG), Plugin Loader (CG-PL), and a Command Line Interface (CG-CLI). The CG-NG processes the CY-Model following a sequence of phases executed in order. This order ensures that the requirements of each phase have been met before its execution. These phases are in order: *Load, Transform, Generate*. Each phase has a responsibility depicted in Table 4.1. The CG-NG module transforms the TH-Models according to the CY-Model. Then, using TH-CGEN as a library generates the low-level code for the target programming language (e.g., Java, C, Go) specified for the instance in the CY-Model. The CG-PL is responsible for loading plugins that are hooked into CG-Core (cf. Section 4.1.3). The CG-CLI offers a means to use the CY-CGEN as a standalone application. It takes as an input the CY-Model along with some optional arguments. Then, based on these elements, it produces the application artifacts.

We concentrated our efforts into the three 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 the network. This process is iterative and can be improved over time to cover more complex networking scenarios.

4.1.2 Model Loading

Model Loading (Mod-Load) is the first major step of code generation. It takes place at the *Load* phase and has the function to load and process the behavior of things. In this thesis, we presume that the behavior of things is based on a TH-Model. We show in Section 4.2 how things are transformed to conform with the network specification. ThingML is, as presumed by their authors, Turing complete. Therefore, it enables to model theoretically any behavior in the form of a statechart. The Mod-Load selects the TH-Models

Table 4.1 – Code generator plugin interfaces

Interface	**Input**	**Output**	**Task**
Load	Network Model	ThingML Models	Load a thing with a behavior, then convert into a ThingML model
Transform	Network Model ThingML models	ThingML Models	Transform the ThingML models to conform with the network model
Generate	Network Model ThingML Models	Network artifacts	Generate network artifacts

inside the CY-Model and transfers them to the relevant transformation procedure in the *Transform* phase. As shown in Table 4.1, the Mod-Load requires as input a CY-Model and returns the included TH-Models.

By opting for ThingML as the only solution to design a thing's behavior, we create an obstacle to the adoption of our methodology. One must learn ThingML to use our methodology correctly. Still, we offer some preliminary insights w.r.t. this issue. To include non-model-based things such as a legacy thing, we suggest loading a behavior from a concrete code, as long as the engineer can provide a procedure to be reverse-engineered into a TH-Model. Thus, satisfying the output requirement of the *Load* phase.

As proof of concept, during our study, we have reverse-engineer the external interfaces of an Arduino program. We developed a plugin that loads the program, finds the chunks of code that handle the protocol commands (e.g., Publish or Subscribe commands for MQTT) and then renders a TH-Model embedding the concrete code of the behavior as is, but abstracting its communication interfaces into ports so that they can be wired within CY-DSL. This approach provided us with some preliminary insights w.r.t. this issue, that manifestly needs more work in the future.

4.1.3 Extensibility

The implementation choices of CY-CGEN are not definitive. Its modular architecture fosters extensibility and separate concerns. CY-CGEN provides some core features necessary for implementing a basic network, i.e., wiring the things and enforcing the policies. It does not implement *expertise knowledge* but offers mechanisms to achieve it by experts using plugins. By *expertise knowledge*, we refer to the ability to use the network model information to extract some advanced knowledge w.r.t. the network, i.e., to use the network elements as the input of a specific procedure to generate some original artifacts that generally requires some expertise (e.g., the specification of access control rules).

The proposed architecture provides access to the core of the generation process via plugins. Table 4.1 shows the available interfaces for plugins where each interface corresponds to a phase. It accomplishes a definite task and imposes a specific type of input and output. Indeed, a typical network of things may involve some expertise (e.g., safety, access control, data consistency). The plugin system provides experts a way to include their concerns in the code generation process. For instance, as a proof of concept, we implemented a plugin to generate access control rules for a Mosquitto MQTT broker (cf. Section 4.2.2). We hook this plugin to the *Generate* interface that provides us with the network model and the transformed ThingML models. We then utilize them to produce a set of access control rules that must be enforced in the broker for a safe deployment. The same principle may be applied for any other network artifact.

4.2 Model Transformation

MT [78] is a process based on transformation rules (usually written by experts) that takes one or more input models to produce a target artifact. We use both types of transformations, i.e., M2MT and M2TT, for the interpretation of the network model.

The Transformation Process (T-PROCESS) (cf. Figure 4.2), i.e., the transformation work accomplished inside the CY-CGEN, takes place at the *Transform* phase. It has the function to transform the input models, i.e., the 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 4.2, we use ATL[1] for M2MT and Acceleo[2] for M2TT, two state-of-the-art transformation tools. We use two technologies for transformation because ATL targets only M2MT while Acceleo targets only M2TT. ATL uses rules to apply the transformation, where it matches an element in the input model to produce another element

1. https://www.eclipse.org/atl/
2. https://www.eclipse.org/acceleo/

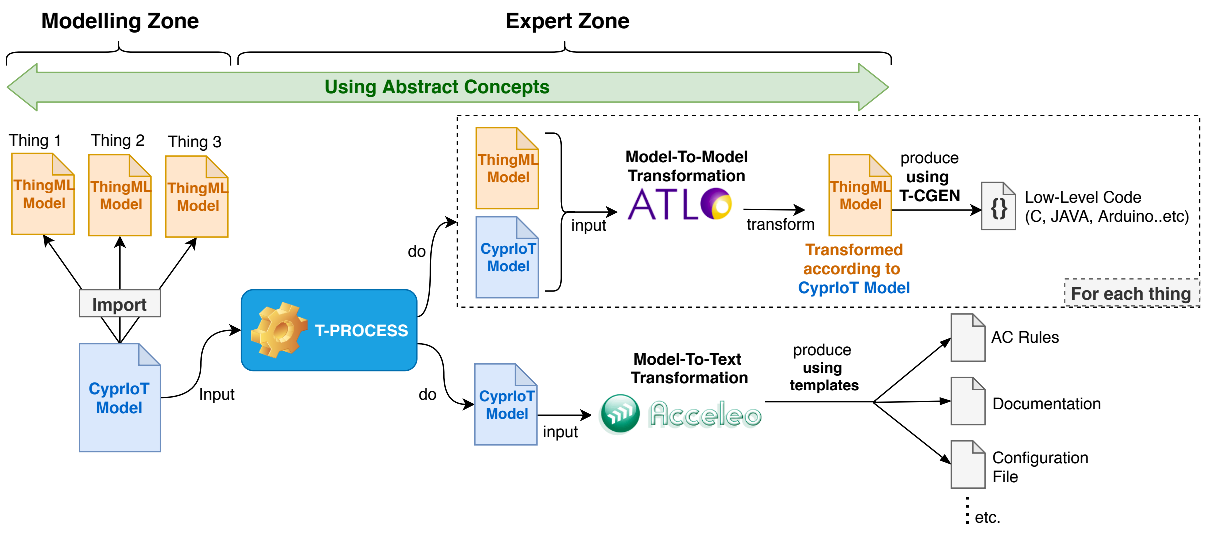


Figure 4.2 – Generation of network artifacts using the T-PROCESS; Modeling Zone: specification of the network and things; Expert Zone: interpretation of abstract concepts into low-level concepts by experts for the generation of artifacts

in the output model that is satisfying the rule. Acceleo relies on templates, where it fills a template's placeholders with information from the input model.

The next subsections explain how ATL and Acceleo generate our network artifacts.

4.2.1 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, 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 of this process are provided in Appendix C and depicted in this section using Graph Transformation Rules (GTR) [155]. 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 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

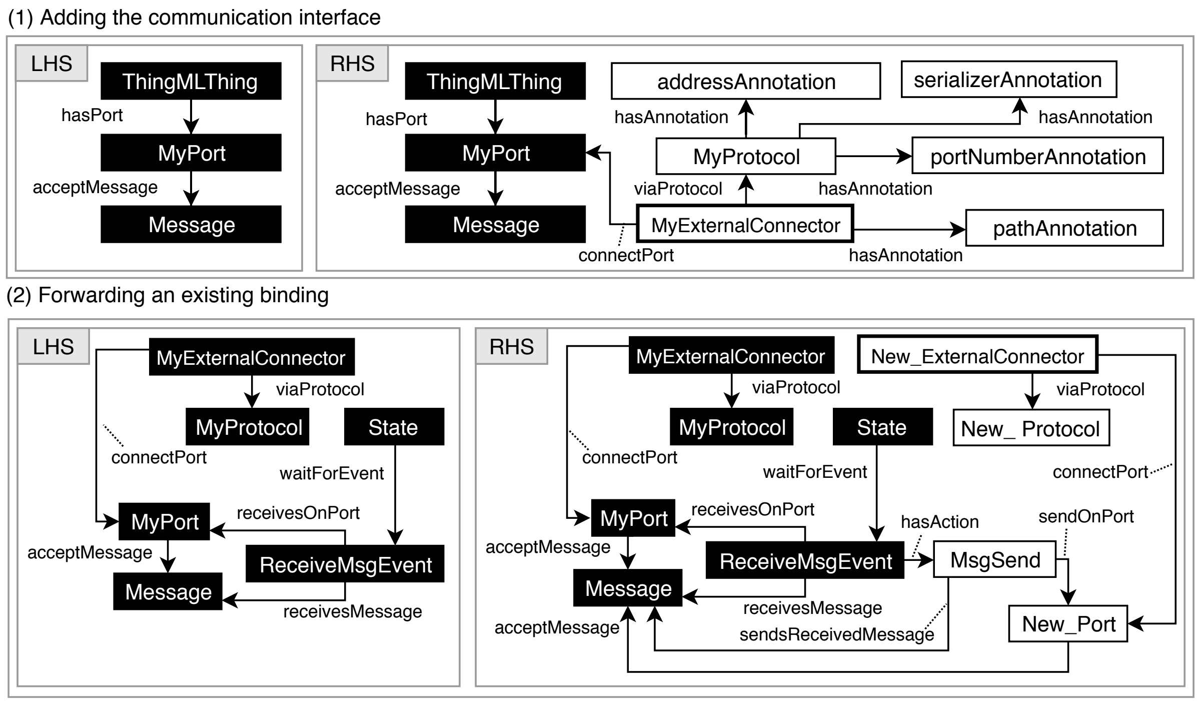


Figure 4.3 – Upper Part (GTR (1)): Adding the communication interface according to the specification of the network. Lower part (GTR (2)): Forwarding an existing binding; white boxes are added based on the CY-Model; the added External Connector box is thicker for readability purpose only.

and path) and (2) *forwarding an existing binding*. (1) and (2) tackle the networking problems in Chapter 2 respectively in Sections 2.1.4 and 2.1.5. In Section 4.3, we show how we tackle the enforcement of the policies.

Wiring

The upper part of Figure 4.3 depicts the GTR of (1), the full code in ATL syntax is provided in Appendix C. On the RHS, the 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, 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 4.1, 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". The annotations values are defined within TH-CGEN and must be filled for each protocol within the model. This specification 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. Indeed, ThingML offers a plugin system to implement protocols. The plugin developer must define how to map the protocol's specification in the model to its equivalent in a target programming language. 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.

Forwarding

The lower part of Figure 4.3 depicts the GTR of (2), the full code in ATL syntax is provided in Appendix C. 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 bride 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.

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 4.2) 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.

4.2.2 Model-to-Text Transformation

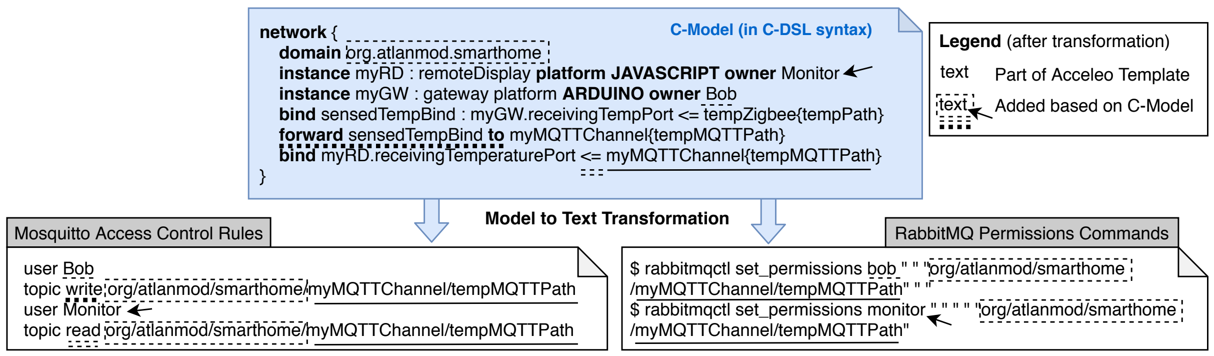
The lack of low-level concepts' unification adds another layer of complexity to interoperability between things. Indeed, heterogeneity of low-level concepts (e.g., user, topic, URL, permission, documentation, configuration) inhibits connecting things safely and leads to poor synchronization between all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

M2TT allows us to generate any related textual artifact that is not part of the internal behavior of a thing (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. For instance, in the example of Listing 4.1, we must 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 must write it in the right syntax, i.e., the syntax of the target broker at low-level; Figure 4.4 shows an example for Mosquitto[3], RabbitMQ[4]. The goal of M2TT is to generate this kind of artifacts based on templates. It uses the network's unified specification to synchronize, using an automatic process, 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 4.4 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 provide a demonstration on 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.

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

3. https://mosquitto.org
4. https://www.rabbitmq.com


C-Model (in C-DSL syntax)
Legend (after transformation)
text
Part of Acceleo Template
text
Added based on C-Model
Model to Text Transformation
Mosquitto Access Control Rules
RabbitMQ Permissions Commands

```
network {
    domain org.atlanmod.smarthome
    instance myRD : remoteDisplay platform JAVASCRIPT owner Monitor
    instance myGW : gateway platform ARDUINO owner Bob
    bind sensedTempBind : myGW.receivingTempPort <= tempZigbee{tempPath}
    forward sensedTempBind to myMQTTChannel{tempMQTTPath}
    bind myRD.receivingTemperaturePort <= myMQTTChannel{tempMQTTPath}
}
```

```
user Bob
topic write org/atlanmod/smarthome/myMQTTChannel/tempMQTTPath
user Monitor
topic read org/atlanmod/smarthome/myMQTTChannel/tempMQTTPath
```

```
$ rabbitmqctl set_permissions bob " " "org/atlanmod/smarthome
/myMQTTChannel/tempMQTTPath" " "
$ rabbitmqctl set_permissions monitor " " " " "org/atlanmod/smarthome
/myMQTTChannel/tempMQTTPath"
```

Figure 4.4 – 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.

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 (e.g., requires learning a new syntax, synchronizing the artifacts manually) and exposes the IoT engineer to introduce more bugs.

4.3 Enforcement Strategies

The enforcement refers to the implementation of the policies in the generated artifacts. It takes place at the *Transform* phase and relies on MT. In Section 4.3.1, we present our strategy to implement the enforcement of communication control rules, and in Section 4.3.2, we present our strategy to implement the smart rules. We also point out the various other strategies that could be implemented.

4.3.1 Enforcement of Communication Control Rules

The enforcement of communication control rules consists of interpreting the rules presented in Section 3.2 to include them inside the deployable network artifacts. In this section, we first discuss the possible enforcement checkpoints in a network (i.e., areas where we can take action for enforcement), then, the enforcement mechanisms.

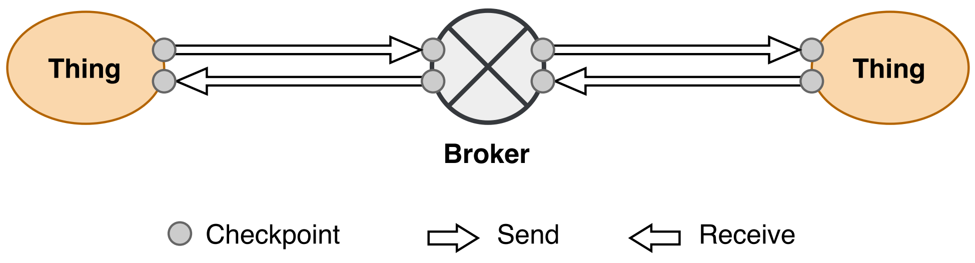


Figure 4.5 – Enforcement checkpoints

Enforcement Checkpoints

As depicted in Figure 4.5, 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 on receive. The choice of the right 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 traveling and demands additional processing on receive. This processing may waste (scarce) resources in case 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 broker.

Enforcement Mechanisms

Our methodology relies on separating the specification from the implementation; the policy's enforcement depends on the parameters of the specified network. Hence, many enforcement strategies at the implementation may be adopted 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.

The full code in ATL syntax is provided in Appendix D.

Figure 4.6 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 4.7, but on the *send checkpoint*. We consider that this enforcement is 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 technical reasons. Indeed, several deployment factors, such as the expected architecture or hardware infrastructure, may influence the enforcement. So far, we experimented with these enforcements with specific usecases consisting of the most used things and channels. If these enforcements do not fit the technical environment at runtime, experts must develop a plugin to fill this gap.

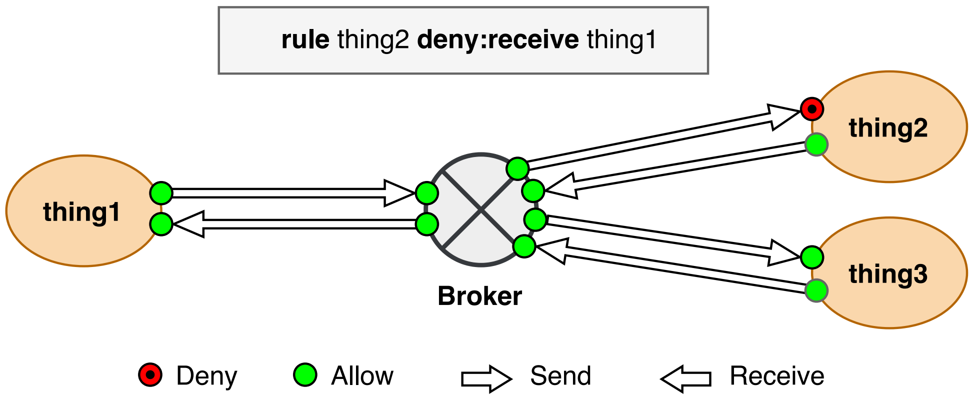


Figure 4.6 – Enforcement at the “on receive” checkpoint

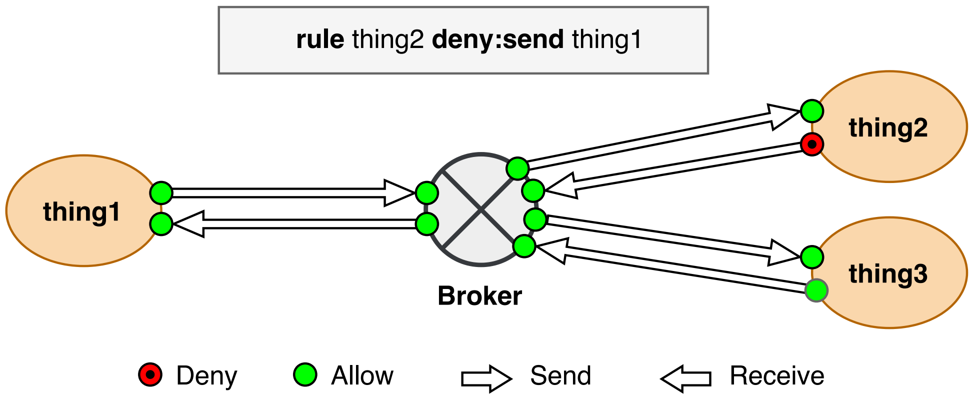


Figure 4.7 – Enforcement at the “on send” checkpoint

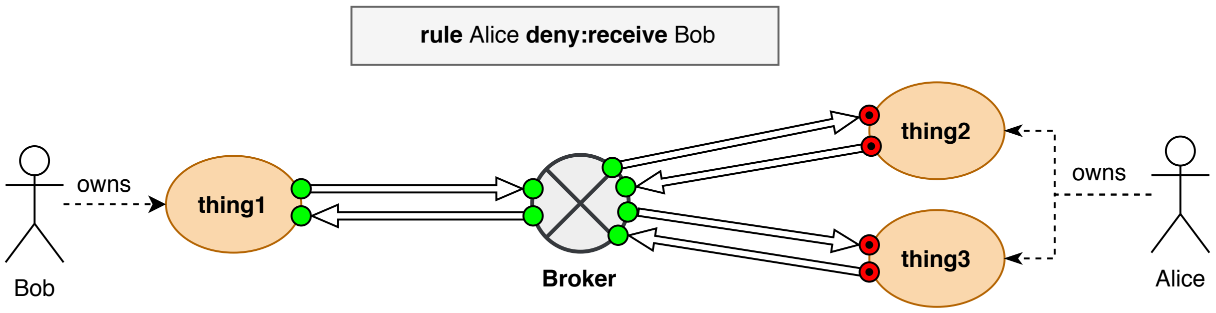


Figure 4.8 – Enforcement of a user-based rule

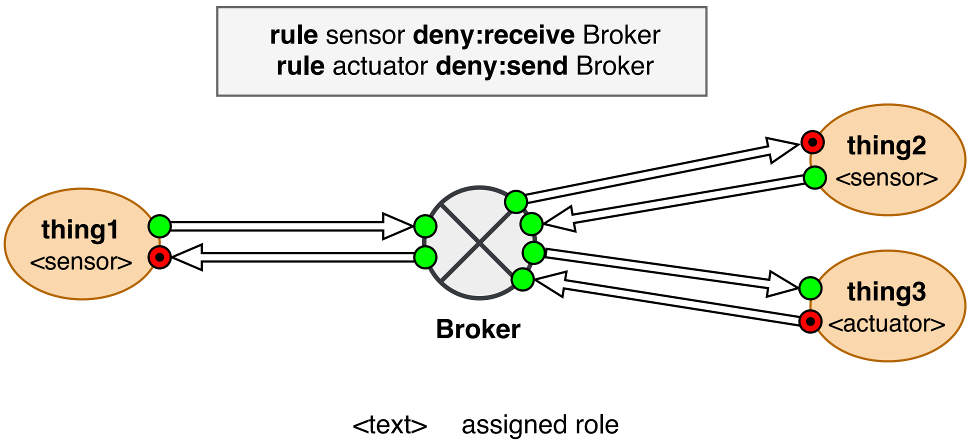
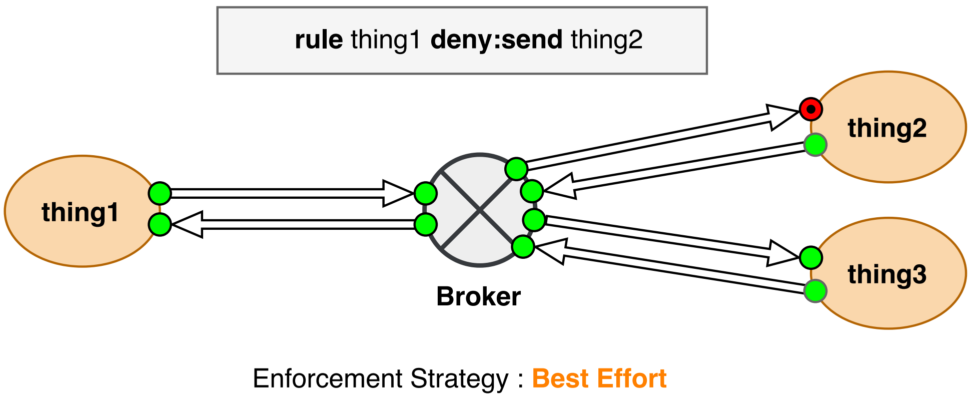


Figure 4.9 – Enforcement of a role-based rule

Figure 4.8 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 4.9 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 undecidable, i.e., it is impossible for the CY-CGEN to decide the correct enforcement. For instance, the rule in Figure 4.10 denies *thing1* from sending a message to *thing2*. *thing2* and *thing3* consumes the same path. If we deny *thing1* from sending to the path of *thing2*, we will be preventing *thing3*



rule thing1 **deny:send** thing2
thing1
thing2
thing3
Broker
Enforcement Strategy : **Best Effort**

Figure 4.10 – Enforcement of a Best-Effort strategy

from receiving the message. Thus, the correct enforcement, as stated by the rule, is not possible in this configuration. In this case, rely on the enforcement strategy, as shown in Line 3 of Listing 4.1, to find a trade-off.

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

Listing 4.1 – Specifying the enforcement strategy

The enforcement strategy serves as a tie break in contentious rules. It backs CY-CGEN to decide how to interpret the rules and implement them at the transformed TH-Models. We suggest three strategies: **Best—Effort**, **Deny—First** and **Allow—First**. The **Best—Effort** strategy is our default strategy. It enforces correctly as many rules as possible and finds a trade-off with minimal risk to safety for contentious rules. For instance, Figure 4.10 presents a contentious rule where we enforce the control at *thing2* by preventing it from receiving any message from *thing1*, instead of enforcing this control at *thing1* as this exposes *thing1* to leak its messages to *thing3*. The strategy **Deny—First** presumes that all communications are denied unless there is a rule allowing them. While the strategy **Allow—First** assumes the opposite, i.e., all communications are allowed unless there is a rule denying them. These enforcement strategies must be implemented once by an expert in the form M2MT or M2TT and are applied automatically by CY-CGEN.

4.3.2 Enforcement of Smart Rules

As the network specification relies on unified concepts at the model-level, we avoid 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 4.2, 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.

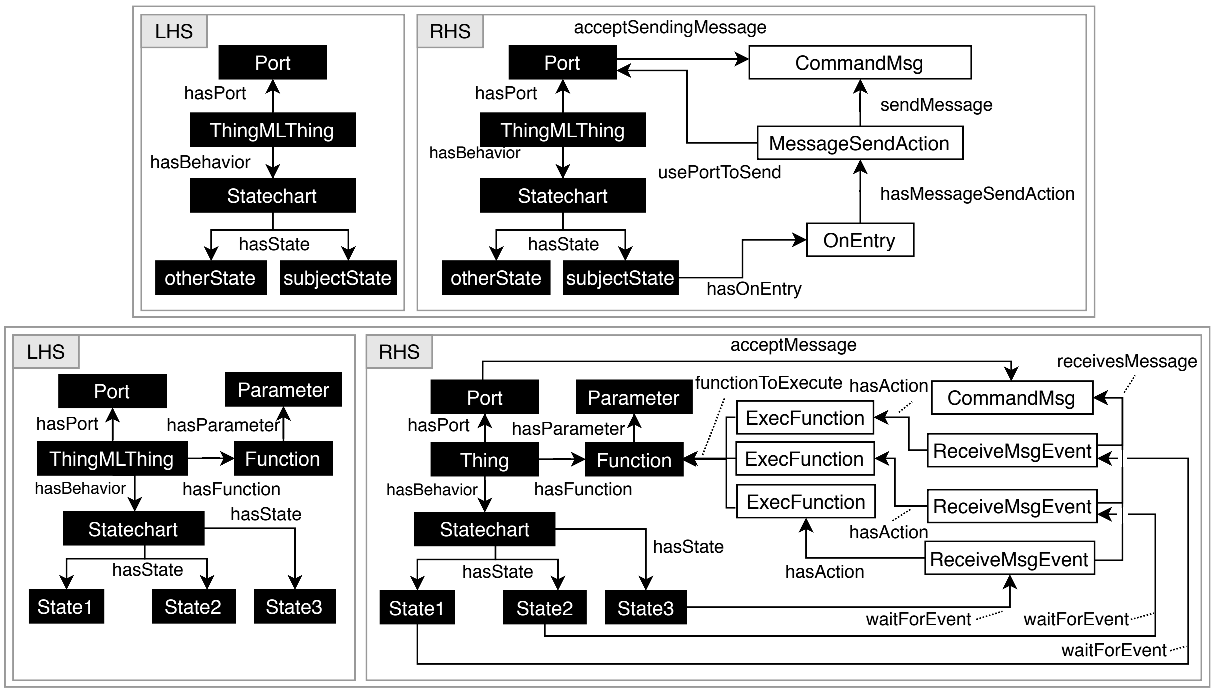


Figure 4.11 – Upper Part for Subject, Lower Part for Object; applying the trigger:executeFunction rule GTRs (3) with the subject thing having two states and the object thing having three states; white boxes are added based on the CY-Model.

There may be various way 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 4.11 depicts the GTR of *trigger:executeFunction* rule (cf. Line 3 of Listing 3.1) and Figure 4.12 the GTR of *trigger:goToState* rule (cf. Line 2 of Listing 3.1). The full code in ATL syntax is provided in Appendix C.

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 must 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. We create a message and send it on the entry of the subject state (i.e., we use *MessageSendAction OnEntry* to send *CommandMsg*). ThingML offers a mechanism to listen to events within a thing (e.g., receiving a new message event). 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 of 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 try to find an indirect path.

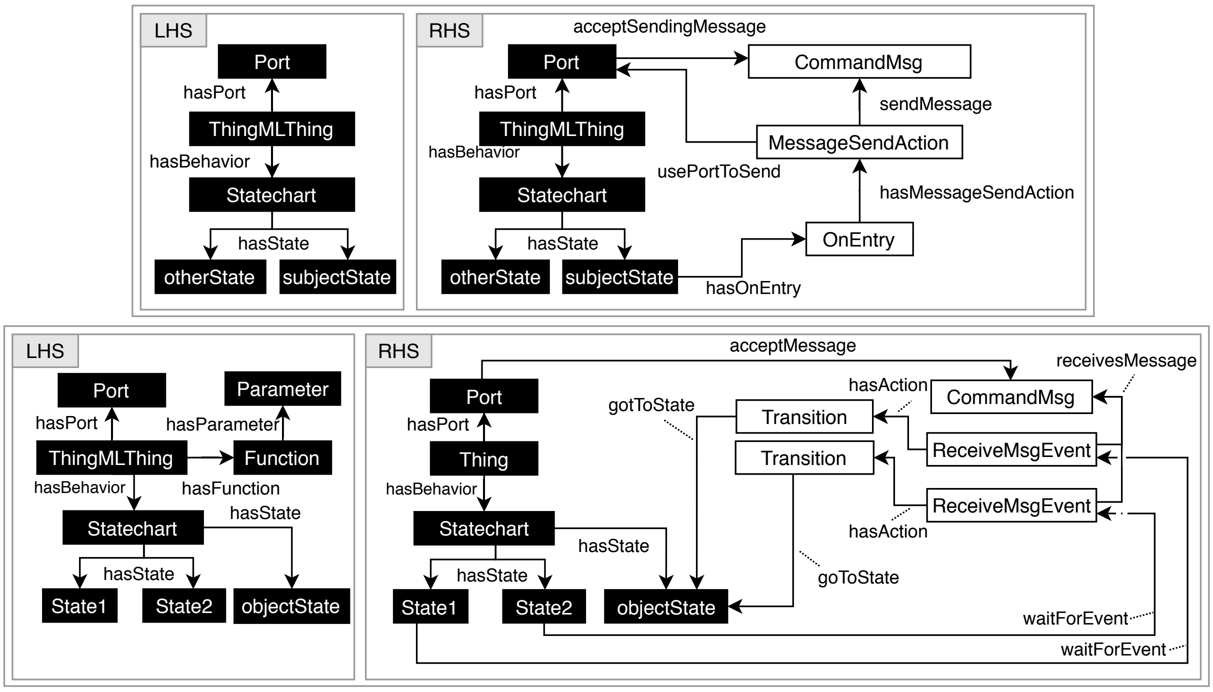


Figure 4.12 – Upper Part for Subject, Lower Part for Object; applying the trigger:goToState rule GTRs (4) with the subject thing having two states and the object thing having three states; white boxes are added based on the CY-Model.

The *trigger:goToState* rule in Figure 4.12 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.

4.4 Summary

The last step of our methodology consists of generating network artifacts from the network model. This chapter introduced an extensible code generator, named CY-CGEN, capable of interpreting the model and generating the specified network's artifacts. Figure 4.13 depicts the global picture of our methodology, compared to the previous version, we added the part responsible of the generation of the artifacts.

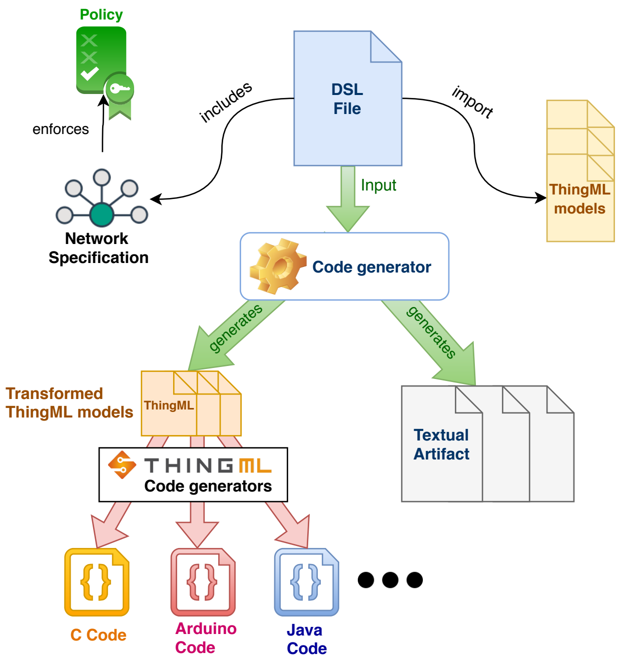


Figure 4.13 – Generation of the network artifacts: the last step of the methodology

Chapter 5

Assessment

This chapter wraps up the content of the present thesis and to highlight its value. Section 5.1 embodies the engineering steps of the methodology. Section 5.2 evaluates it. Section 5.5 discusses its most valuable insights and raises its limitations.

5.1 Methodology

We present our methodology in the form of a rigorous procedure to be easily reproduced or extended. Moreover, we apply it on a hypothetical case study in Section 5.3, and provide directions for some of its potential applications in Section 5.4.

We opted for a BPMN to outline the engineering process to follow by software engineers, according to their responsibilities. BPMN is a model-based method proposed by the OMG to describe the process of business activities using a readable diagram [156]. Figure 5.1 depicts the BPMN diagram of our methodology. Three responsibilities share the execution of the tasks presented previously (cf. Section 2.1), the **Thing Designer**, the **Network Designer**, and the **Policy Designer**. Each responsibility is contained in a row that comprises a set of tasks.

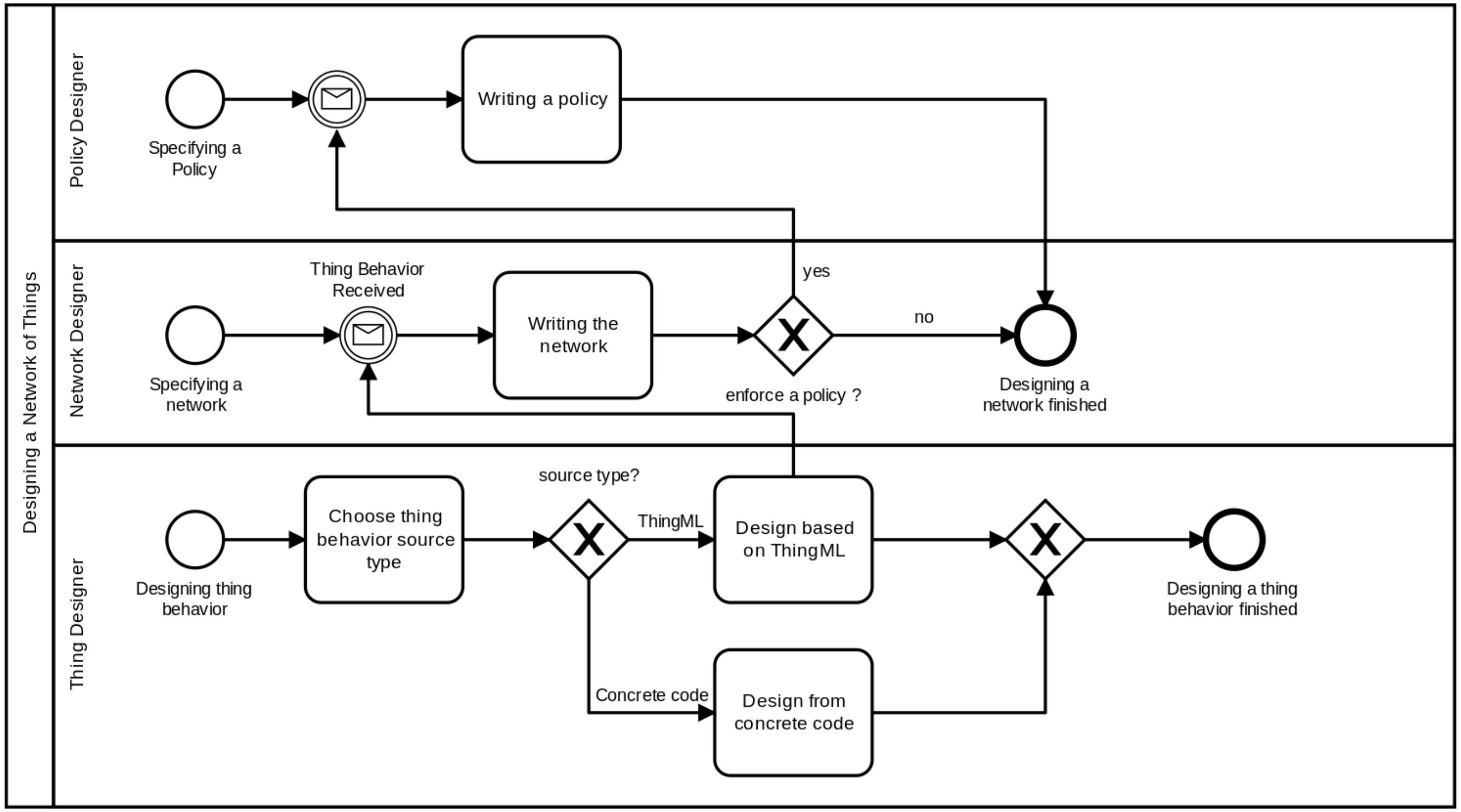


Figure 5.1 – BPMN diagram of the methodology

5.2 Evaluation

In this section, we discuss our results. Section 5.2.1 evaluates from a quantitative perspective the performance of our methodology in terms of Lines of Code (LoC). From a qualitative perspective, Section 5.2.2 points out some of its essential characteristics that improve software engineering for the IoT as a whole.

5.2.1 Quantitative Evaluation

We tested our methodology on networks ranging from 1 to 25 things. We use traditional software engineering as a baseline, with things based on C (suitable for any range), Java (ideal for medium and large things), and Arduino (ideal for small things). We compare the 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 within the ThingML approach [90]. We compare only the LoC needed for networking and policy enforcement. We removed the comments and empty lines from the counts.

For this experiment we used MQTT as a communication mean. C, Java, Arduino and MQTT are among the most used technologies in the IoT [157]. Table 5.1 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.

Table 5.1 – 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

The LoC correspond to the white elements in the GTRs (cf. Section 4.2), added automatically by CY-CGEN. 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 must connect each thing separately (this task is automated by the wiring and forwarding M2MTs, cf. Section 4.2.1), and eventually make it part of a smart scenario (automated by the enforcement of policies, cf. Section 4.3). Traditional engineering is time-consuming in particular for large networks 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.

Figure 5.2 shows the time needed to generate the network according to the number of things. We notice that the execution time grows exponentially, possibly due to some implementation issues. As future work, we will explore some implementation strategies to improve the scalability of code generation.

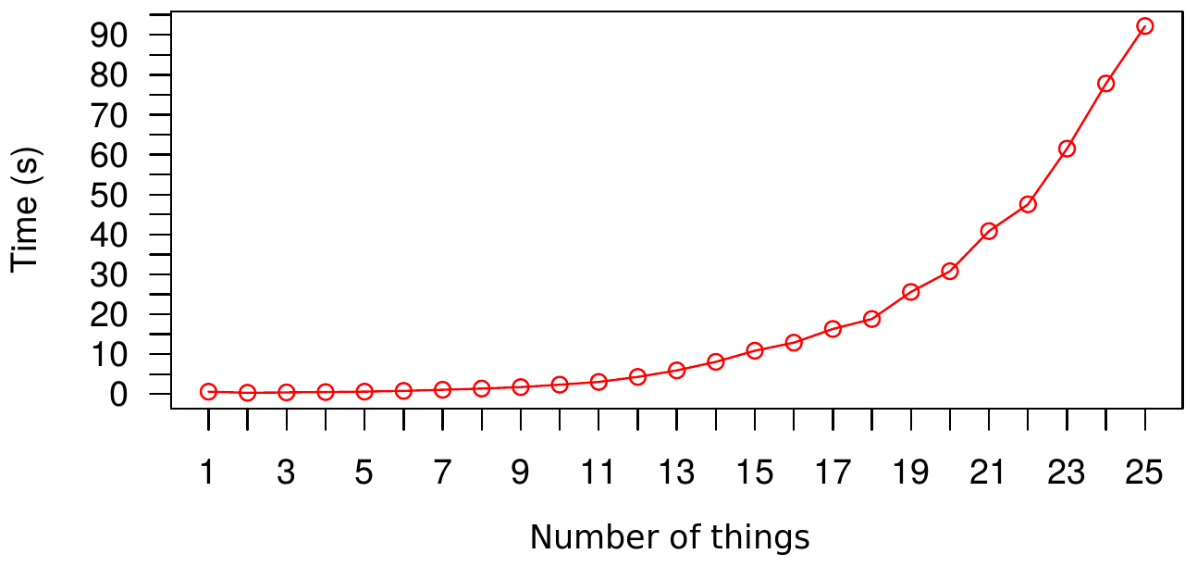


Figure 5.2 – CY-CGEN execution time according to the number of things

5.2.2 Qualitative Evaluation

The present thesis introduces a methodology also contributing to some intangible factors w.r.t. to software engineering for IoT, such as the communication between stakeholders, the separation of concerns, and the ability for early bug detection. We describe here the qualitative value of our contribution in that respect.

Communication skills are crucial in software engineering [147]. Indeed, having a high-level model of the network could improve communication between stakeholders as it fosters sharing development results and facilitates close collaboration [9]. The CY-Model, because it uses primitive concepts and a representation, presents the intended networking scenario in a readable fashion. In this model, changes can be applied earlier to fit stakeholders' needs before moving forwards. Also, the CY-Model serves as a unique source to generate the artifacts needed by each stakeholder using the CY-CGEN or, eventually, a manual process. This disposition allows us to quickly move towards a common goal [148] by relying on a unique global picture.

The process in Figure 5.1 depicts the concerns and their relationships, making each step of the engineering deterministic and consequently manageable. Indeed, each matter has its specific set of tasks directed towards a particular interest (i.e., thing behavior tasks for the Thing Designer, network behavior for the Network Designer, and constraints on

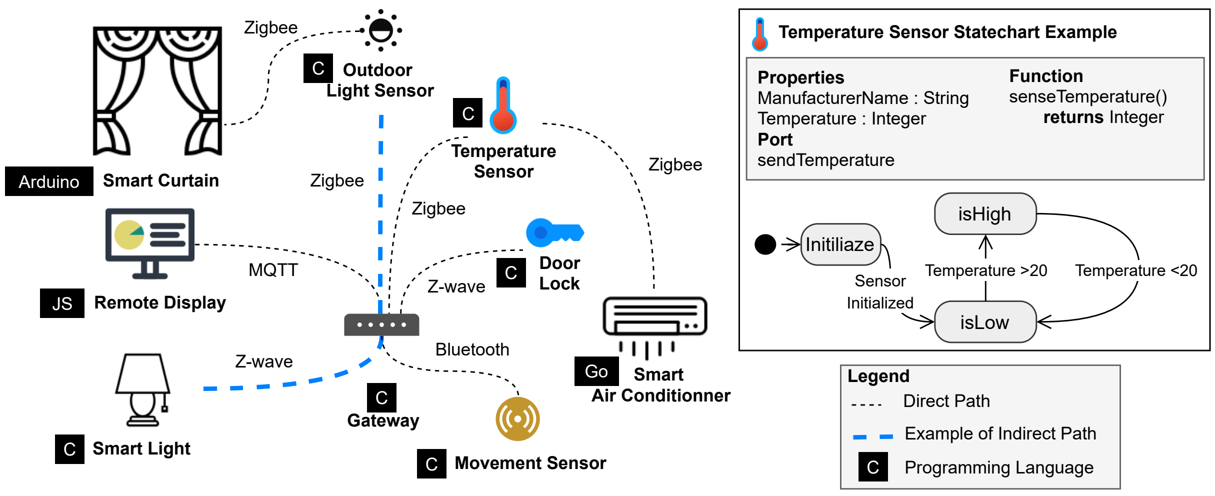


Figure 5.3 – A smart home example

the network for the Policy Designer).

The CY-DSL provides some real-time insights in the editor to prevent inconsistencies (e.g., communication using incompatible message formats, incompatible port and path), using auto-completion or by showing suggestions. These insights give the software engineers a tool to avoid bugs earlier and are consequently saving time [145, 146].

Finally, the evaluation of communication control rules lacks because it was difficult to quantify their benefits (some rules may remove LoC, and some may add more), we will evaluate them in the future. Nevertheless, these rules save time by exempting the IoT engineer from controlling the communication using heterogeneous concepts at a low-level. Generally speaking, we expect that some significant time would be saved using our methodology, w.r.t. dealing with heterogeneity. Indeed, adapting each thing at a low-level requires more engineering skills to find the right solutions to wire heterogeneous things, while this redundant work could be automated in the CY-CGEN.

5.3 Case Study: Smart home

To illustrate our methodology, we present a case study consisting of a simple smart home. In Section 5.4, we point out the possibilities to leverage our methodology on more potential applications.

5.3.1 Description

In this section, we describe the case study, namely the network and the policies, then we provide its implementation with CY-DSL syntax in Section 5.3.2.

Network

We illustrate our study with the smart home of Figure 5.3. Bob is a user and the owner of the smart home. We opted for a smart home as it generally contains heterogeneous things and protocols and requires few smart scenarios. Heterogeneous computing platforms (e.g., C, Java, Python) and communication protocols (e.g., MQTT, Zigbee, Z-Wave, Bluetooth) are used.

The Gateway (C, owner: Bob) is based on an Arduino program and supports Zigbee, Z-wave, and Bluetooth. The first group consists of tiny things, namely the Temperature Sensor (C/Posix, owner: Bob), the Outdoor Light Sensor (C/Posix, owner: Bob), and the Movement Sensor (C/Posix, owner: Bob). The second group consists of medium things; namely the Smart Light (C/Posix, owner: Bob), the Door Lock (C/Posix, owner: Bob). The third group consists of large things, namely the Smart Air Conditioner (Go, owner: Manufacturer—presuming that some devices are owned by their manufacturer for maintenance reasons), the Smart Curtain (Arduino, owner: Manufacturer), and the Remote Display (Javascript, owner: Manufacturer). Three ranges are represented, each corresponds to a role assigned to the thing: small (i.e., resource-constrained, e.g., temperature sensor), medium (sufficient resources, but limited, e.g., gateway), large (no resource constraints, e.g., Remote Display). The wiring of these things follows the scheme in Figure 5.3 and is the Network Designer's responsibility.

The behavior of things is based on TH-Models, consisting of statecharts. Sometimes, we could not represent the entire behavior using only ThingML concepts; in this case, we embedded some low-level code in some states of the statechart. The correctness of this embedded code is the responsibility of the Thing Designer. From a Network Designer perspective, what matters is the wiring of these TH-Models, i.e., the communication flow.

Policies

We presume that we have two policies, one for communication control named commPolicy , encapsulating the interest of a maintenance agent, and the other for smart scenarios called smartPolicy, encapsulating the interest of an automation agent.

The commPolicy consists of these two rules:

— **Usecase 1:** For privacy reasons, Bob devices should not communicate with the manufacturer devices.

— **Usecase 2:** Because of their limited resources, the small things should be denied to communicate directly with large things.

The smartPolicy consists of these three rules:

— **Usecase 3:** If the temperature is high, the air conditioner should start cooling.

— **Usecase 4:** During the night, the light should be turned on and the smart curtain should be closed.

— **Usecase 5:** If there is no movement in the house, the door should be locked.

5.3.2 Implementation

First, in Listing 5.1, we start by declaring the types of things that we use in the network, each declaration imports the TH-Model encapsulating the thing's behavior. We also assign to each thing a role corresponding to its size. We will use this information in the policies (cf. Listing 5.3).

Second, in Listing 5.2, we declare the necessary channels for our network. We have five channels. The first channel is dedicated to the Zigbee communications in the Gateway; it contains two paths, one for the communication with the Temperature Sensor and another for the Outdoor Light Sensor. The second channel is dedicated to the Gateway's Bluetooth communication; it contains one path for communication with the Movement Sensor. The third channel is dedicated to Z-wave communications in the Gateway; it comprises two paths, one for the communication with the Smart Light and another for the Door Lock. The fourth channel is dedicated to the MQTT communication between the Gateway and the Remote Display. This channel aims to forward the existing local communications between the Gateway and the local things, to the Remote Display. The Gateway serves as a more powerful thing to permit these small things (i.e., Temperature Sensor, Outdoor Light Sensor, Smart Light, Door Lock) to communicate with a larger thing (Remote Display). The channel contains four paths, each of these paths is meant to forward the Zigbee and Z-wave communications of the Gateway. The fifth channel is dedicated to the local Zigbee communication between the Smart Curtain and the Outdoor Light Sensor.

Third, Listing 5.3 we declare two policies; the policy named commPolicy aims at controlling the communication in this network. The rule in Line 4 implements the usecase

1 by restricting the things owned by the user Bob from communicating with the things owned by the Manufacturer. The rule in Line 7 implements the usecase 2, by preventing the things with the role smallThing from communicating with those with the role largeThing.

The policy named smartPolicy aims at creating smart scenarios. The rules in Lines 14 and 15 implements the usecase 3, i.e., when the TemperatureSensor is in the state isHigh, the smartAirCond need to in the state isOn and execute the function setTemperature to set the temperature to 25řC. The rules in Lines 18 and 19 implements the usecase 4, i.e., when the outdoorLightSensor is in the state dark, the smartLight needs to go to the state isOn and the smartCurtain needs to go to the state isClose. The rules in Line 22 implements the usecase 5, i.e., when the movementSensor is stable, the smartLock needs to go to the state isLock.

Fourth, Listing 5.4 shows the implementation of the network. We enforce the former policies in Line 5. Then, we create, from the one hand, the instances of things between the Lines 7 and 15, and from the other hand, the instances of channels between the Lines 17 and 20. We bind the ports of the instances of things with the channels as intended by the local network topology between the Lines 23 and 41. Finally, we forward the existing local communications of the Gateway to the Remote Display via MQTT between the Lines 43 and 46.

5.4 Potential Applications

Our methodology offers the first brick for a fully integrated model-based software engineering approach for the IoT. The plugin system opens the door for more extensibility. Thus, if the current version does not satisfy an IoT application's requirements, one can still create a plugin to cover its needs. In this section, we enumerate some future potential applications.

5.4.1 Smart City

At its current version, the CY-DSL supports the specification of only one network. Yet, in the future, we can easily extend it to cover more than one network, with the possibility to interconnect them. In that respect, our methodology could incorporate larger applications involving many networks, such as a smart city.

A smart city [158] may contain multiple networks and policies, e.g., a network for traffic management, a network for electricity distribution, or a network for law enforcement. These networks may be wired at the model-level independently from their technical implementation and enforce various policies.

5.4.2 Industry 4.0

Industry 4.0 [159] consists of multiple networks achieving various scenarios in an industrial context. It generally consists of static networks. Rigor in the specification of these networks is highly desirable for efficiency and failure avoidance.

Moreover, the communication between stakeholders (e.g., quality control officer, safety officer, marketing manager) is crucial, hence the need for a common representation such as a network model. This model could be a great asset to keep the processes transparent for all these stakeholders.

Finally, in such a context, a model could also be used for various purposes, such as real-time visualization, diagrams, security purposes, and automation.

5.5 Discussion

MDE helps automate many redundant tasks during the software engineering cycle of an IoT application. The specification of a network helps unify the language, foster sharing the development results of stakeholders, and separate concerns at the model-level, consequently improving communication between engineers [9]. The code generator avoids manual work that is often a source of bug-prone code. This approach bridges the gap between heterogeneity and interoperability, two necessary but conflicting ideals for the IoT. Nevertheless, some remarks and threats to validity should be considered. We discuss them in this section.

5.5.1 Research Considerations

We justify a few research choices taken throughout the present work. First, we explain the reasons behinds the way of designing a network. Second, we give our point of view w.r.t. heterogeneity in the IoT. Third, we discuss how we are tackling the interoperability problem in the IoT [160, 140].

On Designing the Network

In our approach, we assume that the network's design must be fixed during the modeling phase, i.e., that its components (e.g., things, channels, users) and its topology are defined during this phase. This assumption excludes any dynamic network, i.e., a network that may change at runtime, from the present thesis's scope.

We envision tackling dynamic networks in the future. We consider that the first major milestone of the methodology, that we address in the present work, consists of connecting things regardless of their implementations at low-level (e.g., languages, protocols) at the model-level without interoperability issues.

At low-level, implementing a simple exchange in the IoT requires ensuring separately that the things have a way to exchange information, either directly (using the same protocol) or indirectly (using an intermediary thing). 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 software engineering work is bug-prone, repetitive especially for large networks.

On Heterogeneity

The intrinsic heterogeneity of the IoT creates barriers to interoperability between ranges. A range is any collection of things that are using quite homogeneous technologies. For instance, because of its limited resources, a small thing may not send data directly to a larger one, such as a remote server. We need a bridge between them. Implementing correctly such a bridge is not trivial, as a good deal of expertise, platforms, protocols, and hidden risks are involved. Hence, the necessity to separate concerns, as advocated throughout this work.

Moreover, the lack of consensus for standard and proprietary technologies leads to interoperability issues within the same range. For instance, a thing supporting Zigbee cannot communicate directly with a thing using Z-wave, although there are both in the same range. A common practice consists of using an intermediary thing such as a gateway, supporting both protocols. Its role is to forward the message from the first thing to the second. Implementing this mechanism at low-level is not always trivial among heterogeneous technologies. We address this issue with the forwarding concept at the model-level.

On Interoperability

Interoperability is a turning point to unlock the full potential of the IoT [161, 162]. As shown earlier, many approaches tackle the interoperability problem by avoiding heterogeneity, either with standards [163, 164] or a layer at runtime responsible for communication [110, 104]. Indeed, addressing the problem of interoperability at a 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. Hence, the need for a unifying methodology.

Moreover, while standardization is sufficient in a homogeneous context (e.g., HTTP browsers, specific range of things), the challenge of connecting Anything, Anywhere, Anytime (AAA) requires a more generic and inclusive software engineering solution that embraces heterogeneity as an intrinsic feature of the IoT. The present methodology aims to unify software engineering tools for the IoT, rather than eliminating heterogeneity with a standard.

Finally, solving the interoperability problem in the IoT for any possible scenario is challenging. We showed that MDE offers promising tools to, at least, unravel the problem rigorously. Thus, enabling more straightforward networking, smart scenarios, and the automatic generation of some textual artifacts contribute to interoperability. MDE could help contain the problem without sacrificing the freedom to use the technology that works best for the thing.

5.5.2 Threats to validity

Our approach suffers from some drawbacks of MDE [165, 166]. First, it may add a semantic level for software engineers, provoking resistance for 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. With this regard, we provide some thoughts in conclusion. 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.

Our approach needs more experimental work. First, the metamodel is incomplete and needs more empirical experiments as we still cannot guarantee the inclusiveness and genericity of all networking concepts (i.e., covering all possible low-level elements of a network). 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 code), 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, although 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.

```
// User
user Bob
user Manufacturer

// Roles
role smallThing
role mediumThing
role largeThing

// Importing things
thing homeGateway
  import "homeGateway.thingml"
  assigned mediumThing

thing temperatureSensor
  import "temperatureSensor.thingml"
  assigned smallThing

thing doorLock
  import "doorLock.thingml"
  assigned smallThing

thing outdoorLightSensor
  import "outdoorLightSensor.thingml"
  assigned smallThing

thing smartLight
  import "smartLight.thingml"
  assigned smallThing

thing movementSensor
  import "movementSensor.thingml"
  assigned smallThing

thing smartAirCond
  import "smartAirCond.thingml"
  assigned mediumThing

thing smartCurtain
  import "smartCurtain.thingml"
  assigned mediumThing

thing remoteDisplay
  import "remoteDisplay.thingml"
  assigned largeThing
```

Listing 5.1 – Declaration of users, roles and things of the smart home

```
channel gatewayZigbee {
  path olsPath
  path tsPath
}

channel gatewayBluetooth {
  path msPath
}

channel gatewayZwave {
  path dlPath
  path slPath
}

channel gatewayMqtt {
  path olsPath
  path tsPath
  path dlPath
  path slPath
}
```

Listing 5.2 – Declaration of channels of the smart home

```
policy commPolicy {

// Usecase 1
  rule Bob deny:send−receive Manufacturer

// Usecase 2
  rule smallThing deny:send−receive largeThing

}

policy smartPolicy {

// Usecase 3
  rule myTS−>state:isHigh trigger:goToState mySAC−>state:isOn
  rule myTS−>state:isHigh trigger:executeFunction mySAC−>function:setTemperature(25)

// Usecase 4
  rule myOLS−>state:isDark trigger:goToState mySL−>state:isOn
  rule myOLS−>state:isDark trigger:goToState mySC−>state:isClosed

// Usecase 5
  rule myMS−>state:isStable trigger:goToState myDL−>state:isLocked

}
```

Listing 5.3 – The enforced policies in the smart home

```
network smarthomeNetwork {
///// Domain of the network
  domain fr.nantes.bob
///// Enforcing the policies
  enforce commPolicy, smartPolicy // cf. Listing 5.2
///// Instanciating things
  instance gateway: homeGateway owner Bob platform POSIX
  instance myTS: temperatureSensor owner Bob platform POSIX
  instance mySL: smartLight owner Bob platform POSIX
  instance myDL: doorLock owner Bob platform POSIX
  instance myOLS: outdoorLightSensor owner Bob platform POSIX
  instance myMS: movementSensor owner Bob platform POSIX
  instance mySAC: smartAirCond owner Manufacturer platform GO
  instance mySC: smartCurtain owner Manufacturer platform ARDUINO
  instance myRD: remoteDisplay owner Manufacturer platform JAVASCRIPT
///// Instanciating channels
  instance zigbeeGW: gatewayZigbee platform ZIGBEE
  instance bluetoothGW: gatewayBluetooth platform BLUETOOTH
  instance zwaveGW: gatewayZwave platform ZWAVE
  instance mqttGW: gatewayMqtt platform MQTT
///// Binding things and channels
  // Zigbee
  bind myOLS.myPort => zigbeeGW{olsPath}
  bind olsLink: gateway.olsPort <= zigbeeGW{olsPath} // olsLink is an id for this binding
  bind mySC.olsPort <= zigbeeGW{olsPath}
  bind myTS.myPort => zigbeeGW{tsPath}
  bind mySAC.tsPort <= zigbeeGW{tsPath}
  bind tsLink: gateway.tsPort <= zigbeeGW{tsPath} // tsLink is an id for this binding
  // Bluetooth
  bind myMS.myPort => bluetoothGW{msPath}
  bind gateway.msPort <= bluetoothGW{msPath}
  // Zwave
  bind myDL.myPort => zwaveGW{dlPath}
  bind dlLink: gateway.myTS <= zwaveGW{dlPath} // dlLink is an id for this binding
  bind mySL.myPort => zwaveGW{slPath}
  bind slLink: gateway.slPort <= zwaveGW{slPath} // slLink is an id for this binding
  // MQTT
  bind myRD.olsPort <= mqttGW{olsPath}
  bind myRD.tsPort <= mqttGW{tsPath}
  bind myRD.dlPort <= mqttGW{dlPath}
  bind myRD.slPort <= mqttGW{slPath}
///// Forwarding existing links to an MQTT broker
  forward olsLink to mqttGW{olsPath}
  forward tsLink to mqttGW{tsPath}
  forward dlLink to mqttGW{dlPath}
  forward slLink to mqttGW{slPath}
}
```

Listing 5.4 – Configuration of the smart home network (=> send, <= receive)

Chapter 6

Conclusion & Perspectives

The present thesis's outcome is a novel software engineering approach based on MDE for the IoT. We showed that by abstracting the common concepts of similar technologies and separating the business logic from the technical details, we make software engineering for the IoT deterministic and save a significant amount of LoC. MDE helps accelerate engineering by defining separate responsibilities and automating many redundant tasks that may be error-prone, using code generation.

Thus, the presented contributions lead to the following answers to our research questions:

— **RQ1: What software engineering process should be followed to design a network of heterogeneous things?** The literature lacks a consistent software engineering process for the IoT. Traditional methods contain many redundant tasks and expose the applications to bugs, as they fail to consider the inherent features of the IoT (e.g., heterogeneity, interoperability, smart scenarios). MDE helps unify networking concepts at a higher-level, thus avoiding the low-level heterogeneity. MDE should be followed at least to enable networking based on unified concepts. By relying on these unified concepts, we can enable interoperability and smart scenarios.

— **RQ2: How can MDE enable interoperability between heterogeneous things?** By elevating the level of abstraction, MDE permits to avoid the low-level heterogeneity, that is the leading cause of interoperability issues. At the model-level, we erase this heterogeneity by creating an abstract platform to specify the business logic of the IoT application, and we offer a code generation process to reproduce this application with the low-level (heterogeneous) concepts.

— **RQ3: How can MDE provide effective control mechanisms to regulate the behavior of a network of heterogeneous things?** The proposed model-based networking concepts offer a unified platform enabling the specification the controls and their enforcement during code generation with a specific strategy.

- **RQ4: What are the qualitative and quantitative benefits of using a MDE methodology to design and implement a network of heterogeneous things?** CY-DSL provides a centralized interface for the engineering process, while the code generator exempts the engineer from manual work, by automating the generation of a significant amount of LoC. As this process is rigorous, traceable, and automatic, it helps save time and avoid buggy IoT applications.

Finally, Section 6.1 lists and describes the main contributions, and Section 6.2 presents some potential perspectives to enhance this work.

6.1 Contributions of the thesis

The present thesis advocates for an integrated model-based software engineering methodology to design and deploy a network of things. Thus, we consider that a first contribution is a novel approach to designing IoT applications. The second consists of the networking abstractions to help wire heterogeneous things. The third consists of the control abstractions and mechanisms to help define constraints on the network. The fourth consists of a code generator, based on MT, to interpret the model and generate the artifacts.

These contributions are implemented within an open-source project available on Github, called CyprIoT [1]. To the best of our knowledge, this is the first EMF-based tool that covers the networking side of the IoT with state-of-the-art MDE techniques.

In the following subsections, we explain how each contribution is valuable from a research perspective and how it can be extended in some future work.

6.1.1 Contribution 1: Software Engineering Methodology

Software engineering for the IoT inherited the tools and methods of classical applications. The IoT brings about new challenges that need to be addressed with a dedicated approach. The first contribution provides a methodology that has a more transparent development cycle for the IoT and where responsibilities are separated according to the layers of the MDA reference model. The outcome of this contribution is a software engineering procedure for the IoT.

1. https://github.com/atlanmod/CyprIoT/

6.1.2 Contribution 2: Model-Based Network Abstractions

The ability to create a network of things without being distracted by the technical details is pivotal in the context of the IoT, where connectivity and collaboration of things are highly desirable. The premises of the IoT presumes that any thing needs to connect. Still, in reality, because of the technological limitations, each range of things has its proper way to connect, creating an interoperability problem. We fill this gap by elevating the abstraction level at the model-level to provide more inclusive networking concepts. So, this contribution's outcome is the ability to specify a network of things in a unified manner.

6.1.3 Contribution 3: Model-Based Control Abstractions

The ability to control the network is also crucial in the context of the IoT. On the one hand, we provide abstractions to control the communications flow, and on the other hand, we give abstractions to create smart scenarios. The latter constitutes a defining feature of the IoT as it permits to condition the network according to the contextual factors such as the behavior (e.g., things states) and time. So, the outcome of this contribution is the ability to specify rules to control the network.

6.1.4 Contribution 4: Code Generator

The separation of the specification from the implementation eases the network's design and the automatic code generation of the low-level code. The code generator is based on MTs to wire the things in the form of a network, enforce the policies, and generate the low-level code. So, this contribution's outcome is the ability to generate the artifacts of a network of things from a model.

6.2 Perspectives

This work is far from exhaustive; it presents what we believe to be a scalable strategy in the long run to build robust IoT applications. In the course of our research, we identified some potential research directions that we could not address, generally because of time or resource limitations.

6.2.1 Model-Driven Reverse Engineering

The first potential line of research consists of taking full advantage of Model-Driven Reverse Engineering (MDRE), by reverse engineering things that are programmed using traditional software engineering. It may play a vital role in the adoption of our methodology.

This work could take place in two stages; the first stage consists of implementing a procedure for the reverse engineering of the communication ports. This procedure converts the low-level concepts towards the model-based ones. For instance, identifying an MQTT client and mapping it to concepts such as a channel, port, and protocol in ThingML syntax. The procedure enables us to wire the reversed ports with any other port in the CY-Model, thus avoiding interoperability issues at low-level and allowing us to control their communications with a policy. The second stage consists of implementing a more advanced procedure to understand the low-level behavior and reverse engineer it into a model-based one based on a statechart. Consequently, enabling to includes these things into smart scenarios.

Applying MDRE may have a high price in terms of time investment and research hazards to find the right approach. But, it will undoubtedly ease the design, adoption, interoperability, and control that may, in the opposite, save time.

6.2.2 Better Artifacts Generation

The second potential line of research consists of enriching the code generation process. At its current state, this process can only produce a network of connected thing, with the possibility to enforce policies, and generate template-based textual artifacts, by interpreting the model.

We could explore ways to support the generation of textual artifacts in natural language. This feature may be useful to exempt the manual writing of any human-readable artifact, such as documentation or a security assessment report.

Moreover, as the process is traceable because of model transformation, we could also create a tool to verify the generated artifacts' correctness w.r.t. the model. In that respect, applying the theory behind Verdi [168], based on network semantics, may be a promising start.

6.2.3 Simulation

The third potential line of research consists of simulating a network of things, before deployment (e.g., generating C code for all things to simulate the network on a Linux machine). This work may be useful to test the network's scalability and performance at a lower cost (e.g., predicting the power consumption and testing the probability of a node becoming malicious).

Also, as the model contains the primary data about the network, one can aggregate this data using an advanced technique such as machine learning to predict various aspects of the network. These predictions could enable us to define better strategies for deployment, especially when including resource-constrained ones.

BIBLIOGRAPHY

[1] B. L. R. Stojkoska and K. V. Trivodaliev, “A review of Internet of Things for smart home: Challenges and solutions,” *Journal of Cleaner Production*, vol. 140, pp. 1454–1464, jan 2017.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] M. Hussein, S. Li, and A. Radermacher, “Model-driven Development of Adaptive IoT Systems,” in *MODELS (Satellite Events)*, 2017, pp. 17–23.

[7] 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*, vol. 18, no. 4, pp. 2633–2653, jul 2018.

[8] S. C. Ergen, “ZigBee/IEEE 802.15. 4 Summary,” *UC Berkeley, September*, vol. 10, no. 17, p. 11, 2004.

[9] A. B. Sandberg and I. Crnkovic, “Meeting Industry-Academia Research Collaboration Challenges with Agile Methodologies,” in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, May 2017, pp. 73–82.

[10] M. B. Yassein, W. Mardini, and A. Khalil, "Smart homes automation using Z-wave protocol," in *2016 International Conference on Engineering & MIS (ICEMIS)*. IEEE, 2016, pp. 1–6.

[11] R. W. Picard and J. Healey, "Affective wearables," *Personal Technologies*, vol. 1, no. 4, pp. 231–240, dec 1997.

[12] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of Things for Smart Cities," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22–32, feb 2014.

[13] M. Fell and H. Melin, "The emerging Internet of Things," *Carré & Strauss*, 2014.

[14] G. Omale, "Gartner Identifies Top 10 Strategic IoT Technologies and Trends," *Gartner website*, 2018, [retrieved 2021-04-29]. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2018-11-07-gartner-identifies-top-10-strategic-iot-technologies-and-trends

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

[16] R. Kim and S. Poslad, "The Thing With E. coli: Highlighting Opportunities and Challenges of Integrating Bacteria in IoT and HCI," *arXiv preprint arXiv:1910.01974*, 2019.

[17] Z. Shelby, K. Hartke, and C. Bormann, *The constrained application protocol (CoAP)*, 2014, [retrieved 2021-04-29]. [Online]. Available: https://iottestware.readthedocs.io/en/master/coap_rfc.html

[18] F. Zambonelli, "Key Abstractions for IoT-Oriented Software Engineering," *IEEE Software*, vol. 34, no. 1, pp. 38–45, jan 2017.

[19] D. Spinellis, "Software-Engineering the Internet of Things," *IEEE Software*, vol. 34, no. 1, pp. 4–6, jan 2017.

[20] 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.

[21] 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.

[22] Q. Zhu, R. Wang, Q. Chen, Y. Liu, and W. Qin, "IoT gateway: Bridgingwireless sensor networks into Internet of Things," in *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on.* IEEE, 2010, pp. 347–352.

[23] 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.

[24] 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, pp. 508–513.

[25] 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.* Cham: Springer International Publishing, 2017, pp. 20–35.

[26] 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.

[27] 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.

[28] Trend Micro, "TrendLabs Security Intelligence BlogPersirai: New Internet of Things (IoT) Botnet Targets IP Cameras - TrendLabs Security Intelligence Blog," [retrieved 2021-04-29]. [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/persirai-new-internet-things-iot-botnet-targets-ip-cameras/

[29] N. Woolf, "DDoS attack that disrupted internet was largest of its kind in history, experts say," 2016, [retrieved 2021-04-29]. [Online]. Available: https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet

[30] J. Pescatore and G. Shpantzer, "Securing the Internet of Things survey," *SANS Institute*, pp. 1–22, 2014, [retrieved 2021-04-29]. [Online]. Available: https://www.sans.org/reading-room/whitepapers/covert/paper/34785

[31] G. Booch, J. Rumbaugh, and I. Jacobson, *The unified modeling language reference manual*, 1999, [retrieved 2021-04-29]. [Online]. Available: https://personal.utdallas.edu/~chung/Fujitsu/UML_2.0/Rumbaugh--UML_2.0_Reference_CD.pdf

[32] 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.

[33] 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.

[34] G. Blair, N. Bencomo, and R. B. France, "Models@ run. time," *Computer*, vol. 42, no. 10, 2009.

[35] O. Mavropoulos, H. Mouratidis, A. Fish, and E. Panaousis, "ASTo: A tool for security analysis of IoT systems," in *Software Engineering Research, Management and Applications (SERA), 2017 IEEE 15th International Conference on*. IEEE, 2017, pp. 395–400.

[36] R. F. Baumeister and M. R. Leary, "The need to belong: Desire for interpersonal attachments as a fundamental human motivation," *Psychological Bulletin*, vol. 117, no. 3, p. 497529, 1995.

[37] D. L. Mills and H.-W. Braun, "The NSFNET backbone network," in *Proceedings of the ACM workshop on Frontiers in computer communications technology*, 1987, pp. 191–196.

[38] R. Schaller, "Moore's law: past, present and future," *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, jun 1997.

[39] K. Ashton *et al.*, "That Internet of Things thing," *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.

[40] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth, "Machine learning for Internet of Things data analysis: A survey," *Digital Communications and Networks*, vol. 4, no. 3, pp. 161–175, 2018.

[41] D. J. Cook, J. C. Augusto, and V. R. Jakkula, "Ambient intelligence: Technologies, applications, and opportunities," *Pervasive and Mobile Computing*, vol. 5, no. 4, pp. 277–298, aug 2009.

[42] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad hoc networks*, vol. 10, no. 7, pp. 1497–1516, 2012.

[43] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE communications surveys & tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[44] 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.

[45] 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*. Singapore: Springer Singapore, 2016, pp. 619–624.

[46] A. Ghosh, D. Chakraborty, and A. Law, "Artificial intelligence in Internet of things," *CAAI Transactions on Intelligence Technology*, vol. 3, no. 4, pp. 208–218, 2018.

[47] E. Borgia, "The Internet of Things vision: Key features, applications and open issues," *Computer Communications*, vol. 54, pp. 1–31, dec 2014.

[48] S. M. R. Islam, D. Kwak, M. H. Kabir, M. Hossain, and K.-S. Kwak, "The Internet of Things for Health Care: A Comprehensive Survey," *IEEE Access*, vol. 3, pp. 678–708, 2015.

[49] R. Roman, P. Najera, and J. Lopez, "Securing the Internet of Things," *Computer*, vol. 44, no. 9, pp. 51–58, sep 2011.

[50] Z. Bi, L. D. Xu, and C. Wang, "Internet of Things for Enterprise Systems of Modern Manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1537–1546, may 2014.

[51] R. Want, B. N. Schilit, and S. Jenson, "Enabling the Internet of Things," *Computer*, vol. 48, no. 1, pp. 28–35, jan 2015.

[52] G. Kortuem, F. Kawsar, V. Sundramoorthy, and D. Fitton, "Smart objects as building blocks for the Internet of Things," *IEEE Internet Computing*, vol. 14, no. 1, pp. 44–51, jan 2010.

[53] O. B. Sezer, E. Dogdu, and A. M. Ozbayoglu, “Context-Aware Computing, Learning, and Big Data in Internet of Things: A Survey,” *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 1–27, feb 2018.

[54] 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.

[55] G. Fortino, A. Guerrieri, and W. Russo, “Agent-oriented smart objects development,” in *Proceedings of the 2012 IEEE 16th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, may 2012, pp. 907–912.

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

[57] C. Doukas, *Building Internet of Things with the ARDUINO*. North Charleston,SC, USA: CreateSpace Independent Publishing Platform, 2012.

[58] 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, 2015, pp. 131–134.

[59] 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, 2015, pp. 159–164.

[60] M. R. Palattella, M. Dohler, A. Grieco, G. Rizzo, J. Torsner, T. Engel, and L. Ladid, “Internet of Things in the 5G era: Enablers, architecture, and business models,” *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 3, pp. 510–527, 2016.

[61] V. Casola, L. D’Onofrio, G. Di Lorenzo, and N. 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*. Springer, 2010, pp. 249–263.

[62] S. Schmid, A. Bröring, D. Kramer, S. Käbisch, A. Zappa, M. Lorenz, Y. Wang, A. Rausch, and L. Gioppo, “An architecture for interoperable IoT Ecosystems,” in

International Workshop on Interoperability and Open-Source Solutions. Springer, 2016, pp. 39–55.

[63] I. Gojmerac, P. Reichl, I. P. Žarko, and S. Soursos, "Bridging IoT islands: the symbIoTe project," *e & i Elektrotechnik und Informationstechnik*, vol. 133, no. 7, pp. 315–318, 2016.

[64] G. Fortino, C. Savaglio, C. E. Palau, J. S. de Puga, M. Ganzha, M. Paprzycki, M. Montesinos, A. Liotta, and M. Llop, "Towards multi-layer interoperability of heterogeneous IoT platforms: The INTER-IoT approach," in *Integration, interconnection, and interoperability of IoT systems*. Cham: Springer, 2018, pp. 199–232.

[65] A. Bröring, S. Schmid, C.-K. Schindhelm, A. Khelil, S. Käbisch, D. Kramer, D. Le Phuoc, J. Mitic, D. Anicic, and E. Teniente, "Enabling IoT ecosystems through platform interoperability," *IEEE software*, vol. 34, no. 1, pp. 54–61, 2017.

[66] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, may 2005.

[67] S. Kent, "Model Driven Engineering," in *Integrated Formal Methods*, M. Butler, L. Petre, and K. Sere, Eds. Springer Berlin Heidelberg, 2002, pp. 286–298.

[68] J. M. Siegel, *Model driven architecture (MDA)-MDA Guide rev. 2.0*, 2014.

[69] O. M. Group, *Meta Object Facility (MOF) Specification*, 2000.

[70] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. London, UK: Pearson Education, 2008.

[71] D. Berardi, D. Calvanese, and G. De Giacomo, "Reasoning on UML class diagrams," *Artificial intelligence*, vol. 168, no. 1-2, pp. 70–118, 2005.

[72] M. Dumas and A. H. Ter Hofstede, "UML activity diagrams as a workflow specification language," in *International conference on the unified modeling language*. Springer, 2001, pp. 76–90.

[73] A. Alsaadi, "The UML Communication Diagram Revisited," in *International Conference on Software Engineering Advances (ICSEA 2007)*. IEEE, 2007, pp. 28–28.

[74] M. A. Gamboa and E. Syriani, "Automating Activities in MDE Tools," in *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development*. SCITEPRESS - Science and and Technology Publications, 2016, pp. 123–133.

[75] M. Eysholdt and H. Behrens, “Xtext: implement your language faster than the quick and dirty way,” in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2010, pp. 307–309.

[76] JetBrains, “MPS: Meta Programming System,” 2015, [retrieved 2021-04-29]. [Online]. Available: https://www.jetbrains.com/mps/

[77] L. C. Kats and E. Visser, “The spoofax language workbench: rules for declarative specification of languages and IDEs,” in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2010, pp. 444–463.

[78] 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.

[79] M. Lawley and J. Steel, “Practical declarative model transformation with Tefkat,” in *Satellite Events at the MoDELS 2005 Conference*, J.-M. Bruel, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 139–150.

[80] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, “Weaving executability into object-oriented meta-languages,” in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2005, pp. 264–278.

[81] F. Jouault and I. Kurtev, “Transforming models with ATL,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3844 LNCS, 2006, pp. 128–138.

[82] E. F. Webdev, “Model-to-Model Transformation (MMT),” Jan 2013, [retrieved 2021-04-29]. [Online]. Available: https://projects.eclipse.org/projects/modeling.mmt

[83] O. M. Group, *Meta object facility (MOF) 2.0 query/view/transformation specification*, 2005, [retrieved 2021-04-29]. [Online]. Available: https://www.omg.org/spec/QVT/1.1/PDF

[84] T. Mens and P. V. Gorp, “A Taxonomy of Model Transformation,” *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, mar 2006.

[85] J. Oldevik, *MOFScript user guide*, 2006, [retrieved 2021-04-29]. [Online]. Available: http://umt-qvt.sourceforge.net/mofscript/docs/MOFScript-User-Guide.pdf

[86] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire, *Acceleo user guide*, 2006, [retrieved 2021-04-29]. [Online]. Available: https://wiki.eclipse.org/Acceleo/User_Guide

[87] Ramamoorthy, Prakash, W.-T. Tsai, and Usuda, "Software Engineering: Problems and Perspectives," *Computer*, vol. 17, no. 10, pp. 191–209, oct 1984.

[88] H. Jeffrey, "Addressing the essential difficulties of software engineering," *Journal of Systems and Software*, vol. 32, no. 2, pp. 157–179, feb 1996.

[89] S. Gerard, J.-P. Babau, and J. Champeau, *Model driven engineering for distributed real-time embedded systems.* United Kingdom: Wiley-IEEE Press, 2010.

[90] 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.

[91] B. Morin, N. Harrand, and F. Fleurey, "Model-based software engineering to tame the IoT jungle," *IEEE Software*, vol. 34, no. 1, pp. 30–36, 2017.

[92] A. Vasilevskiy, B. Morin, Ø. Haugen, and P. Evensen, "Agile development of home automation system with ThingML," in *Industrial Informatics (INDIN), 2016 IEEE 14th International Conference on.* IEEE, 2016.

[93] A. Muelder, "Yakindu," [retrieved 2021-04-29]. [Online]. Available: https://www.itemis.com/en/yakindu/state-machine/

[94] Eclipse, "Eclipse Vorto - IoT toolset for standardized device descriptions," [retrieved 2021-04-29]. [Online]. Available: https://www.eclipse.org/vorto/

[95] G. Fuchs and R. German, "UML2 activity diagram based programming of wireless sensor networks," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications.* ACM, 2010, pp. 8–13.

[96] R. B. Smith, "SPOTWorld and the Sun SPOT," in *Proceedings of the 6th international conference on Information processing in sensor networks.* ACM, 2007, pp. 565–566.

[97] 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.

[98] 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. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 97–129.

[99] F. Ciccozzi, I. Crnkovic, D. D. Ruscio, I. Malavolta, P. Pelliccione, and R. Spalazzese, "Model-Driven Engineering for Mission-Critical IoT Systems," *IEEE Software*, vol. 34, no. 1, pp. 46–53, jan 2017.

[100] 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.

[101] K. K. Sandhu, "Specification and description language (SDL)," in *IEE Tutorial Colloquium on Formal Methods and Notations Applicable to Telecommunications*. IET, 1992, pp. 3–1.

[102] P. Levis and D. Gay, *TinyOS programming*. Cambridge, UK: Cambridge University Press, 2009.

[103] 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.

[104] K. Kreuzer *et al.*, "OpenHAB-empowering the smart home," *Openhab. org, Tech. Rep.*, 2013.

[105] M. Amrani, F. Gilson, A. Debieche, and V. Englebert, "Towards User-centric DSLs to Manage IoT Systems," in *MODELSWARD*, 2017, pp. 569–576.

[106] B. Bertran, J. Bruneau, D. Cassou, N. Loriant, E. Balland, and C. Consel, "DiaSuite: A tool suite to develop Sense/Compute/Control applications," *Science of Computer Programming*, vol. 79, pp. 39–51, 2014.

[107] D. Cassou, E. Balland, C. Consel, and J. Lawall, "Leveraging software architectures to guide and verify the development of sense/compute/control applications," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 431–440.

[108] N. Glombitza, D. Pfisterer, and S. Fischer, "Using state machines for a model driven development of web service-based sensor network applications," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*. ACM, 2010, pp. 2–7.

[109] ——, “Integrating wireless sensor networks into web service-based business processes,” in *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks - MidSens '09*. ACM Press, 2009.

[110] IBM Emerging Technologies, “Node-RED. A visual tool for wiring the Internet of Things,” 2016, [retrieved 2021-04-29]. [Online]. Available: http://nodered.org/

[111] J. Im, S. Kim, and D. Kim, “IoT mashup as a service: cloud-based mashup service for the Internet of Things,” in *Services Computing (SCC), 2013 IEEE International Conference on*. IEEE, 2013, pp. 462–469.

[112] M. Blackstock and R. Lea, “IoT mashups with the WoTKit,” in *Internet of Things (IoT), 2012 3rd International Conference on the*. IEEE, 2012, pp. 159–166.

[113] S. Heo, S. Woo, J. Im, and D. Kim, “IoT-MAP: IoT mashup application platform for the flexible IoT ecosystem,” in *Internet of Things (IoT), 2015 5th International Conference on the*. IEEE, 2015, pp. 163–170.

[114] 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.

[115] M. M. Casalino, H. Plate, and S. Trabelsi, “Transversal Policy Conflict Detection,” in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 30–37.

[116] D. Basin, M. Clavel, and M. Egea, “A decade of model-driven security,” in *Proceedings of the 16th ACM symposium on Access control models and technologies*. ACM, 2011, pp. 1–10.

[117] U. Lang and R. Schreiner, “Proximity-Based Access Control (PBAC) using Model-Driven Security,” in *ISSE 2015*, H. Reimer, N. Pohlmann, and W. Schneider, Eds. Wiesbaden: Springer Fachmedien Wiesbaden, 2015, pp. 157–170.

[118] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo, and J. Voas, “Attribute-based access control,” *Computer*, vol. 48, no. 2, pp. 85–88, 2015.

[119] T. Lodderstedt, D. Basin, and J. Doser, “SecureUML: A UML-Based Modeling Language for Model-Driven Security,” in *≪UML≫ 2002 — The Unified Modeling Language*. Springer Berlin Heidelberg, 2002, pp. 426–441.

[120] D. Ferraiolo, D. R. Kuhn, and R. Chandramouli, *Role-based access control.* United States: Artech House Publishers, 2007.

[121] S. Martínez, J. Garcia-Alfaro, F. Cuppens, N. Cuppens-Boulahia, and J. Cabot, "Model-Driven Extraction and Analysis of Network Security Policies," in *Model-Driven Engineering Languages and Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 52–68.

[122] R. Neisse, G. Steri, I. N. Fovino, and G. Baldini, "SecKit: A Model-based Security Toolkit for the Internet of Things," *Computers & Security*, vol. 54, pp. 60–76, oct 2015.

[123] S. Sicari, A. Rizzardi, L. Grieco, G. Piro, and A. Coen-Porisini, "A policy enforcement framework for Internet of Things applications in the smart health," *Smart Health*, vol. 3-4, pp. 39–74, sep 2017.

[124] O. X. T. Committee *et al.*, *eXtensible access control markup language (XACML) Version 3.0*, OASIS, 2013, [retrieved 2021-04-29]. [Online]. Available: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf

[125] A. L. Marra, F. Martinelli, P. Mori, and A. Saracino, "Implementing Usage Control in Internet of Things: A Smart Home Use Case," in *2017 IEEE Trustcom/BigDataSE/ICESS.* IEEE, 2017, pp. 1056–1063.

[126] D. Ferraiolo, R. Chandramouli, R. Kuhn, and V. Hu, "Extensible access control markup language (XACML) and next generation access control (NGAC)," in *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control.* ACM Press, 2016, pp. 13–24.

[127] S. Hada and M. Kudo, *XML access control language (XACL): Provisional authorization for XML documents*, 2001.

[128] S. Reiff-Marganiec, L. Blair, and K. J. Turner, *APPEL: the ACCENT project policy environment/language*, 2005.

[129] L. F. Cranor, "P3P: Making privacy policies more useful," *IEEE Security & Privacy*, vol. 1, no. 6, pp. 50–55, 2003.

[130] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter, "Enterprise privacy authorization language (EPAL)," *IBM Research*, vol. 30, p. 31, 2003.

[131] H. F. Atlam, M. O. Alassafi, A. Alenezi, R. J. Walters, and G. B. Wills, "XACML for Building Access Control Policies in Internet of Things," in *Proceedings of the*

3rd International Conference on Internet of Things, Big Data and Security. SCITEPRESS - Science and Technology Publications, 2018.

[132] C. Buschmann and D. Pfisterer, “iSense: A modular hardware and software platform for wireless sensor networks,” *6. Fachgespräch Sensornetzwerke*, p. 15, 2007.

[133] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend.* United Kingdom: Packt Publishing Ltd, 2016.

[134] C. E. Shannon, “A mathematical theory of communication,” *Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[135] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, “Internet of Things (IoT): A vision, architectural elements, and future directions,” *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013, including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services, Cloud Computing and Scientific Applications Big Data, Scalable Analytics, and Beyond.

[136] A. Rayes and S. Salam, “The Internet in IoT—OSI, TCP/IP, IPv4, IPv6 and Internet Routing,” in *Internet of Things From Hype to Reality.* Cham: Springer International Publishing, oct 2016, pp. 35–56.

[137] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, jun 2003.

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

[139] A. Banks and R. Gupta, *MQTT Version 3.1.1*, 2014, [retrieved 2021-04-29]. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf

[140] 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.

[141] K. Czarnecki and S. Helsen, “Classification of model transformation approaches,” in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003, pp. 1–17.

[142] H. Bruneliere, F. M. de Kerchove, G. Daniel, S. Madani, D. Kolovos, and J. Cabot, “Scalable model views over heterogeneous modeling technologies and resources,” *Software and Systems Modeling*, vol. 19, no. 4, pp. 827–851, 2020.

[143] 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 Symposium on Applied Computing*, ser. SAC ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 832841.

[144] W. Snipes, E. Murphy-Hill, T. Fritz, M. Vakilian, K. Damevski, A. R. Nair, and D. Shepherd, “Chapter 5 - a practical guide to analyzing ide usage data,” in *The Art and Science of Analyzing Software Data*, C. Bird, T. Menzies, and T. Zimmermann, Eds. Boston: Morgan Kaufmann, 2015, pp. 85–138.

[145] C. Ebert and C. Jones, “Embedded software: Facts, figures, and future,” *Computer*, vol. 42, no. 4, pp. 42–52, 2009.

[146] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, “Early detection of configuration errors to reduce failure damage,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’16. Savannah, GA, USA: USENIX Association, 2016, p. 619634.

[147] A. Sukhoo, A. Barnard, M. M. Eloff, J. A. Van der Poll, and M. Motah, “Accommodating soft skills in software project management,” *Issues in Informing Science & Information Technology*, vol. 2, pp. 691–703, 2005.

[148] F. Ahmed, L. F. Capretz, and P. Campbell, “Evaluating the demand for soft skills in software development,” *It Professional*, vol. 14, no. 1, pp. 44–49, 2012.

[149] 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, dec 2011.

[150] 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, M. Mazzara, I. Ober, and G. Salaün, Eds. Springer, 2018, pp. 644–655.

[151] 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.

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

[153] 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.

[154] 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.

[155] G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation*. The Netherlands: World Scientific Publishing Company, 1999.

[156] OMG, *Business Process Modeling Notation, Final Adopted Specification, Version 1.0*, 2006.

[157] A. Eclipse IoT Working Group, IEEE and IoT Council, “IoT Developer Survey 2020,” 2020, [retrieved 2021-04-29]. [Online]. Available: https://outreach.eclipse.foundation/eclipse-iot-developer-survey-2020

[158] V. Albino, U. Berardi, and R. M. Dangelico, “Smart cities: Definitions, dimensions, performance, and initiatives,” *Journal of urban technology*, vol. 22, no. 1, pp. 3–21, 2015.

[159] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, “Industry 4.0,” *Business & information systems engineering*, vol. 6, no. 4, 2014.

[160] M. Noura, M. Atiquzzaman, and M. Gaedke, “Interoperability in Internet of Things: Taxonomies and open challenges,” *Mobile Networks and Applications*, vol. 24, no. 3, pp. 796–809, 2018.

[161] K. K. Patel, S. M. Patel *et al.*, “Internet of Things-IoT: definition, characteristics, architecture, enabling technologies, application & future challenges,” *International journal of engineering science and computing*, vol. 6, no. 5, 2016.

[162] J. Manyika, M. Chui, P. Bisson, J. Woetzel, R. Dobbs, J. Bughin, and D. Aharon, "Unlocking the Potential of the Internet of Things," *McKinsey Global Institute*, 2015, [retrieved 2021-04-29]. [Online]. Available: https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world

[163] J. K. D. Barriga, C. D. G. Romero, and J. I. R. Molano, "Proposal of a standard architecture of IoT for Smart Cities," in *International Workshop on Learning Technology for Education Challenges*, L. Uden, D. Liberona, and B. Feldmann, Eds. Cham: Springer, 2016, pp. 77–89.

[164] S. K. Datta, R. P. F. Da Costa, C. Bonnet, and J. Härri, "oneM2M architecture based IoT framework for mobile crowd sensing in smart cities," in *2016 European conference on networks and communications (EuCNC)*. IEEE, 2016, pp. 168–173.

[165] 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, pp. 91–100.

[166] 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, 2008, pp. 432–443.

[167] E. Reiter and R. Dale, *Building natural language generation systems*. Cambridge, UK: Cambridge university press, 2000.

[168] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," *SIGPLAN Not.*, vol. 50, no. 6, p. 357368, Jun. 2015.

Appendix A

Xtext Grammar for the Networking Language

Listing A.1 – Xtext Grammar for the Networking Language. This grammar provides the language to create the network model in a textual form.

```
grammar org.atlanmod.cypriot.Cypriot with org.eclipse.xtext.common.Terminals

generate cyprIoT "http://www.atlanmod.org/CyprIoT"
import "http://www.thingml.org/xtext/ThingML" as thingml

CyprIoTModel returns CyprIoTModel:
    imports+=Import*
    (declareTime+=Time |
    declareThings+=TypeThing |
    declareChannels+=TypeChannel|
    specifyNetworks+=Network |
    declareUsers+=User |
    specifyPolicies+=Policy |
    declareRoles+=Role)*;

Import:
    'import' importURI=STRING
;

NamedElement:
    User | Role | TypeThing | Network | Path | TypeChannel |
    InstanceThing | InstanceChannel |
    Bind | Policy | ChannelToBind | Rule | ThingAny | ChannelAny | SubjectOther |
        ObjectOther |
    ";" name=ID
;

```

```
Role:
    'role' name=ID
;

User :
    'user' name=ID(':' password=STRING)? ('assigned' assignedRoles+=[Role] ( ","
        assignedRoles+=[Role])*)?
;

TypeThing:
    'thing' name=ID
        ('import' importPath=STRING)?
        ('assigned' assignedRoles+=[Role] ( "," assignedRoles+=[Role])*)?
;

TypeChannel:
    'channel' name=ID '{'
        (hasPaths+=Path)*
    '}'
;

Path:
    'path' name=ID ('=' customName=STRING)? ('(' acceptedMessage=[thingml::Message] (':
        ' serializer=Serializer)? ')')? ('fork' fork+=[Path])?
;

enum Serializer:
    JSON='JSON' |
    BINARY='BINARY'
;

Network:
    'network' name=ID '{'
        domain=Domain
        (hasPolicyEnforcement=PoliciesEnforcement)?
        (instantiate+=Instance |
        hasBinds+=Bind |
        hasForwards+=NetworkForward
        )*
    '}'
;

Domain:
    'domain' name=DomainId
;
```

```

DomainId:
    VALIDID (=>'.' VALIDID)+
;

PoliciesEnforcement :
    'enforce' hasEnforcedPolicies+=[Policy] (strategy+=EnforcementStrategies)? ( ","
        hasEnforcedPolicies+=[Policy] (strategy+=EnforcementStrategies)?)*
;

enum EnforcementStrategies:
    BESTEFFORT='Best-Effort' |
    DENYFIRST='Deny-First' |
    ALLOWFIRST='Allow-First' |
;

NetworkForward:
    'forward' (name=ID ':')? forwardBind=[Bind] 'to' forwardToChannel=ChannelToBind
;

Instance:
    InstanceThing | InstanceChannel
;

InstanceThing :
    'instance' name=ID ':' instantiateTypeThing=ThingToInstantiate
;

InstanceChannel:
    'instance' name=ID ':' instantiateTypeChannel=ChannelToInstantiate
;

ThingToInstantiate :
    thingToInstantiate=[TypeThing] 'platform' targetedPlatform=Platform ('owner' owner=[
        User])?
;

ChannelToInstantiate :
    channelToInstantiate=[TypeChannel] 'protocol' targetedProtocol=PubSubProtocol ('(''
        server=' server=STRING')')?
;

Bind:
    'bind' (name=ID ':')? bindsInstanceThing=[InstanceThing] ('['thingPosition=INT']')? ('.'
        portToBind=[thingml::Port]) bindAction=BindAction channelToBind=ChannelToBind
```

```
;

enum Platform:
    POSIX='POSIX' |
    POSIXMT='POSIMT' |
    JAVA='JAVA' |
    ARDUINO='ARDUINO' |
    JAVASCRIPT='JAVASCRIPT' |
    GO='GO'
;

enum PubSubProtocol:
    MQTT='MQTT' |
    BLUETOOTH='BLUETOOTH' |
    HTTP='HTTP' |
    COAP='COAP' |
    ZIGBEE='ZIGBEE' |
    ZWAVE='ZWAVE'
;

enum BindAction:
    READ='<=' |
    WRITE='=>'
;

ChannelToBind:
     targetChannel=[InstanceChannel] '{' bindPaths+=[Path] ( "," bindPaths+=[Path])*'}'
;

IntLiteral:
    INT
;

VALIDID:
    ID;
```

Appendix B

Xtext Grammar for the Policy Language

Listing B.1 – Xtex grammar for the Policy Language; continuation of Listing A.1. This grammar provides the language to specify the policy.

```
Policy:
    'policy' name=ID '{'
        (hasRules+=Rule)*
    '}'
;

Rule:
    (RuleComm | RuleTrigger) hasCondition=Conditions
;

RuleComm:
    'rule' (name=ID ':')? hasCommSubject=CommSubject setTypeComm=TypeComm
        hasCommObject=CommObject
;

TypeComm:
    (deny?='deny:' | allow?='allow:') actionComm=ActionComm
;

enum ActionComm:
    send='send' | receive='receive' | sendreceive='send-receive'
;

Conditions:
    'when' hasTime=Time
;

```

```
Time:
    'time' name=ID ':'hasExpression=CronExpression
;

RuleTrigger:
    'rule' (name=ID ':')? thingWithState=ThingWithState setTypeTrigger=TypeTrigger
;

TypeTrigger:
    'trigger:' setActionToTrigger=ActionTrigger
;

ActionTrigger:
    (goToState='goToState' triggerGoToState=ThingWithState) |
    (executeFunction='executeFunction' triggerFunction=ThingWithFunction)
;

CommObject :
    hasThingWithStateOrPort=ThingWithStateOrPort | hasOtherObject=[ObjectOther]
;

ObjectOther:
    Role | User | ThingAny | Path | TypeChannel
;

CommSubject :
    hasThingWithStateOrPort=ThingWithStateOrPort | hasOtherSubject=[SubjectOther]
;

ThingAny:
    InstanceThing | TypeThing
;

ChannelAny:
    InstanceChannel | TypeChannel
;

SubjectOther:
    Role | User | ThingAny
;

ThingWithStateOrPort:
    ThingWithPort | ThingWithState
;

```

```
ThingWithPort:
    thing=[ThingAny] '->' getPort=GetPort
;

ThingWithState:
    thing=[ThingAny] '->' getState=GetState
;

ThingWithFunction:
    thing=[ThingAny] '->' getFunction=GetFunction
;

GetPort:
    'port:' port=[thingml::Port]
;

GetState:
    'state:' state=[thingml::State]
;

GetFunction:
    'function:' function=[thingml::Function] ('(' (parameters+=[thingml::Parameter]) (',' (
        parameters+=[thingml::Parameter]))* ')')?
;

CronExpression:
    seconds=CronElement minutes=CronElement hours=CronElement
    days=CronElement months=CronElement daysOfWeek=CronElement (year=CronElement)?
         | '@' constant=ID
;
CronElement:
    RangeCronElement | PeriodicCronElement
;
RangeCronElement hidden():
    TerminalCronElement ({RangeCronElement.start=current} '-' end=IntLiteral)*
;
TerminalCronElement:
    expression=(IntLiteral | ID | '*' | '?')
;
PeriodicCronElement hidden():
    expression=TerminalCronElement '/' elements=RangeCronList
;
RangeCronList hidden():
    elements+=RangeCronElement (',' elements+=RangeCronElement)*
;
```

Appendix C

ATL Rules for Networking and Forwarding Transformations

Listing C.1 – The ATL rules for networking

```
module Network2Thing;

create OUT: ThingML from TH: ThingML, CY : CyprIoT;

uses Copier;
uses Helpers;

helper def : mqttNumber : Integer = 0;


rule copyState {
    from s : ThingML!State(not (s.oclIsTypeOf(ThingML!CompositeState) or s.oclIsTypeOf(
        ThingML!FinalState)))
    to t : ThingML!State(
        annotations <- s.annotations,
        entry <- s.entry,
        exit <- s.exit,
        internal <- s.internal,
        name <- s.name,
        outgoing <- s.outgoing,
        properties <- s.properties
    )
}

rule createExternalConnectorFromBind {
    from s : CyprIoT!Bind(s.isBindMatchesInputThing())
    using {
        protocolName : String = s.getTargetedProtocolFromBind();
    }
    to
    configuration : ThingML!Configuration(
        name <— s.getInstanceThingNameFromBind()+'_Cfg',
        instances <— Sequence{instance},
        connectors <— Sequence{externalConnector},
        annotations <— Sequence{compiler,debug}
    ),
    compiler : ThingML!PlatformAnnotation(
        name <— 'compiler',
        value <— s.getTargetedPlatfomFromBind()
    ),
    debug : ThingML!PlatformAnnotation(
        name <— 'debug',
        value <— 'true'
    ),
    instance : ThingML!Instance(
        name <— s.bindsInstanceThing.name,
        type <— thisModule.inputThing()
    ),
    externalConnector : ThingML!ExternalConnector(
        inst <— instance,
        port <— ThingML!Port.allInstances()—>select(p | p.name=s.portToBind.name).first(),
        protocol <— protocol,
        annotations <— s.getAllPathsFromBind()—>collect(t | thisModule.multiplePaths(s, t))
    ),
    protocol : ThingML!Protocol (
        name <— protocolName,
        annotations <— Sequence{brokerAdress,portNumber,serializer}
    ),
    brokerAdress : ThingML!PlatformAnnotation(
        name <— 'mqtt_broker_address',
        value <— s.getServerFromBind()
    ),
    portNumber : ThingML!PlatformAnnotation(
        name <— 'mqtt_port_number',
        value <— s.getPortNumberFromBind()
    ),
    serializer : ThingML!PlatformAnnotation(
        name <— 'serializer',
        value <— s.getAllPathsFromBind().first().serializer.toString().removeHash().toLower()
            —— Create new protocol in ThingML when multiple type of serializers
    )
    do {
        if(thisModule.mqttNumber>=1){
            protocol.name <− protocolName+thisModule.mqttNumber.toString();
        }
        thisModule.mqttNumber <− thisModule.mqttNumber + 1;
    }
}

lazy rule multiplePaths {
    from s : CyprIoT!Bind, t : CyprIoT!Path
    to annotationMqtt : ThingML!PlatformAnnotation(
        name <− s.bindAction.toString().removeHash().transformArrowToMQTTSyntax,
        value <− s.getFullPathName(t)
    )
}
```

Listing C.2 – The ATL rules for forwarding.

```
module NetworkForward;

create OUT: ThingML from TH: ThingML, CY : CyprIoT;

uses Copier;
uses Helpers;

rule copyThingMLModel {
    from s : ThingML!ThingMLModel
    to t : ThingML!ThingMLModel(
        configs <− s.configs,
        imports <− s.imports,
        protocols <− if(thisModule.firstNetwork().hasForwards.first().bridgeSubject.oclIsTypeOf
            (CyprIoT!Bind) and thisModule.firstNetwork().hasForwards.first().bridgeSubject.
            oclAsType(CyprIoT!Bind).getInstanceThingNameFromBind()=thisModule.
            nameOfInputThing()) then CyprIoT!NetworkForward.allInstances()−>collect(b |
            thisModule.resolveTemp(b, 'proto'))−>union(s.protocols) else s.protocols endif ,
        types <− s.types
    )
}

rule copyThing {
    from s : ThingML!Thing
    to t : ThingML!Thing(
        name <− s.name,
        ports <− if(thisModule.firstNetwork().hasForwards.first().bridgeSubject.oclIsTypeOf(
            CyprIoT!Bind) and thisModule.firstNetwork().hasForwards.first().bridgeSubject.
            oclAsType(CyprIoT!Bind).getInstanceThingNameFromBind()=thisModule.
            nameOfInputThing()) then CyprIoT!NetworkForward.allInstances()−>collect(b |
            thisModule.resolveTemp(b, 'port'))−>union(s.ports) else s.ports endif,
        annotations <− s.annotations,
        assign <− s.assign,
        behaviour <− s.behaviour,
        fragment <− s.fragment,
        functions <− s.functions,
        messages <− s.messages,
        includes <− s.includes,
        properties <− s.properties
    )
}

rule copyConfig {
    from s : ThingML!Configuration
    to t : ThingML!Configuration(
        annotations <− s.annotations,
        connectors <− if(thisModule.firstNetwork().hasForwards.first().bridgeSubject.
            oclIsTypeOf(CyprIoT!Bind) and thisModule.firstNetwork().hasForwards.first().
            bridgeSubject.oclAsType(CyprIoT!Bind).getInstanceThingNameFromBind()=
            thisModule.nameOfInputThing()) then s.connectors−>union(CyprIoT!
            NetworkForward.allInstances()) else s.connectors endif ,
        instances <− s.instances,
        name <− s.name,
        propassigns <− s.propassigns
    )
}

rule copyState {
    from s : ThingML!State(not (s.oclIsTypeOf(ThingML!CompositeState) or s.oclIsTypeOf(
        ThingML!FinalState)))
    to t : ThingML!State(
        annotations <− s.annotations,
        entry <− s.entry,
        exit <− s.exit,
        internal <− s.internal,
        name <− s.name,
        outgoing <− s.outgoing,
        properties <− s.properties
    )
}
rule copyCompositeState {
    from s : ThingML!CompositeState
    to t : ThingML!CompositeState(
        name <- s.name,
        annotations <- s.annotations,
        entry <- s.entry,
        exit <- s.exit,
        history <- s.history,
        initial <- s.initial,
        internal <- s.internal,
        outgoing <- s.outgoing,
        properties <- s.properties,
        region <- s.region,
        session <- s.session,
        substate <- s.substate
    )
}

helper context CyprIoT!Network def : collectEnforcedPoliciesInNetwork() : Sequence(CyprIoT!
    Policy) = self.hasPolicyEnforcement.hasEnforcedPolicies;
helper context CyprIoT!Policy def : collectRuleBridgeFromPolicy() : Sequence(CyprIoT!
    RuleBridge) =
    self.hasRules->select(r | r.oclIsTypeOf(CyprIoT!RuleBridge))
;

helper context ThingML!ExternalConnector def : applyRuleBridge(annotation : ThingML!
    PlatformAnnotation) : Sequence(ThingML!PlatformAnnotation) = thisModule.firstNetwork
    ().collectEnforcedPoliciesInNetwork()->collect(p | p.collectRuleBridgeFromPolicy()->
    collect(r | if(r.isAnnotationPathMatchesRuleBridge(annotation)) then thisModule.
    multiplePaths(r.bridgeSubject.oclAsType(CyprIoT!ChannelWithPath).getPath.path,r,
    thisModule.firstNetwork()) else Sequence{} endif))->flatten()->union(self.annotations);

helper context CyprIoT!RuleBridge def : isAnnotationPathMatchesRuleBridge(annotation :
    ThingML!PlatformAnnotation) : Boolean = self.bridgeSubject.oclAsType(CyprIoT!
    ChannelWithPath).channel.oclIsTypeOf(CyprIoT!TypeChannel) and annotation.name='
    mqtt_publish_topic' and not(annotation.value=self.getFullPathNameBridge(thisModule.
    firstNetwork(),self.bridgeSubject.oclAsType(CyprIoT!ChannelWithPath).getPath.path)) and
    thisModule.firstNetwork().instantiate->exists(s | s.oclIsTypeOf(CyprIoT!InstanceChannel)
    and s.oclAsType(CyprIoT!InstanceChannel).typeChannel.pubSubToInstantiate.name=self.
    bridgeSubject.oclAsType(CyprIoT!ChannelWithPath).channel.name) and annotation.value=
    self.getFullPathNameBridge(thisModule.firstNetwork(),self.bridgeSubject.oclAsType(CyprIoT
    !ChannelWithPath).getPath.path);

```

```

rule copyExternalConnector {
    from s : ThingML!ExternalConnector
    to t : ThingML!ExternalConnector(
        annotations <- s.annotations->collect(a | s.applyRuleBridge(a))->flatten(),
        inst <- s.inst,
        name <- s.name,
        port <- s.port,
        protocol <- s.protocol
    )
}

rule copyTransition {
    from s : ThingML!Transition
    to t : ThingML!Transition(
        action <- if(thisModule.firstNetwork().hasForwards.first().bridgeSubject.oclIsTypeOf(
            CyprIoT!Bind)
                and thisModule.firstNetwork().hasForwards.first().bridgeSubject.oclAsType(
                    CyprIoT!Bind).getInstanceThingNameFromBind()=thisModule.
                    nameOfInputThing()
                and not(s.event.oclIsUndefined())
                and s.event.oclAsType(ThingML!ReceiveMessage).port.name=ThingML!Port.
                    allInstances().first().name
                and s.event.oclAsType(ThingML!ReceiveMessage).message.name=ThingML!
                    Port.allInstances().first().receives.first().name)
            then thisModule.groupActionTransition(s,s.event.oclAsType(ThingML!
                ReceiveMessage))
            else s.action endif,
        annotations <- s.annotations,
        event <- s.event,
        guard <- s.guard,
        name <- s.name,
        target <- s.target
    )
}
lazy rule groupActionTransition {
    from s : ThingML!Transition, r : ThingML!ReceiveMessage
    to
    groupAction : ThingML!ActionBlock(
        actions <- if(not(s.action.oclIsUndefined())) then Sequence{sendAction,s.action}
            else Sequence{sendAction} endif
    ),
    sendAction : ThingML!SendAction(
        message <-  ThingML!Port.allInstances().first().receives.first(),
        port <- thisModule.resolveTemp(CyprIoT!NetworkForward.allInstances().first(), 'port'),
        parameters <- Sequence{eventRef}
    ),
    eventRef : ThingML!EventReference(
        receiveMsg <- r,
        parameter <- r.message.parameters.first()
    )

}

rule bridgePaths {
    from s : CyprIoT!NetworkForward (s.bridgeSubject.oclIsTypeOf(CyprIoT!Bind) and s.
        bridgeSubject.oclAsType(CyprIoT!Bind).getInstanceThingNameFromBind()=thisModule.
        nameOfInputThing())
    to t : ThingML!ExternalConnector(
        annotations <- ThingML!ExternalConnector.allInstances().first().annotations->collect
            (a | thisModule.addAnnotations(a)),
        inst <- ThingML!ExternalConnector.allInstances().first().inst,
        port <- port,
        protocol <- proto
    ),
    proto : ThingML!Protocol(
        name <- s.bridgeToChannel.targetedChannelInstance.typeChannel.targetedProtocol.
            toString().removeHash()+'1',
        annotations <- Sequence{brokerAdress,portNumber,serializer}

    ),
    port : ThingML!RequiredPort(
        name <- 'bridge_'+s.bridgeSubject.oclAsType(CyprIoT!Bind).
            getInstanceThingNameFromBind()+'_'+s.bridgeSubject.oclAsType(CyprIoT!Bind).
            portToBind.name,
        sends <- ThingML!Port.allInstances().first().receives,
        optional <- true
    ),
    brokerAdress : ThingML!PlatformAnnotation(
        name <- 'mqtt_broker_address',
        value <- s.bridgeToChannel.targetedChannelInstance.typeChannel.server.toString().split
            (':').first()
    ),
    portNumber : ThingML!PlatformAnnotation(
        name <- 'mqtt_port_number',
        value <- s.bridgeToChannel.targetedChannelInstance.typeChannel.server.toString().split
            (':').last()
    ),
    serializer : ThingML!PlatformAnnotation(
        name <- 'serializer',
        value <— 'json' —— Create new protocol in ThingML when multiple type of serializers
    )
}

lazy rule addAnnotations {
    from s : ThingML!PlatformAnnotation
    to t : ThingML!PlatformAnnotation(
        name <— s.name.mirrorMqttPubSub(),
        value <— s.value
    )
}

lazy rule multiplePaths {
    from t : CyprIoT!Path, r : CyprIoT!RuleBridge, n : CyprIoT!Network
    to annotationMqtt : ThingML!PlatformAnnotation(
        name <— 'mqtt_publish_topic',
        value <— r.getFullPathNameBridge(n,t)
    )
}
```

Listing C.3 – The ATL helpers for networking and forwarding.

```
module Helpers;

create OUT: ThingML from TH: ThingML, CY : CyprIoT;

helper def : inputThing() : String = ThingML!Thing.allInstances().first();

helper def : nameOfInputThing() : String = thisModule.inputThing().name;

helper context String def : convertArrowToSendOrReceive() : String =
    if self.startsWith('#=>')
        then 'send'
    else
        'receive'
    endif
;

```

```
helper context String def : mirrorSendOrReceive() : String =
    if self.startsWith('send')
        then 'receive'
    else
        'send'
    endif
;

helper context String def : mirrorMqttPubSub() : String =
    if self.startsWith('mqtt_publish_topic')
        then 'mqtt_subscribe_topic'
    else
        'mqtt_publish_topic'
    endif
;

helper context String def : replaceDotsWithSlashInDomain() : String =
    self.replace('.', '/')
;

helper context String def : removeHash() : String =
    self.replaceAll('#', '')
;

helper context String def : transformArrowToMQTTSyntax : String =
    if self.startsWith('=>')
        then 'mqtt_publish_topic'
    else
        'mqtt_subscribe_topic'
    endif
;

helper context CyprIoT!Bind def : getPathName() : String =
    if(self.getPathFromBind().customName.oclIsUndefined())
        then self.getDomainFromBind().replaceDotsWithSlashInDomain()+'/'+self.
            channelToBind.targetedChannelInstance.name+'/'+self.getPathFromBind().name
    else
        self.getPathFromBind().customName
    endif
;

helper context CyprIoT!Bind def : getFullPathName(p : CyprIoT!Path) : String =
    if(p.customName.oclIsUndefined())
        then self.getDomainFromBind().replaceDotsWithSlashInDomain()+'/'+self.
            channelToBind.targetedChannelInstance.name+'/'+p.name
    else
        p.customName
    endif
;

helper context CyprIoT!RuleBridge def : getFullPathNameBridge(n : CyprIoT!Network,p :
    CyprIoT!Path) : String =
    if(p.customName.oclIsUndefined())
        then n.domain.name.replaceDotsWithSlashInDomain()+'/'+self.bridgeObject.oclAsType
            (CyprIoT!ChannelWithPath).channel.name+'/'+p.name
    else
        p.customName
    endif
;

helper context CyprIoT!Bind def : isBindMatchesInputThing() : Boolean =
    self.getInstanceThingNameFromBind()=thisModule.nameOfInputThing()
;

helper context CyprIoT!Bind def : getInstanceThingFromBind() : CyprIoT!InstanceThing =
    self.bindsInstanceThing
;

helper context CyprIoT!Bind def : getInstanceThingNameFromBind() : String =
    self.getInstanceThingFromBind().name
;

helper context CyprIoT!Bind def : getThingFromBind() : CyprIoT!TypeThing =
    self.bindsInstanceThing.typeThing.thingToInstantiate
;

helper context CyprIoT!Bind def : getThingNameFromBind() : String =
    self.getThingFromBind().name
;

helper context CyprIoT!Bind def : getTypeChannelFromBind() : CyprIoT!ChannelToInstanciate
    =
    self.channelToBind.targetedChannelInstance.typeChannel
;

helper context CyprIoT!Bind def : getServerFromBind() : String =
    self.getTypeChannelFromBind().server.toString().split(':').first()
;

helper context CyprIoT!Bind def : getPortNumberFromBind() : String =
    self.getTypeChannelFromBind().server.toString().split(':').last()
;

helper context CyprIoT!Bind def : getTargetedProtocolFromBind() : String =
    self.getTypeChannelFromBind().targetedProtocol.toString().removeHash()
;

helper context CyprIoT!Bind def : getDomainFromBind() : String =
    self.refImmediateComposite().oclAsType(CyprIoT!Network).domain.name
;

helper context CyprIoT!Bind def : getAllPathsFromBind() : Sequence(CyprIoT!Path) =
    self.channelToBind.paths
;

helper context CyprIoT!Bind def : getTargetedPlatfomFromBind() : String =
    self.bindsInstanceThing.typeThing.targetedPlatform.toString().removeHash().toLower()
;

helper def : firstNetwork() : CyprIoT!Network = CyprIoT!Network.allInstances().first();

helper def : bindOfInputThing() : CyprIoT!Bind = thisModule.firstNetwork().
    bindsContainingThingInNetwork().first();

helper context CyprIoT!Network def : bindsContainingThingInNetwork() : Sequence(CyprIoT!
    Bind) =
    self.hasBinds->select(b | b.isBindMatchesInputThing())
;
```

Appendix D

ATL Rules for Communication Control Transformations

Listing D.1 – The ATL rules for communication control

```
module RuleComm;

create OUT: ThingML from TH: ThingML, CY : CyprIoT;

uses HelpersComm;

rule copyExternalConnector {
    from s : ThingML!ExternalConnector
    to t : ThingML!ExternalConnector(
        annotations <— s.addAnnotationsAfterEnforcements(),
        inst <— s.inst,
        name <— s.name,
        port <— s.port,
        protocol <— s.protocol
    )
}

lazy rule enforcePlatformAnnotation {
    from s : ThingML!PlatformAnnotation(s.enforceAnyCommRule())
    to t : ThingML!PlatformAnnotation(
        name <— s.name,
        value <— s.value
    )
}
```

Listing D.2 – The ATL helpers for communication control

```
module HelpersComm;
create OUT: ThingML from TH: ThingML, CY : CyprIoT;
uses Copier;
uses Helpers;

rule copyThingMLModel {
    from s : ThingML!ThingMLModel
    to t : ThingML!ThingMLModel(
        configs <— s.configs,
        imports <— s.imports,
        protocols <— s.protocols,
        types <— s.types
    )
}

rule copyState {
    from s : ThingML!State(not (s.oclIsTypeOf(ThingML!CompositeState) or s.oclIsTypeOf(
        ThingML!FinalState)))
    to t : ThingML!State(
        annotations <— s.annotations,
        entry <— s.entry,
        exit <— s.exit,
        internal <— s.internal,
        name <— s.name,
        outgoing <— s.outgoing,
        properties <— s.properties
    )
}

—— Get the network to make (support for first network only for the moment)
helper def : firstNetwork() : CyprIoT!Network = CyprIoT!Network.allInstances().first();

helper context CyprIoT!Network def : bindsContainingThingInNetwork() : Sequence(CyprIoT!
    Bind) =
    self.hasBinds—>select(b | b.isBindMatchesInputThing())
;

helper context CyprIoT!Network def : bindsContainingThingObjectInNetwork(pathName :
    String, commAction : String) : Sequence(Sequence(String)) =
    self.hasBinds—>select(b | not(b.channelToBind.paths—>select(p | p.name=pathName).
        isEmpty()) and b.bindAction.toString().convertArrowToSendOrReceive()=commAction)
```

```
;

helper context CyprIoT!Network def : collectEnforcedPoliciesInNetwork() : Sequence(CyprIoT!
    Policy) = self.hasPolicyEnforcement.hasEnforcedPolicies;


helper context Sequence(CyprIoT!RuleComm) def : collectRulesCommElements() : Sequence(
    Sequence(String)) =
    self->collect(r | Sequence{r.commSubject.subjectOther.name ,r.effectComm.allow,r.
        effectComm.actionComm.toString().removeHash(),r.commObject.objectOther})
;

helper context CyprIoT!Policy def : collectRuleCommFromPolicy() : Sequence(CyprIoT!
    RuleComm) =
    self.hasRules->select(r | r.oclIsTypeOf(CyprIoT!RuleComm))
;

helper context Sequence(CyprIoT!RuleComm) def : collectRulesCommWithThingInSubject() :
    Sequence(CyprIoT!RuleComm) =
    self->select(r | r.oclAsType(CyprIoT!RuleComm).isSubjectOfRuleTypeThing() and r.
        oclAsType(CyprIoT!RuleComm).commSubject.subjectOther.name=thisModule.
        nameOfInputThing())->flatten()
;

helper context CyprIoT!Policy def : collectRulesCommWithThingInSubjectFromPolicy() :
    Sequence(CyprIoT!RuleComm) =
    self.collectRuleCommFromPolicy().collectRulesCommWithThingInSubject()
;

helper context Sequence(CyprIoT!Policy) def : rulesContainingThingInSubjectInPolicies() :
    Sequence(CyprIoT!RuleComm) =
    self->collect(p | p.collectRulesCommWithThingInSubjectFromPolicy())->flatten()
;

helper context CyprIoT!Network def : rulesContainingThingInSubjectInEnforcedPolicies() :
    Sequence(CyprIoT!RuleComm) =
    self.collectEnforcedPoliciesInNetwork().rulesContainingThingInSubjectInPolicies()
;

helper context ThingML!PlatformAnnotation def : isAnnotationContainerExternalConnector() :
     Boolean =
    self.refImmediateComposite().oclIsTypeOf(ThingML!ExternalConnector)
;

helper context CyprIoT!RuleComm def : isRuleSend() : Boolean = self.effectComm.
```

```
    actionComm.toString().removeHash()='send';

helper context CyprIoT!RuleComm def : isRuleReceive() : Boolean = self.effectComm.
    actionComm.toString().removeHash()='receive';

helper context CyprIoT!RuleComm def : isRuleDeny() : Boolean = self.effectComm.deny;

helper context CyprIoT!RuleComm def : isThingInRuleSubject() : Boolean =
    self.commSubject.subjectOther.oclAsType(CyprIoT!TypeThing).name=thisModule.
        nameOfInputThing()
;

helper context CyprIoT!Bind def : isThingInBind() : Boolean =
    self.bindsInstanceThing.name=thisModule.nameOfInputThing()
;

--Subjects checks
helper context CyprIoT!RuleComm def : isSubjectOfRuleThingAny() : Boolean = self.
    commSubject.subjectOther.oclIsTypeOf(CyprIoT!ThingAny);

helper context CyprIoT!RuleComm def : isSubjectOfRuleTypeThing() : Boolean = self.
    commSubject.subjectOther.oclIsTypeOf(CyprIoT!TypeThing);

helper context CyprIoT!RuleComm def : isSubjectOfRuleInstanceThing() : Boolean = self.
    commSubject.subjectOther.oclIsTypeOf(CyprIoT!InstanceThing);

helper context CyprIoT!RuleComm def : isSubjectOfRuleRole() : Boolean = self.commSubject.
    subjectOther.oclIsTypeOf(CyprIoT!Role);

helper context CyprIoT!RuleComm def : isSubjectOfRuleUser() : Boolean = self.commSubject.
    subjectOther.oclIsTypeOf(CyprIoT!User);

--Objects checks
helper context CyprIoT!RuleComm def : isObjectOfRuleTypeChannel() : Boolean = self.
    commObject.objectOther.oclIsTypeOf(CyprIoT!TypeChannel);

helper context CyprIoT!RuleComm def : isObjectOfRuleRole() : Boolean = self.commObject.
    objectOther.oclIsTypeOf(CyprIoT!Role);

helper context CyprIoT!RuleComm def : isObjectOfRuleUser() : Boolean = self.commObject.
    objectOther.oclIsTypeOf(CyprIoT!User);

helper context CyprIoT!RuleComm def : isObjectOfRuleThingAny() : Boolean = self.
    commObject.objectOther.oclIsTypeOf(CyprIoT!ThingAny);

```

```
helper context CyprIoT!RuleComm def : isObjectOfRuleTypeThing() : Boolean = self.
    commObject.objectOther.oclIsTypeOf(CyprIoT!TypeThing);

helper context CyprIoT!RuleComm def : isObjectOfRuleInstanceThing() : Boolean = self.
    commObject.objectOther.oclIsTypeOf(CyprIoT!InstanceThing);

helper context CyprIoT!RuleComm def : isObjectOfRulePath() : Boolean = self.commObject.
    objectOther.oclIsTypeOf(CyprIoT!Path);

-- First element of a type ... (for debugging)
helper def : firstEnforcedPolicy() : CyprIoT!Policy = thisModule.enforcedPoliciesInFirstNetwork
    ().first();

helper def : enforcedPoliciesInFirstNetwork() : Sequence(CyprIoT!Policy) = thisModule.
    firstNetwork().collectEnforcedPoliciesInNetwork();

helper def : firstRuleInPolicyFromFirstEnforcedPolicy() : CyprIoT!Rule = thisModule.
    firstEnforcedPolicy().hasRules.first();

helper def : firstBind() : CyprIoT!Bind = thisModule.firstNetwork().hasBinds.first();

helper context CyprIoT!Policy def : firstRuleOfPolicyAsRuleComm() : CyprIoT!RuleComm =
    self.hasRules.first().oclAsType(CyprIoT!RuleComm);

helper def : rulesContainingThingInSubjectInFirstNetwork() : Sequence(CyprIoT!RuleComm) =
    thisModule.enforcedPoliciesInFirstNetwork().rulesContainingThingInSubjectInPolicies()
;

-- Checks on the first element only (for debugging)
helper context CyprIoT!PoliciesEnforcement def : isThingInFirstBind() : Boolean =
    self.refImmediateComposite().oclAsType(CyprIoT!Network).hasBinds.first().isThingInBind()
;

helper context CyprIoT!Policy def : isThingInFirstRuleSubject() : Boolean =
    if(self.firstRuleOfPolicyAsRuleComm().isSubjectOfRuleTypeThing())
        then (self.firstRuleOfPolicyAsRuleComm().isThingInRuleSubject())
    else false endif
;

helper def : isFirstRuleOfFirstPolicyEnforcingThing() : Boolean = thisModule.
    firstEnforcedPolicy().isThingInFirstRuleSubject();

helper def : isThingInFirstBindFromFirstEnforcedPolicy() : String = thisModule.firstNetwork().
    hasPolicyEnforcement.isThingInFirstBind();

```

```
helper def : isFirstRuleOfFirstPolicyDeny() : Boolean = thisModule.
    firstRuleInPolicyFromFirstEnforcedPolicy().oclAsType(CyprIoT!RuleComm).isRuleDeny();

helper def : isFirstRuleOfFirstPolicySend() : Boolean = thisModule.
    firstRuleInPolicyFromFirstEnforcedPolicy().oclAsType(CyprIoT!RuleComm).isRuleSend();

helper def : isFirstRuleOfFirstPolicyReceive() : Boolean = thisModule.
    firstRuleInPolicyFromFirstEnforcedPolicy().oclAsType(CyprIoT!RuleComm).isRuleReceive();

helper def : isObjectPubSubInFirstRule() : Boolean = thisModule.
    firstRuleInPolicyFromFirstEnforcedPolicy().oclAsType(CyprIoT!RuleComm).
    isObjectOfRulePubSub();

helper def : actionOfFirstBind() : String = thisModule.firstBind().bindAction.toString();

-- Enforce or not (Boolean)
helper context ThingML!PlatformAnnotation def : noEnforcing() : Boolean = not(self.
    enforceDenySubscribe() or self.enforceDenyPublish());

helper context ThingML!PlatformAnnotation def : enforceMe() : Boolean =
    thisModule.firstNetwork().bindsContainingThingInNetwork().first()
;

helper def : bindOfInputThing() : CyprIoT!Bind = thisModule.firstNetwork().
    bindsContainingThingInNetwork().first();

helper context CyprIoT!Bind def : typeThingOfBind() : CyprIoT!TypeThing = self.
    bindsInstanceThing.typeThing.thingToInstantiate;

helper context CyprIoT!Bind def : actionOfBindConverted() : String = self.bindAction.toString
    ().convertArrowToSendOrReceive();

helper def : inputThingFromBind() : CyprIoT!TypeThing = thisModule.bindOfInputThing().
    typeThingOfBind();

helper context CyprIoT!RuleComm def : isInputThingInstanceInSubject() : Boolean = self.
    isSubjectOfRuleInstanceThing() and thisModule.nameOfInputThing()=self.commSubject.
    subjectOther.name;

helper context CyprIoT!RuleComm def : isTypeInputThingInSubject() : Boolean = self.
    isSubjectOfRuleTypeThing() and thisModule.bindOfInputThing().typeThingOfBind().name=
    self.commSubject.subjectOther.name;

helper context CyprIoT!RuleComm def : isInputThingInSubjectRoles() : Boolean = self.
    isSubjectOfRuleRole() and not(thisModule.inputThingFromBind().assignedRoles.
    oclIsUndefined()) and
    (self.isInputThingInSubjectRole());

helper context CyprIoT!RuleComm def : isInputThingInSubjectRole() : Boolean = thisModule.
    inputThingFromBind().assignedRoles->exists(r | r.name=self.commSubject.subjectOther.
    name);

helper context CyprIoT!RuleComm def : isInputThingOwnedByUser() : Boolean = if(not(
    thisModule.bindOfInputThing().bindsInstanceThing.typeThing.owner.oclIsUndefined()))
    then self.isSubjectOfRuleUser() and thisModule.bindOfInputThing().bindsInstanceThing.
    typeThing.owner.name=self.commSubject.subjectOther.name else false endif;

helper context CyprIoT!RuleComm def : isInputThingInObjectRole() : Boolean = thisModule.
    inputThingFromBind().assignedRoles->exists(r | r.name=self.commObject.objectOther.
    name);

helper context CyprIoT!RuleComm def : isActionInBindAndRuleMatching() : Boolean =
    thisModule.bindOfInputThing().actionOfBindConverted()=self.effectComm.actionComm.
    toString().removeHash();

helper context CyprIoT!RuleComm def : isAnnotationPathMatchesRulePath(annotation :
    ThingML!PlatformAnnotation) : Boolean = annotation.value=self.getFullPathOfRule();

helper context CyprIoT!RuleComm def : isAnnotationPathMatchesRuleChannel(annotation :
    ThingML!PlatformAnnotation) : Boolean = self.commObject.objectOther.oclAsType(
    CyprIoT!TypeChannel).hasPaths->exists(p | annotation.value=thisModule.
    bindOfInputThing().getFullPathName(p));

helper context CyprIoT!RuleComm def : isAnyBindMirrorActionInputThing(annotation :
    ThingML!PlatformAnnotation) : Boolean = thisModule.firstNetwork().hasBinds->exists(b
    | b.typeThingOfBind().name=self.commObject.objectOther.name and b.
    actionOfBindConverted().mirrorSendOrReceive()=thisModule.bindOfInputThing().
    actionOfBindConverted() and b.channelToBind.paths->exists(p | annotation.value=
    thisModule.bindOfInputThing().getFullPathName(p)));

helper context CyprIoT!RuleComm def : isAnyBindMirrorActionInputInstanceThing(annotation
    : ThingML!PlatformAnnotation) : Boolean = thisModule.firstNetwork().hasBinds->exists(
    b | b.getInstanceThingNameFromBind()=self.commObject.objectOther.name and b.
    actionOfBindConverted().mirrorSendOrReceive()=thisModule.bindOfInputThing().
    actionOfBindConverted() and b.channelToBind.paths->exists(p | annotation.value=
    thisModule.bindOfInputThing().getFullPathName(p)));

helper context CyprIoT!RuleComm def : getFullPathOfRule() : String = thisModule.
    bindOfInputThing().getFullPathName(self.commObject.objectOther.oclAsType(CyprIoT!
    Path));
```

```

helper context ThingML!ExternalConnector def : addAnnotationsAfterEnforcements() :
    Sequence(ThingML!PlatformAnnotation) = self.annotations−>collect(a | if(thisModule.
    enforcePlatformAnnotation(a).oclIsUndefined()) then Sequence{} else thisModule.
    enforcePlatformAnnotation(a) endif)−>flatten();

helper context ThingML!PlatformAnnotation def : enforceDenyPath(rulecomm : CyprIoT!
    RuleComm) : Boolean =
    ((rulecomm.isInputThingInstanceInSubject() or rulecomm.isTypeInputThingInSubject()) or
    rulecomm.isInputThingInSubjectRoles() or rulecomm.isInputThingOwnedByUser()) and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    rulecomm.isObjectOfRulePath() and
    rulecomm.isAnnotationPathMatchesRulePath(self)
;

helper context ThingML!PlatformAnnotation def : enforceDenyChannel(rulecomm : CyprIoT!
    RuleComm) : Boolean =
    ((rulecomm.isInputThingInstanceInSubject() or rulecomm.isTypeInputThingInSubject()) or
    rulecomm.isInputThingInSubjectRoles() or rulecomm.isInputThingOwnedByUser()) and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    rulecomm.isObjectOfRuleTypeChannel() and
    rulecomm.isAnnotationPathMatchesRuleChannel(self)
;

helper context ThingML!PlatformAnnotation def : enforceSubjectAndObjectThings(rulecomm :
     CyprIoT!RuleComm) : Boolean =
    rulecomm.isTypeInputThingInSubject() and
    rulecomm.isObjectOfRuleTypeThing() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    rulecomm.isAnyBindMirrorActionInputThing(self)
;

helper context ThingML!PlatformAnnotation def : enforceSubjectAndObjectInstanceThings(
    rulecomm : CyprIoT!RuleComm) : Boolean =
    rulecomm.isInputThingInstanceInSubject() and
    rulecomm.isObjectOfRuleInstanceThing() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    rulecomm.isAnyBindMirrorActionInputInstanceThing(self)
;

helper context ThingML!PlatformAnnotation def :
    enforceSubjectThingAndObjectInstanceThing(rulecomm : CyprIoT!RuleComm) : Boolean =
    rulecomm.isTypeInputThingInSubject() and
    rulecomm.isObjectOfRuleInstanceThing() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    rulecomm.isAnyBindMirrorActionInputInstanceThing(self)
;

helper context ThingML!PlatformAnnotation def :
    enforceSubjectInstanceThingAndObjectThing(rulecomm : CyprIoT!RuleComm) : Boolean =
    rulecomm.isInputThingInstanceInSubject() and
    rulecomm.isObjectOfRuleTypeThing() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    rulecomm.isAnyBindMirrorActionInputThing(self)
;

helper context ThingML!PlatformAnnotation def : enforceUsers(rulecomm : CyprIoT!
    RuleComm) : Boolean =
    rulecomm.isInputThingOwnedByUser() and
    rulecomm.isObjectOfRuleUser() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    thisModule.firstNetwork().hasBinds->exists(b | b.actionOfBindConverted().
        mirrorSendOrReceive()=thisModule.bindOfInputThing().actionOfBindConverted() and b
        .bindsInstanceThing.typeThing.owner.name=rulecomm.commObject.objectOther.name
        and b.channelToBind.paths->exists(p | self.value=thisModule.bindOfInputThing().
        getFullPathName(p)))
;

helper context ThingML!PlatformAnnotation def : enforceRoles(rulecomm : CyprIoT!
    RuleComm) : Boolean =
    rulecomm.isSubjectOfRuleRole() and
    rulecomm.isInputThingInSubjectRole() and
    rulecomm.isObjectOfRuleRole() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    thisModule.firstNetwork().hasBinds->exists(b | b.actionOfBindConverted().
        mirrorSendOrReceive()=thisModule.bindOfInputThing().actionOfBindConverted() and b
        .typeThingOfBind().assignedRoles->exists(r | r.name=rulecomm.commObject.
        objectOther.name) and b.channelToBind.paths->exists(p | self.value=thisModule.
        bindOfInputThing().getFullPathName(p)))
;

helper context ThingML!PlatformAnnotation def : enforceRoleAndThing(rulecomm : CyprIoT!
    RuleComm) : Boolean =
    rulecomm.isSubjectOfRuleRole() and
    rulecomm.isInputThingInSubjectRole() and
    rulecomm.isObjectOfRuleTypeThing() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    thisModule.firstNetwork().hasBinds->exists(b | b.actionOfBindConverted().
        mirrorSendOrReceive()=thisModule.bindOfInputThing().actionOfBindConverted() and b
        .typeThingOfBind().name=rulecomm.commObject.objectOther.name and b.
        channelToBind.paths->exists(p | self.value=thisModule.bindOfInputThing().
        getFullPathName(p)))
;

helper context ThingML!PlatformAnnotation def : enforceRoleAndInstanceThing(rulecomm :
    CyprIoT!RuleComm) : Boolean =
    rulecomm.isSubjectOfRuleUser() and
    rulecomm.isInputThingInSubjectRole() and
    rulecomm.isObjectOfRuleInstanceThing() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    thisModule.firstNetwork().hasBinds->exists(b | b.actionOfBindConverted().
        mirrorSendOrReceive()=thisModule.bindOfInputThing().actionOfBindConverted() and b
        .getInstanceThingNameFromBind()=rulecomm.commObject.objectOther.name and b.
        channelToBind.paths->exists(p | self.value=thisModule.bindOfInputThing().
        getFullPathName(p)))
;

helper context ThingML!PlatformAnnotation def : enforceUserAndThing(rulecomm : CyprIoT!
    RuleComm) : Boolean =
    rulecomm.isSubjectOfRuleUser() and
    rulecomm.isInputThingOwnedByUser() and
    rulecomm.isObjectOfRuleTypeThing() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    thisModule.firstNetwork().hasBinds->exists(b | b.actionOfBindConverted().
        mirrorSendOrReceive()=thisModule.bindOfInputThing().actionOfBindConverted() and b
        .typeThingOfBind().name=rulecomm.commObject.objectOther.name and b.
        channelToBind.paths->exists(p | self.value=thisModule.bindOfInputThing().
        getFullPathName(p)))
;

helper context ThingML!PlatformAnnotation def : enforceUserAndInstanceThing(rulecomm :
    CyprIoT!RuleComm) : Boolean =
    rulecomm.isSubjectOfRuleUser() and
    rulecomm.isInputThingOwnedByUser() and
    rulecomm.isObjectOfRuleInstanceThing() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    thisModule.firstNetwork().hasBinds−>exists(b | b.actionOfBindConverted().
        mirrorSendOrReceive()=thisModule.bindOfInputThing().actionOfBindConverted() and b
        .getInstanceThingNameFromBind()=rulecomm.commObject.objectOther.name and b.
        channelToBind.paths−>exists(p | self.value=thisModule.bindOfInputThing().
        getFullPathName(p)))
;

helper context ThingML!PlatformAnnotation def : enforceThingAndRole(rulecomm : CyprIoT!
    RuleComm) : Boolean =
    rulecomm.isTypeInputThingInSubject() and
    rulecomm.isObjectOfRuleRole() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    thisModule.firstNetwork().hasBinds−>exists(b | b.actionOfBindConverted().
        mirrorSendOrReceive()=thisModule.bindOfInputThing().actionOfBindConverted() and b
        .typeThingOfBind().assignedRoles−>exists(r | r.name=rulecomm.commObject.
        objectOther.name) and thisModule.bindOfInputThing().isThingInBind() and b.
        channelToBind.paths−>exists(p | self.value=thisModule.bindOfInputThing().
        getFullPathName(p)))
;

helper context ThingML!PlatformAnnotation def : enforceUserAndRole(rulecomm : CyprIoT!
    RuleComm) : Boolean =
    rulecomm.isInputThingOwnedByUser() and
    rulecomm.isObjectOfRuleRole() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    thisModule.firstNetwork().hasBinds−>exists(b | b.actionOfBindConverted().
        mirrorSendOrReceive()=thisModule.bindOfInputThing().actionOfBindConverted() and b
        .typeThingOfBind().assignedRoles−>exists(r | r.name=rulecomm.commObject.
        objectOther.name) and thisModule.bindOfInputThing().isThingInBind() and b.
        channelToBind.paths−>exists(p | self.value=thisModule.bindOfInputThing().
        getFullPathName(p)))
;


helper context ThingML!PlatformAnnotation def : enforceInstanceThingAndRole(rulecomm :
    CyprIoT!RuleComm) : Boolean =
    rulecomm.isInputThingInstanceInSubject() and
    rulecomm.isObjectOfRuleRole() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    thisModule.firstNetwork().hasBinds->exists(b | b.actionOfBindConverted().
        mirrorSendOrReceive()=thisModule.bindOfInputThing().actionOfBindConverted() and b
        .typeThingOfBind().assignedRoles->exists(r | r.name=rulecomm.commObject.
        objectOther.name) and thisModule.bindOfInputThing().isThingInBind() and b.
        channelToBind.paths->exists(p | self.value=thisModule.bindOfInputThing().
        getFullPathName(p)))
;

helper context ThingML!PlatformAnnotation def : enforceThingAndUser(rulecomm : CyprIoT!
    RuleComm) : Boolean =
    rulecomm.isTypeInputThingInSubject() and
    rulecomm.isObjectOfRuleUser() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    thisModule.firstNetwork().hasBinds->exists(b | b.actionOfBindConverted().
        mirrorSendOrReceive()=thisModule.bindOfInputThing().actionOfBindConverted() and b
        .bindsInstanceThing.typeThing.owner.name=rulecomm.commObject.objectOther.name
        and thisModule.bindOfInputThing().isThingInBind() and b.channelToBind.paths->
        exists(p | self.value=thisModule.bindOfInputThing().getFullPathName(p)))
;

helper context ThingML!PlatformAnnotation def : enforceRoleAndUser(rulecomm : CyprIoT!
    RuleComm) : Boolean =
    rulecomm.isInputThingInSubjectRoles() and
    rulecomm.isObjectOfRuleUser() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    thisModule.firstNetwork().hasBinds->exists(b | b.actionOfBindConverted().
        mirrorSendOrReceive()=thisModule.bindOfInputThing().actionOfBindConverted() and b
        .bindsInstanceThing.typeThing.owner.name=rulecomm.commObject.objectOther.name
        and thisModule.bindOfInputThing().isThingInBind() and b.channelToBind.paths->
        exists(p | self.value=thisModule.bindOfInputThing().getFullPathName(p)))
;

helper context ThingML!PlatformAnnotation def : enforceInstanceThingAndUser(rulecomm :
    CyprIoT!RuleComm) : Boolean =
    rulecomm.isInputThingInstanceInSubject() and
    rulecomm.isObjectOfRuleUser() and
    rulecomm.effectComm.deny and
    rulecomm.isActionInBindAndRuleMatching() and
    thisModule.firstNetwork().hasBinds->exists(b | b.actionOfBindConverted().
        mirrorSendOrReceive()=thisModule.bindOfInputThing().actionOfBindConverted() and b
        .bindsInstanceThing.typeThing.owner.name=rulecomm.commObject.objectOther.name
        and thisModule.bindOfInputThing().isThingInBind() and b.channelToBind.paths->
        exists(p | self.value=thisModule.bindOfInputThing().getFullPathName(p)))
```

```
;

helper context ThingML!PlatformAnnotation def : enforceAnyCommRule() : Boolean =
    not(thisModule.firstNetwork().collectEnforcedPoliciesInNetwork()
        ->forAll(p | p.collectRuleCommFromPolicy()
            ->exists(r |
                self.enforceDenyPath(r) or
                self.enforceDenyChannel(r) or
                self.enforceSubjectAndObjectThings(r) or
                self.enforceSubjectAndObjectInstanceThings(r) or
                self.enforceRoles(r) or
                self.enforceRoleAndThing(r) or
                self.enforceRoleAndInstanceThing(r) or
                self.enforceThingAndRole(r) or
                self.enforceInstanceThingAndRole(r) or
                self.enforceSubjectInstanceThingAndObjectThing(r) or
                self.enforceSubjectThingAndObjectInstanceThing(r) or
                self.enforceThingAndUser(r) or
                self.enforceInstanceThingAndUser(r) or
                self.enforceUserAndThing(r) or
                self.enforceUserAndInstanceThing(r) or
                self.enforceUsers(r) or
                self.enforceUserAndRole(r) or
                self.enforceRoleAndUser(r)
                )))
;
```

Appendix E

ATL Rules for Smart Rules Transformations

Listing E.1 – The ATL rules for smart rules

```
module RuleTrigger;

create OUT: ThingML from TH: ThingML, CY : CyprIoT;

uses HelpersTrigger;

rule copyThingMLModel {
    from s : ThingML!ThingMLModel
    to t : ThingML!ThingMLModel(
        configs <— s.configs,
        imports <— s.imports,
        protocols <— s.protocols,
        types <— s.types
    )
}


rule copyCompositeState {
    from s : ThingML!CompositeState
    to t : ThingML!CompositeState(
        name <— s.name,
        annotations <— s.annotations,
        entry <— s.entry,
        exit <— s.exit,
        history <— s.history,
        initial <— s.initial,
        internal <— s.internal,
        outgoing <— s.outgoing,
        properties <— s.properties,
        region <— s.region,
        session <— s.session,
        substate <— s.substate
    )
}

rule copyExternalConnector {
    from s : ThingML!ExternalConnector
    to t : ThingML!ExternalConnector(
        annotations <— s.annotations,
        inst <— s.inst,
        name <— s.name,
        port <— s.port,
        protocol <— s.protocol
    )
}

rule copyThing {
    from s : ThingML!Thing
    to t : ThingML!Thing(
        name <— s.name,
        ports <— s.ports,
        annotations <— s.annotations,
        assign <— s.assign,
        behaviour <— s.behaviour,
        fragment <— s.fragment,
        functions <— s.functions,
        messages <— s.messages—>union(CyprIoT!RuleTrigger.allInstances()),
        includes <— s.includes,
        properties <— s.properties
    )
}

rule copyProvidedPort {
    from s : ThingML!ProvidedPort
    to t : ThingML!ProvidedPort(
        name <— s.name,
        receives <— s.receives—>union(CyprIoT!RuleTrigger.allInstances()),
        sends <— s.sends—>union(CyprIoT!RuleTrigger.allInstances()),
        annotations <— s.annotations
    )
}

rule copyRequiredPort {
    from s : ThingML!RequiredPort
    to t : ThingML!RequiredPort(
        name <− s.name,
        receives <− s.setReceivesPort(),
        sends <− s.setSendsPort(),
        annotations <− s.annotations
    )
}

rule triggerMess {
    from s : CyprIoT!RuleTrigger
    to triggerMessage : ThingML!Message(
        name <− if(s.isGoToState())
                    then s.triggerObjectStateName()
                    else if(s.isExecuteFunction())
                        then s.triggerObjectFunctionName()
                    else 'perfomTransition'
                    endif endif
    )
}

rule copyState {
    from s : ThingML!State( not(s.oclIsTypeOf(ThingML!CompositeState) or s.oclIsTypeOf(
        ThingML!FinalState)))
    to
    t : ThingML!State(
        annotations <− s.annotations,
        entry <− s.setOnEntry(),
        exit <− s.exit,
        internal <− s.setInternals(),
        name <− s.name,
        outgoing <− s.setOutgoing(),
        properties <− s.properties
    )
}

lazy rule getTransition {
    from s : ThingML!State , k : CyprIoT!RuleTrigger
    to
    transition : ThingML!Transition(
        target <− k.setTransitionTarget(),
        event <− receive
    ),
    receive : ThingML!ReceiveMessage(
        port <− ThingML!Thing.allInstances().first().ports.first(),
        message <— thisModule.resolveTemp(k, 'triggerMessage'),
        name <— 'trigger'
    ),
    action2 : ThingML!FunctionCallStatement(
        function <— if(not(k.isInputThingInRuleTriggerSubject()) and k.isExecuteFunction())
            then k.getFunctionToExecute() else OclUndefined endif,
        parameters <— if(k.isExecuteFunction() and not(k.effectTrigger.actionTrigger.
            thingWithFunction.getFunction.parameters.oclIsUndefined())) then k.effectTrigger.
            actionTrigger.thingWithFunction.getFunction.parameters—>collect( p | thisModule.
            multipleParameters(p))
                        else Sequence{} endif
    )
}

lazy rule getInternalTransition {
    from s : ThingML!State , k : CyprIoT!RuleTrigger
    to
    transitionInternal : ThingML!InternalTransition(
        event <— receive2,
        action <— action
    ),
    receive2 : ThingML!ReceiveMessage(
        port <— ThingML!Thing.allInstances().first().ports.first(),
        message <— thisModule.resolveTemp(k, 'triggerMessage'),
        name <— 'trigger'
    ),
    action : ThingML!FunctionCallStatement(
        function <— if(not(k.isInputThingInRuleTriggerSubject()) and k.isExecuteFunction())
            then k.getFunctionToExecute() else OclUndefined endif,
        parameters <— if(k.isExecuteFunction() and not(k.effectTrigger.actionTrigger.
            thingWithFunction.getFunction.parameters.oclIsUndefined())) then k.effectTrigger.
            actionTrigger.thingWithFunction.getFunction.parameters—>collect( p | thisModule.
            multipleParameters(p))
                        else Sequence{} endif
    )
}

lazy rule multipleParameters {
    from s: String
    to
    expression : ThingML!IntegerLiteral(
        intValue <— s.toInteger().refInvokeOperation('longValue', Sequence{}) —— bug : use
            of reflexivity to convert integer to long
    )
}
```

```

lazy rule multipleTransition {
    from s: ThingML!Transition , k : CyprIoT!RuleTrigger
    to
    performTransition : ThingML!Transition(
        target <— s.target,
        event <— receive
    ),
    receive : ThingML!ReceiveMessage(
        port <— ThingML!Thing.allInstances().first().ports.first(),
        message <— thisModule.resolveTemp(k, 'triggerMessage'),
        name <— 'trigger'
    )
}

lazy rule groupActionOnEntry {
    from s : ThingML!Action , k : CyprIoT!RuleTrigger
    to
    groupAction : ThingML!ActionBlock(
        actions <— if(not(s.oclIsUndefined())) then Sequence{action,s} else Sequence{
            action} endif
    ),
    action : ThingML!SendAction(
        message <— thisModule.resolveTemp(k, 'triggerMessage'),
        port <— ThingML!Port.allInstances().first()
    )

}
```

Listing E.2 – The ATL helpers for smart rules

```
module HelpersTrigger;

create OUT: ThingML from TH: ThingML, CY : CyprIoT;

uses Copier;
uses Helpers;

```

```
-- Get the network to make (support for the first network only)
helper def : firstNetwork() : CyprIoT!Network = CyprIoT!Network.allInstances().first();

helper context CyprIoT!Network def : collectEnforcedPoliciesInNetwork() : Sequence(CyprIoT!
    Policy) =
    self.hasPolicyEnforcement.hasEnforcedPolicies;

helper context CyprIoT!Policy def : collectRuleTriggerFromPolicy() : Sequence(CyprIoT!
    RuleTrigger) =
    self.hasRules->select(r | r.oclIsTypeOf(CyprIoT!RuleTrigger))
;

helper def : bindOfInputThing() : CyprIoT!Bind = thisModule.firstNetwork().
    bindsContainingThingInNetwork().first();

helper context CyprIoT!Network def : bindsContainingThingInNetwork() : Sequence(CyprIoT!
    Bind) =
    self.hasBinds->select(b | b.isBindMatchesInputThing())
;

helper context CyprIoT!Network def : collectEnforcedEnforcedTriggerRulesInNetwork() :
    Sequence(CyprIoT!RuleTrigger) =
    self.collectEnforcedPoliciesInNetwork()->collect(p | p.collectRuleTriggerFromPolicy())->
        flatten()
;

helper def : collectTriggerRulesInMyNetwork() : Sequence(CyprIoT!RuleTrigger) =
    thisModule.firstNetwork().collectEnforcedEnforcedTriggerRulesInNetwork()
;

helper context CyprIoT!RuleTrigger def : isInputThingInRuleTriggerFunctionObject() :
    Boolean = self.effectTrigger.actionTrigger.thingWithFunction.thing.name=thisModule.
    bindOfInputThing().bindsInstanceThing.typeThing.thingToInstantiate.name;


helper context CyprIoT!RuleTrigger def : isInputThingInRuleTriggerObject() : Boolean = self.
    effectTrigger.actionTrigger.thingWithState.thing.name=thisModule.bindOfInputThing().
    bindsInstanceThing.typeThing.thingToInstantiate.name;

helper context CyprIoT!RuleTrigger def : isInputThingInRuleTriggerSubject() : Boolean = self.
    thingWithState.thing.name=thisModule.bindOfInputThing().bindsInstanceThing.typeThing.
    thingToInstantiate.name;

helper context ThingML!State def : isStateNameEqualToRuleTriggerObjectState(k : CyprIoT!
    RuleTrigger) : Boolean = self.name=k.effectTrigger.actionTrigger.thingWithState.getState.
```

```
        state.name;

helper context ThingML!State def : isStateNameEqualToRuleTriggerSubjectState(k : CyprIoT!
        RuleTrigger) : Boolean = self.name=k.thingWithState.getState.state.name;

helper context CyprIoT!RuleTrigger def : getFunctionToExecute() : ThingML!Function =
        ThingML!Function.allInstances()−>select(f | f.name=self.effectTrigger.actionTrigger.
        thingWithFunction.getFunction.function.name).first();

helper context CyprIoT!RuleTrigger def : isGoToState() : Boolean = not(self.effectTrigger.
        actionTrigger.goToState.oclIsUndefined());

helper context CyprIoT!RuleTrigger def : isExecuteFunction() : Boolean = not(self.
        effectTrigger.actionTrigger.executeFunction.oclIsUndefined());

helper context CyprIoT!RuleTrigger def : isPerformTransition() : Boolean = not(self.
        effectTrigger.actionTrigger.performTransition.oclIsUndefined());

helper context CyprIoT!RuleTrigger def : triggerObjectStateName() : String = self.
        effectTrigger.actionTrigger.thingWithState.getState.state.name;

helper context CyprIoT!RuleTrigger def : triggerObjectFunctionName() : String = self.
        effectTrigger.actionTrigger.thingWithFunction.getFunction.function.name;

helper context ThingML!RequiredPort def : setReceivesPort() : Sequence(ThingML!Message)
        =
thisModule.collectTriggerRulesInMyNetwork()−>iterate(r ; receives : Sequence(ThingML!
        Message) = self.receives |
        if(r.isGoToState() and r.isInputThingInRuleTriggerObject())
                        then receives−>union(CyprIoT!RuleTrigger.allInstances()) else receives endif
                )
;

helper context ThingML!RequiredPort def : setSendsPort() : Sequence(ThingML!Message) =
thisModule.collectTriggerRulesInMyNetwork()−>iterate(r ; sends : Sequence(ThingML!Message
        ) = self.sends |
        if(r.isGoToState() and r.isInputThingInRuleTriggerObject())
            then sends else sends−>union(CyprIoT!RuleTrigger.allInstances()) endif
)
;

helper context ThingML!State def : setInternals() : Sequence(ThingML!InternalTransition) =
        thisModule.collectTriggerRulesInMyNetwork()−>iterate(r ; internals : Sequence(ThingML!
            InternalTransition) = self.internal |
            if(r.isExecuteFunction()
                    and r.isInputThingInRuleTriggerFunctionObject())
                        then internals->union(Sequence{thisModule.getInternalTransition(self,r)})
                        else internals endif
    )
;
helper context ThingML!State def : setOutgoing() : Sequence(ThingML!Transition) =
    thisModule.collectTriggerRulesInMyNetwork()->iterate(r ; transitions : Sequence(ThingML!Transition) = self.outgoing |

if(r.isGoToState() and
    r.isInputThingInRuleTriggerObject() and
    not(self.isStateNameEqualToRuleTriggerObjectState(r)))
        then if(transitions->exists(tr | tr.guard.oclIsUndefined() and tr.event.oclIsUndefined())) then transitions else transitions->union(Sequence{thisModule.getTransition(self,r)}) endif
    else if(r.isPerformTransition() and not(r.isInputThingInRuleTriggerSubject())) then
    transitions->union(if(ThingML!Transition.allInstances()->select(i | self.name=i.refImmediateComposite().oclAsType(ThingML!State).name)->collect(a | thisModule.multipleTransition(a,r)).asOrderedSet().size()>0 and r.effectTrigger.actionTrigger.transitionRank=0)
        then
        Sequence{ThingML!Transition.allInstances()->select(i | self.name=i.refImmediateComposite().oclAsType(ThingML!State).name)->collect(a | thisModule.multipleTransition(a,r)).asOrderedSet().at(1)}
        else if(ThingML!Transition.allInstances()->select(i | self.name=i.refImmediateComposite().oclAsType(ThingML!State).name)->collect(a | thisModule.multipleTransition(a,r)).asOrderedSet().size()>0 and not(r.effectTrigger.actionTrigger.transitionRank=0) and ThingML!Transition.allInstances()->select(i | self.name=i.refImmediateComposite().oclAsType(ThingML!State).name)->collect(a | thisModule.multipleTransition(a,r)).asOrderedSet().size()<=r.effectTrigger.actionTrigger.transitionRank)
        then
        Sequence{ThingML!Transition.allInstances()->select(i | self.name=i.refImmediateComposite().oclAsType(ThingML!State).name)->collect(a | thisModule.multipleTransition(a,r)).asOrderedSet().at(r.effectTrigger.actionTrigger.transitionRank)}
        else Sequence{} endif endif
    )
    else transitions endif endif
    )
;

helper context ThingML!State def : setOnEntry() : ThingML!Action =
thisModule.collectTriggerRulesInMyNetwork()->iterate(r ; action : ThingML!Action = self.entry
    |
    if(r.isInputThingInRuleTriggerSubject() and self.
        isStateNameEqualToRuleTriggerSubjectState(r))
        then thisModule.groupActionOnEntry(action, r)
        else action endif
    )
;

helper context CyprIoT!RuleTrigger def : setTransitionTarget() : ThingML!State =

if(self.isGoToState()
        and self.isInputThingInRuleTriggerObject())
    then ThingML!State.allInstances()->select(t | t.
        isStateNameEqualToRuleTriggerObjectState(self)).first()
    else ThingML!State.allInstances().first() endif
;
```

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Une méthodologie fondée sur les modèles pour unifier l'ingénierie logicielle dans l'Internet des objets

Mot clés : Internet des objets, Génie logiciel, Ingénierie dirigée par les modèles, Transformation des modèles, Politiques de contrôle, Génération de code

Résumé : L'Internet des objets (IdO) vise à connecter tout objet, partout, en tout temps (TTT). Cette hypothèse entraîne de nombreux défis en matière de génie logiciel. Ces défis constituent un sérieux obstacle à son adoption à grande échelle. L'une des principales caractéristiques de l'IdO est la généricité, c'est-à-dire permettre aux objets de se connecter de manière transparente, quelles que soient la technologie qu'ils utilisent. L'Ingénierie Dirigée par les Modèles (IDM) est un paradigme qui préconise l'utilisation de modèles pour résoudre les problèmes de génie logiciel. L'IDM pourrait aider à répondre au besoin de généricité de l'IdO du point de vue du génie logiciel. Les approches d'IDM existantes se focalisent essentiellement sur la modélisation du comportement des objets. Peu d'attention a été accordée à la modélisation liée à leur réseautage. La présente thèse présente une méthodologie pour l'IdO basée sur l'IDM. Fondamentalement, elle fournit une solution pour créer des réseaux intelligents d'objets. Le principe que nous utilisons consiste à contourner l'hétérogénéité intrinsèque de l'IdO en séparant la spécification du réseau, c'est-à-dire les objets, le schéma de communication et les contraintes, de son implémentation concrète, c'est-à-dire les artefacts logiciels de bas niveau (par exemple, le code source). Techniquement, la méthodologie repose sur un langage dédié basé sur les modèles pour la spécification du réseau et une procédure pour la génération du code des artefacts de bas niveau à partir de cette spécification. L'adoption de cette méthodologie rend l'ingénierie logicielle des applications d'IdO plus rigoureuse, permet de prévenir les bogues plus tôt et de gagner du temps.

Title: A Model-Driven Methodology to Unify Software Engineering in the Internet of Things

Keywords: Internet of Things, Software Engineering, Model-Driven Engineering, Model Transformation, Policy Enforcement, Code Generation

Abstract: The Internet of Things (IoT) aims for connecting Anything, Anywhere, Anytime (AAA). This assumption brings about a good deal of software engineering challenges. These challenges constitute a serious obstacle to its wider adoption. The main feature of the IoT is genericity, i.e., enabling things to connect seamlessly regardless of their technologies. Model-Driven Engineering (MDE) is a paradigm that advocates using models to address software engineering problems. MDE could help to meet the genericity of the IoT from a software engineering perspective. In that sense, the IoT could be a requirement provider on the one hand and MDE its solution provider on the other. Existing MDE approaches focus on modeling the behavior of things. But, little attention has been paid to network-related modeling. The present thesis presents a methodology to create smart networks of things based on MDE. It aims to cover and leverage the network-related aspects of an IoT application compared to the existing work. The principle we use consists of avoiding the intrinsic heterogeneity of the IoT by separating the specification of the network, i.e., the things, the communication scheme, and the constraints, from their concrete implementation, i.e., the low-level artifacts (e.g., source code). Technically, the methodology relies on a model-based Domain-Specific Language and a code generator. The former enables the specification of the network, and the latter provides a procedure to generate the low-level artifacts from this specification. The adoption of this methodology permits making software engineering of IoT applications more deterministic and saving a significant amount of lines of code compared to the state of practice.