

الجمهورية الجزائرية الديمقراطية الشعبية  
وزارة التعليم العالي و البحث العلمي

République Algérienne Démocratique et Populaire  
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Mohammed Seddik  
Benyahia-Jijel  
Faculté des Sciences Exactes et  
Informatique  
Département d'Informatique



جامعة محمد الصديق بن يحيى - جيجل  
كلية العلوم الدقيقة و الإعلام الآلي  
قسم الإعلام الآلي

## THÈSE

PRÉSENTÉ PAR  
ABDELOUAHAB FORTAS

POUR L'OBTENTION DU DIPLÔME DE DOCTORAT EN SCIENCES  
FILIÈRE : INFORMATIQUE  
OPTION : SYSTÈME D'INFORMATION

## THÈME

---

MODÉLISATION ET ANALYSE DES APPLICATIONS DE L'INTERNET DES  
OBJETS : UNE APPROCHE BASÉE SUR L'INGÉNIERIE DIRIGÉE PAR LES  
MODÈLES

---

Soutenu publiquement le: 15/06/2023      Devant le jury composé de:

Nom et prénom	Grade		
M. Melit Ali	Pr.	Univ. de Jijel	Président
M. Kerkouche Elhillali	M.C.A	Univ. de Jijel	Rapporteur
M. Chaoui Allaoua	Pr.	Univ. de Constantine 2	Co-Rapporteur
M. Bourouis Abdelhabib	Pr.	Univ. d'Oum el Bouaghi	Examineur
M. Lemouari Ali	Pr.	Univ. de Tamaghasset	Examineur
M. Khelfaoui Khaled	M.C.A	Univ. de Jijel	Examineur

الجمهورية الجزائرية الديمقراطية الشعبية  
وزارة التعليم العالي و البحث العلمي  
Democratic and Popular Republic of Algeria  
Ministry of Higher Education and Scientific Research

University of Mohammed Seddik  
Ben Yahia- Jijel  
Faculty of Exact Sciences and  
Computer Science  
Department of Computer Science



جامعة محمد الصديق بن يحيى - جيجل  
كلية العلوم الدقيقة و الإعلام الآلي  
قسم الإعلام الآلي

## THESIS

PRESENTED BY  
ABDELOUAHAB FORTAS

TO OBTAIN THE DEGREE OF DOCTOR OF SCIENCE  
FIELD: COMPUTER SCIENCE  
OPTION: INFORMATION SYSTEM

### THEME

---

MODELING AND ANALYSIS OF INTERNET OF THINGS APPLICATIONS:  
A MODEL DRIVEN ENGINEERING BASED APPROACH

---

Publicly defended on: 15/06/2023

In front of the jury composed of:

Name and surname	Grade		
M. Melit Ali	Pr.	Univ. of Jijel	President
M. Kerkouche Elhillali	M.C.A	Univ. of Jijel	Supervisor
M. Chaoui Allaoua	Pr.	Univ. of Constantine 2	Co-Supervisor
M. Bourouis Abdelhabib	Pr.	Univ. of Oum el Bouaghi	Examiner
M. Lemouari Ali	Pr.	Univ. of Tamaghasset	Examiner
M. Khelfaoui Khaled	M.C.A	Univ. of Jijel	Examiner

Academic year: 2022/2023



*To my parents, may God keep them.*

*To my dear brothers and sisters.*

*To my dear wife and children.*

*To all my friends.*

# Table of Contents

---

<b>Table of Contents</b>	<b>v</b>
<b>List of Tables</b>	<b>ix</b>
<b>List of Figures</b>	<b>x</b>
<b>Acknowledgements</b>	<b>xii</b>
<b>Abstract</b>	<b>xiii</b>
<b>General introduction</b>	<b>1</b>
<b>1 Internet of Things: An overview</b>	<b>5</b>
1.1 Introduction . . . . .	5
1.2 Definitions . . . . .	5
1.3 Applications of IoT . . . . .	6
1.3.1 Smart homes . . . . .	7
1.3.2 Smart cities . . . . .	7
1.3.3 Energy . . . . .	8
1.3.4 Health . . . . .	8
1.3.5 Transport . . . . .	8
1.3.6 Manufacture . . . . .	8
1.3.7 Environment and agriculture . . . . .	9
1.4 Architecture of IoT . . . . .	9
1.5 Enabling technologies . . . . .	11
1.5.1 Object domain . . . . .	11
1.5.2 Network domain . . . . .	14
1.5.3 Middleware domain . . . . .	15
1.6 IoT key issues and challenges . . . . .	16
1.6.1 Standardization and interoperability . . . . .	16
1.6.2 Scalability and availability . . . . .	17
1.6.3 Security and privacy . . . . .	17
1.6.4 Management and self-configuration . . . . .	18
1.6.5 Modeling and simulation . . . . .	19
1.7 Conclusion . . . . .	20
<b>2 Modeling of IoT systems</b>	<b>21</b>
2.1 Introduction . . . . .	21
2.2 Model-Driven Engineering (MDE) . . . . .	22
2.2.1 Meta-modeling . . . . .	22

2.2.2	Model transformation . . . . .	23
2.2.3	Model Driven Architecture . . . . .	23
2.2.4	Domain-Specific Modeling Language . . . . .	24
2.2.4.1	Abstract syntax . . . . .	25
2.2.4.2	Concrete syntax . . . . .	26
2.2.4.3	Semantics . . . . .	26
2.3	MDE to address the IoT development challenges . . . . .	27
2.3.1	Heterogeneity . . . . .	28
2.3.2	Large-scale and emergent properties . . . . .	28
2.3.3	Context awareness and uncertainty . . . . .	29
2.3.4	Dynamic discoverability of resources . . . . .	29
2.3.5	Reusability . . . . .	29
2.3.6	Security and trust . . . . .	30
2.4	MDE-based approaches for modeling IoT systems . . . . .	30
2.4.1	Approaches with code generators . . . . .	31
2.4.2	Approaches with formal verification tools . . . . .	33
2.4.3	Synthesis . . . . .	34
2.5	Conclusion . . . . .	37
<b>3</b>	<b>The ThingML approach</b>	<b>38</b>
3.1	Introduction . . . . .	38
3.2	ThingML Domain-Specific Language . . . . .	38
3.2.1	Meta-model . . . . .	39
3.2.2	Thing . . . . .	41
3.2.3	The platform-independent action language . . . . .	43
3.2.4	State machine . . . . .	43
3.2.5	Configuration . . . . .	44
3.3	Code generation framework . . . . .	45
3.4	Lacks and limits of the ThingML approach . . . . .	46
3.5	Conclusion . . . . .	47
<b>4</b>	<b>Rewriting Logic and Maude</b>	<b>48</b>
4.1	Introduction . . . . .	48
4.2	Verification techniques . . . . .	49
4.2.1	Test . . . . .	49
4.2.2	Simulation . . . . .	49
4.2.3	Formal verification techniques . . . . .	50
4.3	Rewriting Logic . . . . .	51
4.3.1	Rewrite theory . . . . .	51
4.3.2	Deduction rules . . . . .	52
4.4	Maude language . . . . .	53
4.4.1	Functional module . . . . .	53
4.4.2	System module . . . . .	55
4.4.3	Simulation and analysis in Maude . . . . .	56
4.4.3.1	Rewriting and search . . . . .	57
4.4.3.2	The Maude's LTL model-checker . . . . .	57
4.5	Executable operational semantics in Maude . . . . .	58
4.5.1	Syntax definition . . . . .	59
4.5.2	Big-step semantics . . . . .	61
4.6	Conclusion . . . . .	64

<b>5</b>	<b>MDE-based formal approach</b>	<b>65</b>
5.1	Introduction . . . . .	65
5.2	General overview . . . . .	65
5.3	Formalization of ThingML constructs . . . . .	66
5.3.1	Thing . . . . .	68
5.3.2	Messages and ports . . . . .	69
5.3.3	Platform-independent language . . . . .	70
5.3.3.1	Expressions . . . . .	70
5.3.3.2	Action language . . . . .	72
5.3.4	State machine . . . . .	73
5.3.5	Configuration . . . . .	74
5.4	ThingML2Maude: A translator tool of ThingML models to Maude . . . . .	76
5.5	Case study . . . . .	78
5.5.1	Specification . . . . .	79
5.5.2	Transformation . . . . .	83
5.5.3	Simulation and analysis . . . . .	85
5.5.3.1	Rewriting . . . . .	85
5.5.3.2	Search . . . . .	87
5.5.3.3	Linear Temporal Logic Model Checking . . . . .	87
5.6	Conclusion . . . . .	91
<b>6</b>	<b>A simulation-based MDE approach</b>	<b>93</b>
6.1	Introduction . . . . .	93
6.2	General overview . . . . .	93
6.3	Underlying technologies . . . . .	95
6.3.1	Eclipse Modeling Project (EMP) . . . . .	95
6.3.1.1	Eclipse Modeling Framework (EMF) . . . . .	95
6.3.1.2	Sirius Framework . . . . .	95
6.3.1.3	Xtext Framework . . . . .	96
6.3.2	Arduino platform . . . . .	96
6.3.3	Proteus software . . . . .	96
6.4	The hybrid graphical-textual modeling editor . . . . .	97
6.4.1	Xtext-based editor . . . . .	98
6.4.2	Sirius-based editor . . . . .	98
6.5	Case study . . . . .	99
6.5.1	Specification . . . . .	99
6.5.2	Code generation . . . . .	102
6.5.3	Compilation of the sketches . . . . .	103
6.5.4	Simulation . . . . .	103
6.6	Conclusion . . . . .	104
	<b>General conclusion</b>	<b>105</b>
	<b>Bibliography</b>	<b>xv</b>
	<b>Appendix A</b>	<b>xxviii</b>
A.1	Abstract syntax for the ThingML action language . . . . .	xxviii
A.2	Evaluation semantics for expressions . . . . .	xxix
A.3	Evaluation semantics for ThingML actions . . . . .	xxx

---

<b>Appendix B</b>	<b>xxxix</b>
B.1 The THINGML-CONVERSION module . . . . .	xxxix
B.2 The THINGML-EXP-SYNTAX module . . . . .	xxxix
B.3 The THINGML-SYNTAX module . . . . .	xxxix
B.4 The THINGML-STORE module . . . . .	xxxix
B.5 The THINGML-AP module . . . . .	xxxix
B.6 The THINGML-EXP-EVALUATION module . . . . .	xxxix
B.7 The THINGML-ACTION-SEMANTICS module . . . . .	xxxix
B.8 The THINGML-MSG-SEMANTICS module . . . . .	xxxix



# List of Tables

---

1.1	IoT enabling technologies and functional blocks [1]. . . . .	12
2.1	A comparative study of approaches to modeling IoT systems. . . . .	36
3.1	The syntax of platform-independent action language in BNF. . . . .	43
3.2	Platforms supported by the code generation framework [2]. . . . .	45
5.1	Summary of the correspondences between the main ThingML and Maude constructs. . . . .	67
5.2	Description of the atomic propositions . . . . .	88
6.1	The simulation results . . . . .	104

# List of Figures

---

1.1	IoT applications . . . . .	7
1.2	Three-layer (a) and five-layer (b) architectures of IoT. . . . .	10
2.1	The Meta-modeling concepts. . . . .	22
2.2	Basic concepts of model transformation. . . . .	23
2.3	The Modeling Language definition [3]. . . . .	25
3.1	An excerpt of the ThingML meta-model. . . . .	40
3.2	An excerpt of the ThingML meta-model (the state machine part). . . . .	40
3.3	An excerpt of the ThingML meta-model (the configuration part). . . . .	41
3.4	ThingML framework extension points [2]. . . . .	46
4.1	Model Checking Process. . . . .	50
4.2	Abstract syntax for Exp4 . . . . .	59
4.3	Evaluation semantics for arithmetic expressions: $\Longrightarrow_A$ . . . . .	62
4.4	Evaluation semantics for Boolean expressions: $\Longrightarrow_B$ . . . . .	62
5.1	The workflow of the proposed MDE-based formal approach. . . . .	66
5.2	Evaluation semantics for Boolean expression. . . . .	71
5.3	Evaluation semantics for some ThingML actions . . . . .	72
5.4	The message routing . . . . .	75
5.5	Automatic transformation process. . . . .	77
5.6	The statechart of the PingServer thing. . . . .	80
5.7	The statechart of the PingClient thing. . . . .	82
5.8	Execution result of the fair rewrite command. . . . .	86
5.9	Execution result of the rewriting to a terminal state. . . . .	86
5.10	Result of the search command on states with counter > count-max. . . . .	87
5.11	Result of the search command on a state with counter = 5. . . . .	87
5.12	Verification result of the 1st property. . . . .	89
5.13	Verification result of the 2nd property. . . . .	89
5.14	Verification result of the 3rd property. . . . .	90
5.15	Verification result of the 4th property. . . . .	90
5.16	Verification result of the 4th property on the modified specification. . . . .	91
6.1	The workflow of the simulation based-approach. . . . .	94
6.2	(a) Arduino IDE. (b) Arduino Uno board. . . . .	97
6.3	The structure of the state machine Viewpoint . . . . .	99
6.4	Graphical view of Things in the Traffic_Light application. . . . .	100
6.5	Graphical view of thing Traffic_Light state chart. . . . .	101
6.6	Graphical view of the Traffic_Light_App configuration. . . . .	102
6.7	The code generation using Jar file. . . . .	103

---

6.8	The compilation of traffic light specification. . . . .	103
6.9	The traffic light hardware circuit . . . . .	104

# Acknowledgements

---

El-Hamdou Li ALLAH the Almighty, for giving me the moral and physical strength to complete this thesis.

The research presented in this thesis was conducted under the supervision of Dr. Kerkouche Elhillali and Pr. Allaoua Chaoui. I want to express my deep gratitude to them for guiding and supporting me throughout this research work. I thank them for their availability and patience, all the advice they gave me, and the time they took to supervise me. Many thanks.

Thank Pr. Melit Ali, for giving me the honor of accepting to chair the jury of this thesis.

I also thank Pr. Bourouis Abdelhabib, Pr. Lemouari Ali, as well as Dr. Khelfaoui Khaled, for having accepted to participate in the jury of this thesis.

I want to thank my family, colleagues, and friends for their moral support.

Finally, I want to express my gratitude to the people who supported me in one way or another and contributed in one way or another during the realization of my doctoral thesis.

---

**January, 2023**

**Abdelouahab FORTAS**

# Abstract

---

## ملخص

أنظمة إنترنت الأشياء (IoT) عبارة عن مجموعات معقدة من المكونات التي تتعاون لتحقيق أهداف مشتركة. تركز هذه المكونات على تقنيات مختلفة غير متجانسة وتتواصل مع بعضها البعض باستخدام بروتوكولات اتصال مختلفة. هذا التباين يجعل تصميم وتطوير تطبيقات إنترنت الأشياء مشكلة صعبة. تم اقتراح مناهج متنوعة تعتمد على الهندسة الموجهة بالنموذج (MDE) للتغلب على هذه المشكلة الرئيسية وذلك باستخدام لغات نمذجة مناسبة. ThingML هو إمتداد للغة UML واعد لنمذجة تطبيقات إنترنت الأشياء يهدف إلى مواجهة تحديات عدم التجانس. ومع ذلك ، لا يحتوي ThingML على دلالات صارمة ، مما يجعله غير مناسب للتحقق الدقيق وتحليل تصميمات النظام. كما أنه يفتقر كذلك إلى الأدوات اللازمة لاختبار الكود الذي تم إنشاؤه قبل استعماله في أجهزة إنترنت الأشياء. في هذه الأطروحة ، نقترح نهجًا دقيقًا قائمًا على MDE لتحديد دلالات دقيقة للغة ThingML باستخدام منطق إعادة الكتابة ولغتها Maude . بالإضافة لذلك، قمنا بتطوير محرر نصي - رسومي للغة ThingML ونقدم نهجًا قائمًا على المحاكاة لاختبار كود المصدر الذي تم إنشاؤه بواسطة خاصية إنشاء الكود لـ ThingML . الأساليب المقترحة تم توضيحها بواسطة دراسات حالة.

**كلمات مفتاحية :** إنترنت الأشياء، التحقق الدقيق، منطق إعادة الكتابة، لغة مود، الهندسة النموذجية.

## Abstract

Internet of Things (IoT) systems are complex assemblies of components that collaborate to achieve common goals. These components are based on different heterogeneous technologies and communicate with each other using various communication protocols. This heterogeneity makes the design and development of IoT applications a challenging issue. Diverse approaches based on Model-Driven Engineering (MDE) have been proposed to overcome this major issue using suitable modeling languages. ThingML is a promising UML profile for modeling IoT applications that aims to address the challenges of heterogeneity. However, ThingML does not have rigorous semantics, which makes it unsuitable for formal verification and analysis of system designs. It also lacks tools to test the generated code before deploying it in IoT

devices. In this thesis, we propose an MDE-based formal approach to define a formal semantics for ThingML language using Rewriting Logic and its language Maude. In addition, we develop a hybrid textual-graphical editor for the ThingML language and we present a simulation-based approach to test the source code generated by the ThingML code generation framework. The proposed approaches are illustrated through case studies.

**Key Words :** *Internet of Things, Formal verification, Rewriting logic, Maude language, Model Driven Engineering.*

## Résumé

Les systèmes de l'Internet des objets (IoT) sont des assemblages complexes de composants qui collaborent pour atteindre des objectifs communs. Ces composants sont basés sur différentes technologies hétérogènes et communiquent entre eux à l'aide de divers protocoles de communication. Cette hétérogénéité fait de la conception et du développement d'applications IoT un véritable défi. Diverses approches basées sur l'Ingénierie Dirigée par les Modèles (IDM) ont été proposées pour surmonter ce problème majeur en utilisant des langages de modélisation appropriés. ThingML est un profil UML prometteur pour la modélisation des applications IoT qui vise à relever les défis de l'hétérogénéité. Cependant, ThingML ne possède pas de sémantique rigoureuse, ce qui le rend inadapté à la vérification et à l'analyse formelles des conceptions de systèmes. Il manque également des outils pour tester le code généré avant de le déployer dans les dispositifs IoT. Dans cette thèse, nous proposons une approche formelle basée sur l'IDM pour définir une sémantique formelle pour le langage ThingML en utilisant la logique de réécriture et son langage Maude. En outre, nous développons un éditeur hybride textuel graphique pour le langage ThingML et nous présentons une approche basée sur la simulation pour tester le code source généré par le cadre de génération de code ThingML. Les approches proposées sont illustrées à travers des études de cas.

**Mots clés :** *Internet des Objets, Vérification formelle, Logique de réécriture, Langage Maude, Ingénierie Dirigée par les Modèles.*

# General introduction

---

TODAY, the Internet of Things (IoT) is undergoing a remarkable development where the number of devices linked to the Internet has reached tens of billions, and the number is expected to increase continuously [4]. Its use covers various areas of life, such as smart homes, industry, transportation, health, and others. The development of modern technologies such as wireless sensor networks and identification systems has helped to provide traditional objects with new features such as identification, capture, communication, and computation [5]. These features have been transformed from these traditional things to intelligent things that can collect information from their environment, perform calculations, and communicate with each other to achieve specific goals [6].

The IoT systems are based on heterogeneous hardware components ranging from microcontrollers to powerful cloud servers. Heterogeneity includes variability in resources, technologies and protocols of communication, hardware and software platforms, data formats, and programming languages. Such heterogeneity, with the lack of standardized software solutions, makes the design and the development of IoT applications challenging [7, 8].

The Model-Driven Engineering (MDE) approach can help to surmount IoT applications developments' technical challenges [8]. In MDE-based methods, the *model* represents an essential artifact that can describe the system at a level of abstraction to facilitate system understanding and analysis. The models describe the specified systems according to the used modeling language's *meta-model*. Furthermore, *model transformation* techniques are used to manipulate models and mappings between models. They are used for different activities such as the translation of models (expressed in different modeling languages), optimization of models, generating code from models,

... . In MDE-based approaches, model transformation aims to save efforts and to reduce human errors by automating model manipulation.

Diverse MDE-based approaches have been proposed to overcome IoT applications design and development issue by using suitable modeling languages. ThingML [9] is a promising UML profile for modeling IoT applications that aims to address the challenges of heterogeneity and distribution. It consists of concepts (things, messages, ports, state machines, platform-independent action language, and configurations) that provide straightforward modeling of IoT applications. Moreover, the ThingML language includes a highly customizable framework that supports automatic code generation. The code generation framework can transform the ThingML models into operational code in various programming languages (Java, Javascript, and C / C++) [9]. Therefore, the time and effort needed to develop IoT applications can be significantly reduced.

However, the ThingML Domain-Specific Language (DSL) provides a textual syntax to describe IoT applications. It describes the dynamic behavior of components using a mix of state charts, communication by asynchronous messages, a platform-independent action language, and target languages. Therefore, these specifications can include many details that decrease their legibility. On the other hand, the ThingML approach lacks tools to test and verify generated code before deployment on devices. In addition, ThingML does not have rigorous semantics to support formal reasoning about system designs. Consequently, detecting unwanted behaviors becomes extra complicated, notably for mission-critical IoT systems where reliability is necessary because failure is potentially catastrophic.

Formal methods are effective techniques for analyzing such systems [10]. Rewriting Logic [11] provides a powerful formal method to describe and analyze concurrent and distributed systems. It has a well-defined semantics that can formally represent a wide range of languages [12–14]. There are several language implementations of Rewriting Logic, such as Maude [15]. Maude is a promising system and language for formally defining the semantics of programming languages [15]. It provides powerful verification tools, including simulation and Linear Temporal Logic (LTL) Model Checking [16].

The principle contributions of this thesis are:



The first and major contribution [17] is proposing a formal approach to verify the ThingML designs using Rewriting Logic [11]. This approach transforms the ThingML designs into Maude's Rewriting Logic language [15]. The main advantage of this approach over other approaches lies in the universality and versatility of Maude's mathematical notation, which implements all ThingML concepts and their behavioral aspects (components, instances, configurations, state machines, communications by asynchronous messages, action language ...) in a unified formal logic. In addition, the existing Maude language verification tools provide powerful analysis techniques, including symbolic simulation and model checking, which enable rigorous analysis and verification of ThingML designs. The approach proposes a semantics mapping between ThingML concepts and Maude constructs. To this end, we have defined Maude structures to describe all ThingML components and their behavioral aspects. We have also defined and implemented an operational semantics [18] for the ThingML action language in Maude. Finally, we have developed a tool based on the Acceleo framework [19] that automatically transforms ThingML specifications into the corresponding Maude models.

The second contribution [20] is developing a hybrid textual-graphical editor for the ThingML language and proposing a simulation-guided approach based on the Proteus software [21]. This approach allows testing and verifying the source code generated by the ThingML generation framework before deployment on IoT devices. It should be noted here that both approaches can be integrated to verify and to test the developed artifacts at two different levels of abstraction. The presence of bugs in the ThingML specification implies the presence of bugs in the resulting code. At the same time, a bug-free ThingML specification does not necessarily mean a bug-free source code. It may include some through the code generator itself. This is why we propose to use simulation techniques to verify the generated code. It is worth mentioning that through the second contribution, we presented the simulation of the Arduino code [22] in the Proteus program. However, the same approach can be generalized to verify the resulting code for other languages or platforms, and this can be done by using similar or new tools adapted to the languages and platforms.

### **Organization of the thesis**

This thesis contains six chapters:

- The first chapter presents generalities on the IoT systems, describing their application domains and architectures. We present the IoT-enabling technologies and end with key open problems and challenges such as standardization and Interoperability, scalability and availability, modeling and simulation,...
- In the second chapter, we discuss the modeling and the analysis of IoT systems using the MDE approach. We first present the essential concepts of the MDE as well as the terminology we will use throughout this manuscript. Then, we present some challenges of developing IoT systems that can be addressed using the MDE. The last part of the chapter discusses some MDE-based approaches for modeling and analysis of IoT systems.
- In the third chapter, we focus on the ThingML promoter approach. We start by presenting the ThingML domain-specific language and the main concepts of this language, and then we demonstrate the code generation framework. Finally, we present some lacks and limits of the ThingML approach.
- The fourth chapter focuses on rewriting logic and their language Maude. In the first part, we present the essential concepts of rewriting logic and give a detailed syntax of the Maude language and the characteristics of its environment. Then, the second part details the implementation of the executable operational semantics in Maude.
- The fifth chapter describes the proposed formal approach to verify and analyze ThingML specifications. We will detail the formalization of ThingML constructs using the Maude language. Then, we will present the ThingML2Maude tool, a model-text translator allowing to translation of ThingML models into Maude code. Finally, we will illustrate our approach through a case study.
- In the sixth chapter, we will present an MDE and simulation-based approach to testing IoT applications without the availability of devices. We will also present a hybrid modeling editor for ThingML to facilitate the development process. Finally, we will illustrate our approach through a case study.
- Finally, our thesis ends with a general conclusion summarizing the main points addressed in this thesis and research perspectives.

---

# CHAPTER 1

---

---

## INTERNET OF THINGS: AN OVERVIEW

### 1.1 Introduction

THE Internet of Things (IoT) is one of the core technologies of current and future information and communications technology sectors. IoT technologies will be deployed in numerous industries, including: health, transport, smart cities, utility sectors, environment, security, and many other areas. In this chapter, we introduce the concept of the Internet of Things. At the beginning, we provide a complete definition and a general overview of the impact of IoT on our societies through its different applications. After that, we present the enabling technologies that are expected to form the building blocks of the IoT. Finally, we describe IoT architectures and present the main challenges and difficulties encountered in developing IoT applications.

### 1.2 Definitions

The Internet of Things (IoT) is one of the significant communication developments of recent years. It makes our everyday objects (e.g., health sensors, industrial equipment, vehicles, clothes) connected to each other and to the Internet to achieve common goals. The definition of IoT is still the subject of debate, as there is still no universal and unified definition for this concept. The lack of clarity of this term may be due to its association with two concepts or terms (Internet & Things). The Internet can be defined as *a global network of interconnected computer networks based on the TCP/IP communication protocol*. A *Thing* is something that is not precisely identifiable. It may be anything imaginable with features such as identifying, sensing, actuation, and the ability to communicate. That gives the possibility to integrate them into the IoT environment.

Consequently, the IoT has been defined in many different ways. Van Kranenburg's definition [23] is among the first definitions of IoT. It defines this concept as *"a dynamic global network infrastructure with self-configuring capabilities based on interoperable communication protocols where virtual and physical things have an identity, physical attributes, and virtual personality and that use intelligent interfaces embedded in an information network"*.

More straightforwardly, Coetzee and Eksteen [24] define IoT as *"a vision where objects become part of the Internet, where all objects are uniquely identified and accessible through the network, their positions and statuses are known, where the notion of intelligence is added to the Internet thus merging the digital and physical world, impacting all professionals, people, and the social environment"*. Another definition by Vermesan et al. [25] defines the Internet of Things as simply *"an interaction between the physical and digital worlds. The digital world interacts with the physical world using many sensors and actuators"*.

According to [26], the basic concept behind IoT is the pervasive presence around us of various wireless technologies, such as Radio-Frequency IDentification (RFID) tags, sensors, actuators, and mobile phones, in which computing and communication systems are seamlessly embedded. These objects interact with each other through unique addressing schemes and cooperate to reach common goals. In common parlance, IoT refers to a new world where almost all devices and appliances are connected to a network. We can use them collaboratively to achieve complex tasks requiring high intelligence [27].

### 1.3 Applications of IoT

IoT has a huge potential for developing intelligent applications in almost every vertical market. IoT applications provide a set of functionalities and capabilities that can be grouped according to the domain of utilization into four areas: monitoring (devices condition, environment state, notifications, alert, etc.), control (control of devices functions), optimization (device performances, diagnostics, repair, etc.) and autonomy (autonomous operations). Today, the IoT covers a wide range of applications. It covers almost all areas of daily life, allowing the emergence of intelligent spaces (see Figure 1.1).

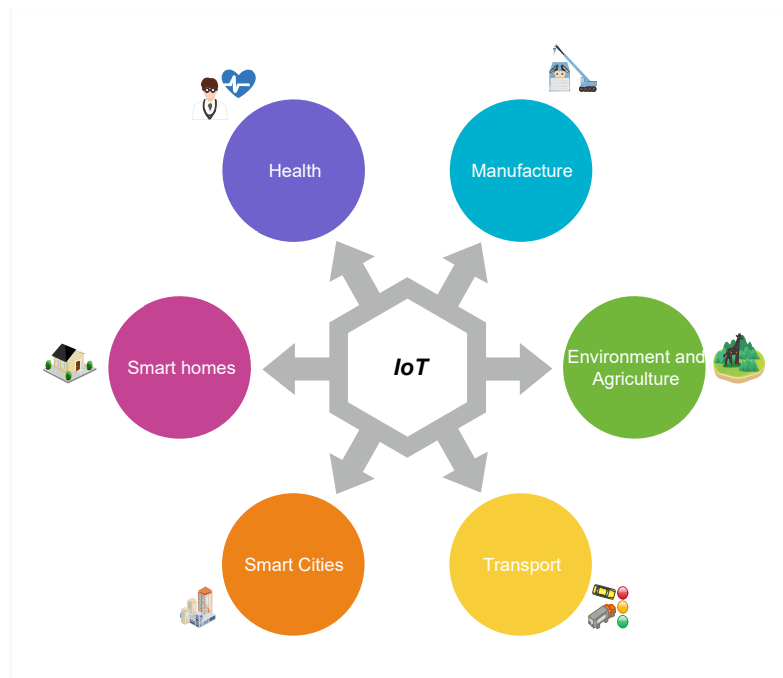


FIGURE 1.1: IoT applications

### 1.3.1 Smart homes

Smart homes are one of the most popular IoT applications [28]. Its idea is to design a home intelligently for the benefit of its residents. It allows them to control and monitor appliances, lighting, heating, ventilation and air conditioning, as well as security systems. Equipping the home with different sensors for light, humidity, and temperature allows the collection of information to generate the optimum climate according to the parameters set by the user. Security systems make the home more secure by automatically detecting and deterring intrusions using a variety of infrared, sound, vibration, and motion sensors, as well as alarm systems [29]. In addition, smart homes and the aids they provide make the elderly and disabled people more comfortable and safe in their homes.

### 1.3.2 Smart cities

IoT will allow better management of the various networks that supply our cities (such as water, electricity, and gas) by allowing continuous real-time and accurate monitoring. Sensors can be used to economize water and energy, improve the management of parking lots and urban traffic, and reduce traffic jams.

### 1.3.3 Energy

The management of electrical grids will be improved thanks to telemetry, allowing real-time management of the energy distribution infrastructure and improving efficiency and productivity. This large-scale interconnection will facilitate fault location, maintenance, consumption control, and fraud detection. In addition, integrating sensors and actuators is likely to reduce the energy consumption of all energy-consuming devices.

### 1.3.4 Health

In the field of health, IoT will enable the deployment of personal networks for controlling and monitoring clinical parameters. This will facilitate the remote monitoring of patients and offer solutions for the autonomy of people with reduced mobility [30]. One popular approach is to use wearable technology. These wearable devices can provide information about clinical body signs, such as heart rate, body temperature, and blood pressure, which can then be transmitted in real-time to a remote site for storage and further analysis. Thus obtaining more and better information about the patients to be able to treat them in a personalized way, without the need to go to a medical center, avoiding transfers and assistance costs, and allowing doctor-patient communication to be less complicated [31].

### 1.3.5 Transport

The IoT can significantly improve transport systems. It will support current efforts around intelligent vehicles for road safety and driver assistance. This will include inter-vehicle communication and communication between vehicles and road infrastructure. IoT will thus be a natural extension of "intelligent transport systems" and their contributions to road safety, comfort, efficient traffic management, and time and energy savings.

### 1.3.6 Manufacture

The application of IoT in the industry is often referred to as Industry 4.0 or the fourth Industrial Revolution. Industry 4.0 builds on cyber-physical systems that tightly integrate machines,

software, sensors, Internet, and users. IoT will enable full tracking of products, from the production chain to the logistics and distribution chain, by supervising supply conditions.

### 1.3.7 Environment and agriculture

By deploying environmental sensors, it will be possible to effectively monitor and measure water and air quality, soil conditions, and hazardous chemicals and radiation. The IoT can contribute to better predictions of earthquakes and tsunamis and earlier detection of forest fires, avalanches, and landslides. All of this will help to preserve the environment better. On the other hand, the IoT can distinguish and track wild animals, especially endangered species, allowing for better study and understanding of these animals' behavior, thus, better protection.

In the agriculture field, sensor networks interconnected to the IoT can be used to monitor the crop environment and health. This will enable better decision support in agriculture, notably in optimizing irrigation water, using fertilizers and plant protection products, and planning agricultural work. These networks can also be used to combat air, soil, and water pollution.

## 1.4 Architecture of IoT

There is no consensus on a unified architecture for IoT, where several architectures have been proposed. However, a three-layer high-level architecture is commonly accepted [32]. It is the most basic architecture and it was introduced in this area's early stages of research [32–34]. This architecture consists of three layers: The perception Layer, Network Layer, and Application layer (see Figure 1.2). A brief description of each layer is given [27]:

- (i) **The perception layer** is the physical layer, whose main task is to perceive and gather the physical properties of the environment. It has sensors to detect specific physical parameters or to identify other intelligent objects. In addition, this layer is responsible for converting the information into digital signals that are more convenient for transmission over the network.
- (ii) **The network layer** ensures connectivity between smart things, network devices, and servers. Its features are also used for the reliable transmission of data generated in the perception layer.

- (iii) **The application layer** constitutes the front end of the IoT architecture through which IoT potential will be exploited. It uses the processed data by the previous layer for delivering application-specific services to the user. Moreover, this layer provides the required tools (e.g., actuating devices) for developers to realize the IoT vision.

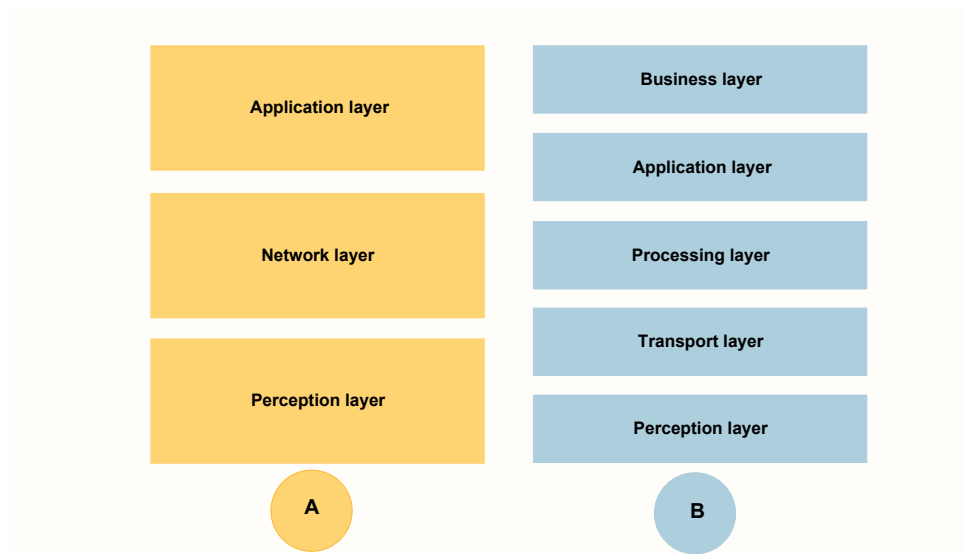


FIGURE 1.2: Three-layer (a) and five-layer (b) architectures of IoT.

The three-layer architecture defines the main idea of the IoT, but it is insufficient for IoT research because it often focuses on finer aspects of the IoT. This is why many other layered architectures are proposed in the literature. One is the five-layer architecture, which includes adding processing and business layers [5, 32–35]. The five layers are perception, transport, processing, application, and business (see Figure 1.2). The role of the perception and application layers is the same as the architecture with three layers. We outline the function of the remaining three layers.

- (i) **The transport layer** is similar to the network layer in the three-layer architecture. It transmits and receives information from the perception layer to the processing layer and vice versa. It contains many technologies such as wireless, 3G, LAN, infrared, WiFi, RFID, NFC, and Bluetooth.
- (ii) **The processing layer** is also known as the *middleware layer*. Its responsibility is to process the data that comes from the transport layer. The processing process has two main aspects storage and analysis. It also can manage and provide a diverse set of services to the lower layers. It is not easy to achieve the objective of this layer due to the large amount of



information collected from the system elements. Therefore, it employs many technologies, such as databases, cloud computing, and big data processing modules.

(iii) **The business (management) layer** manages the overall IoT system activities and services. The responsibilities of this layer are to build a business model, graphs, and flowcharts. Based on the received data from the application layer. It should also design, analyze, implement, evaluate, monitor, and develop IoT system-related elements. The business layer makes it possible to support decision-making processes based on Big Data analysis. In addition, monitoring and management of the underlying four layers are achieved at this layer. Moreover, this layer compares the output of each layer with the expected output to enhance services and maintain users' privacy [5, 32, 35].

## 1.5 Enabling technologies

IoT is not a single technology; it is an agglomeration of various technologies that work together. This section focuses on the enabling technologies expected to form the IoT building blocks. IoT enabling technologies can be summarized into several categories: identification and recognition technologies, sensing technologies, communication technologies and networks, cloud platforms, data processing solutions, security mechanisms, etc. The different technologies are classified into three main domains: object domain, network domain, and middleware domain (see Table 1.1). All these domains include hardware, software, and technologies with specific functionalities and capabilities. In the following, we briefly introduce the building blocks of each category [1, 27].

### 1.5.1 Object domain

The object domain presents the endpoint layer that includes things. These objects have various capabilities, such as sensing, actuation, identifying, data storage and processing, connecting with other objects, and integration into communication networks. IoT objects include embedded software (operating system, onboard application) and hardware (electrical and mechanical components with embedded sensors, processors, and connectivity antennas).

TABLE 1.1: IoT enabling technologies and functional blocks [1].

Domain	Enabling technologies		Functionalities
Middleware domain	Software and API	OS (Contiki, FreeRTOS, LiteOS, Android, Riot OS, uClinux, Mbed...), APIs (JML, WebGL, RAML...), Embedded and custom apps built using «thing» data	Data storage, Data aggregation and processing, Big data analysis, Decision support (Expert and DSS systems), Machine Learning...
	Cloud platforms	OpenIoT, Amazon, Google Cloud, Libellium, IBM Watson, FIWARE, Arkessa, OnePlatform, SensorCloud, SmartThings, ThinkWorks, Oracle IoT, Plotly, Nimbits, ThinkSpeak, Xively, etc.	
	Date processing mechanisms	Data mining, Big Query, Cloud Datalab, Apache Hadoop, Kafka, Storm, RapidMQ, Scribe, SPARQL, SciDB, Semantic technologies (JSON, W3C, OWL, RDF, EXI, WSDL...), etc	
	Data storage	Storage infrastructure (public, private, hybrid), DB (MongoDB, Cassandra, Hadoop, HBase, CouchDB, Redis...), Storage architecture, etc	
Network domain	Communication Protocols	Application(CoAP, MQTT, AMQP, XMPP, DDS, WebSocket, ...), Transport (TCP, UDP), Network(IPv4, IPv6), Service Discovery (mDNS, DNS-SD, SSDP, SLP, ...), etc.	Seamless connectivity (anytime, anywhere by anyone and anything), Data transfer (device-to-device, device-to-application, application-to-device)...
	Network interface	IEEE 802.11, 802.15.4, IEEE 802.15.1, IEEE.802.16, 3GPP, IEEE 802.15.6, WSN, Z-Wave, IEEE 802.3, RFID, NFC, ....	
	Adoption mechanisms	Adoption layer (6Lo WPAN, 6TiSCH, 6Lo, IEEE 1095.1...),	
Device domain	Hardware platforms	Arduino, Raspberry Pi, Intel Galileo, Intel Edison, Beaglebone Black, Netduino, Marvell, Tessel 2, Particle, etc.	Identification, sensing, actuating, data processing and computation, power supply
	Embedded objects	Embedded sensors, Actuators, RFID/NFC tags, Identification(EPC, uCode, QR, ...), touch screen displays, firmware, onboard software, etc.	
	Mechanical & electrical parts	Mechanical and electrical parts (e.g. batteries), Processing units(e.g. microcontrollers, microprocessors), Digital signal processors, Peripheral controller chips, etc.	

*Sensors* are hardware components that capture and measure certain events or changes in their environment, such as heat, light, sound, pressure, magnetism, or a particular movement. They perform various actions to provide an output for further processing. Various sensors are embedded in many objects (e.g., smartphones) to enable IoT-based value-added services. Sensors transmit captured information using electrical signals to the devices to which they are connected. An *actuator* is a device that can change the physical environment. Actuators receive commands from their connected device and translate those electrical signals into actions. Some examples are heating or cooling elements, speakers, lights, displays, and motors. For example, we can consider a smart home system with many sensors and actuators. The actuators are used to lock/unlock the doors, switch on/off the lights or other electrical appliances, alert users of any threats through alarms or notifications, and control the temperature of a home (via a thermostat).

A *unique ID* should identify all devices. There are many identification methods, including EPC (Electronic Product Code), uCode (Ubiquitous codes), and QR (Quick Response). Radio-Frequency IDentification (RFID) technology is an important development in the embedded devices field. RFIID allows the design of tiny microchips (called tags), which can be appended to objects. As a result, stored data in these tags can automatically be used to identify and extract useful information from the object. RFID tags have a unique identifier; the most commonly used is EPC. Many applications from several fields use this kind of tag. Notably in retail, supply chain management, and transportation. They are also used in bank cards, road toll tags as an access control means, in the smart home context, and in hotels to provide automated customer check-in. Another technology with similar identification management is NFC.

Connectivity components enable wireless or wired connections by using different communication technologies which allow an exchange of information between different objects. Sensor Networks (SN) is a collection of sensors that communicate between each other or/and transmit data to another infrastructure (e.g., Fog or/and Cloud). All these capabilities enable objects to be aware of their environment and to exchange data which is one of the goals of IoT. Most IoT products use Wireless Sensor Networks (WSN) solutions. IoT devices may contain gateways that collect data from sensors and send it over the Internet to other infrastructure (e.g., Cloud). They may be connected to other objects or networks via multiple gateways that can act as a proxy

between devices and networks. IoT hardware platforms can facilitate communication, data flow, device, data, and application management.

### 1.5.2 Network domain

The network domain includes hardware, software, technologies, and protocols that enable connectivity between objects and between objects and global infrastructure (e.g., the Internet). As the IoT is proliferating, many heterogeneous smart devices are connecting to the Internet. A wide range of IoT devices are battery-powered, with minimal computing and storage resources. Because of their constrained nature, there are various communication challenges involved, which are as follows [36]:

- (1) *Addressing and identification:* millions of smart things will be connected to the Internet. They will have to be identified through a unique address, based on which they will communicate. We need a large addressing space and a unique address for each smart object.
- (2) *Low power communication:* communication of data between devices is a power-consuming task, especially wireless communication. Therefore, we need a solution that facilitates communication with low power consumption.
- (3) Routing protocols with low memory requirements and efficient communication patterns.
- (4) High-speed and non-lossy communication.
- (5) Mobility of smart things.

IoT devices typically connect to the Internet through the IP (Internet Protocol) stack. This stack is complex and demands a large amount of power and memory from the connecting devices. The IoT devices can also connect locally through non-IP networks, which consume less power, and connect to the Internet via a smart gateway. Non-IP communication channels such as Bluetooth, RFID, and NFC are popular but limited in their range (up to a few meters). Therefore, their applications are limited to small personal area networks. It was necessary to modify the IP stack to facilitate low-power communication and to increase the range of such local networks. One of the solutions is 6LoWPAN, which incorporates IPv6 with low-power personal area networks. The

range of a PAN with 6LoWPAN is similar to local area networks, and the power consumption is much lower. The leading communication technologies in the IoT world are IEEE 802.15.4, low-power WiFi, 6LoWPAN, RFID, NFC, Sigfox, LoraWAN, and other proprietary protocols for wireless networks.

### 1.5.3 Middleware domain

The interoperability of IoT heterogeneous devices needs well-defined standards. However, standardization is difficult because of the varied requirements of different applications and devices. For such heterogeneous applications, the solution is to have a middleware platform, which will abstract the details of the things for applications [1]. IoT middleware is considered a system constrained by software and infrastructure designed to be the intermediary between IoT objects and the application layer. It will hide the details of the smart things. Some IoT middleware provides software (including OS) and Application Programming Interface (API) management while enabling IoT applications to communicate over heterogeneous interfaces. Some common embedded operating systems enable IoT applications' functionalities, such as TinyOS, Contiki, LiteOS, Android, and RIoT OS. These systems support low-power Internet communication and require very few kilobytes of RAM. To summarize, the middleware abstracts the hardware and provides an API for communication, data management, computation, security, and privacy [1, 27]. It can meet a variety of challenges such as [37, 38]:

- (1) **Interoperability:** different types of things can interact with each other easily with the help of middleware services. Interoperability insulates the applications from the intricacies of different protocols, and it ensures that applications are oblivious to different formats, structures, and encoding of data.
- (2) **Device discovery and management:** this feature enables the devices to be aware of all other devices in the neighborhood and the services provided by them. Any IoT middleware needs to perform load balancing, manage devices based on their levels of battery power, and report problems in devices to the users.
- (3) **Scalability:** a large number of devices are expected to communicate in an IoT setup. Moreover, IoT applications need to scale due to ever-increasing requirements.

- (4) **Big data and analytics:** IoT sensors typically collect a large amount of data. It is necessary to analyze all of this data in more details.
- (5) **Security and privacy:** the middleware should have built-in mechanisms to address security and privacy issues in IoT environments that are mostly related to personal life or industry.

There are many middleware solutions available for the internet of Things that address one or more of the aforementioned issues. All of them support interoperability and abstraction, which is the foremost requirement of middleware.

## 1.6 IoT key issues and challenges

The involvement of IoT-based systems in all aspects of human lives and various technologies involved in data transfer between embedded devices made it complex and gave rise to several issues and challenges. These issues are also a challenge for IoT developers in the advanced smart tech society. Therefore, IoT developers need to think of new issues arising and should provide solutions for them.

### 1.6.1 Standardization and interoperability

Diversities in technologies and standards are identified as one of the major challenges in the development of IoT applications [5, 39]. Standardization of IoT architecture and communication technologies is considered a backbone for IoT development in the future [5, 40]. Interoperability is the ability of multiple devices and systems to interoperate regardless of deployed hardware and software. The interoperability issue arises due to the heterogeneous nature of different technology and solutions used for IoT development. The four interoperability levels are technical, semantic, syntactic, and organizational [1, 41].

- **Technical interoperability** is usually associated with communication infrastructure and protocols. IoT systems need to provide interoperability over heterogeneous devices, networks, and a variety of communication protocols such as IPv6, IPv4, 6LoWPAN/RPL, CoAP/CoRE, ZigBee, GSM/GPRS, WiFi, Bluetooth, RFID, etc.

- **Syntactical interoperability** is associated with understanding content (information) and refers to data formats, syntaxes, and codings such as XML and HTML.
- **Semantic interoperability** enables the interpretation of content (the meaning of information) to be shared by communicating parties. The term "semantic" in the IoT refers to the possibility of extracting knowledge from raw data collected from sensors. This "knowledge" enables the provision of useful services and reports based on analyzed data.
- **Organizational interoperability** is usually associated with the ability to exchange data regardless of the different information systems and infrastructure used.

Various functionalities are being provided by IoT systems to improve the interoperability that ensures communication between different objects in a heterogeneous environment. Considering interoperability an important issue, researchers approved several solutions that are also known as interoperability handling approaches [42]. These solutions could be based on adapters/gateways, virtual networks/overlay, service-oriented architecture, etc.

### 1.6.2 Scalability and availability

A system is scalable if it is possible to add new services, types of equipment, and devices without degrading its performance. The main issue with IoT is to support a large number of devices with different memory, processing, storage power, and bandwidth [43]. A great example of scalability is Cloud-based IoT systems which provide sufficient support to scale the IoT network by adding up new devices, storage, and processing power as required. Another important issue that must be taken into consideration is availability. Availability means that IoT applications should be available anywhere and anytime for every authorized object. Availability of the network and its coverage area must enable the continuity of the services to use regardless of mobility, dynamic change of network topology, or currently used technologies.

### 1.6.3 Security and privacy

One of the most important and challenging issues in the IoT is security and privacy due to several threats, cyber-attacks, risks, and vulnerabilities. The issues that give rise to device-level privacy

are insufficient authorization and authentication, insecure software, firmware, web interface, and poor transport layer encryption [41]. Besides, if communication takes place using wireless technologies within the IoT system, it becomes more vulnerable to security risks. Therefore, Security mechanisms must be embedded at every layer of IoT architecture to prevent security threats and attacks [44]. Developing such mechanisms is a difficult challenge, especially since a large part of IoT devices have limited resources (memory, computation). Another issue is the different privacy policies for different objects communicating within the IoT system. Therefore, each object should be able to verify the privacy policies of other objects in the IoT system before transmitting the data.

#### 1.6.4 Management and self-configuration

Managing IoT applications and devices is a very critical factor for successful IoT deployments [45]. Management functionalities such as monitoring, control, and configuration are a big challenge due to IoT complexity, heterogeneity, a large number of deployed devices, and traffic demands. IoT software must be able to identify various smart objects and interact with them to provide efficient management and self-configuration functionalities. Self-configuration means the IoT system has capabilities of the dynamic adoption of changes in its environment. For example, if devices could switch off when there is no activity, it would provide more efficiency in energy consumption [1].

*Data management mechanisms* need to provide various functionalities such as raw data aggregation, data analytics, data recovery, and security. They need to enable different kinds of reports. Another challenge is to provide automatic decisions and self-configuring operations in complex, integrated, and open IoT systems. The objects must acquire knowledge from the collected data and, based on this, perform some context-aware actions.

*Network management* functionalities need to provide efficiency in network topology management, device synchronization as well as traffic and congestion control management. A new network's design needs to deploy efficient management mechanisms to manage the large-scale of connected devices, an enormous amount of data (traffic loads), and various services with different quality of service requirements. Monitoring network infrastructure enables the detection of any changes and events that affect network resource usage and security.

*Devices management mechanisms* need to provide monitoring and remote-control functionalities,



including remote devices' activation or deactivation, firmware update, etc. Managing devices and enabling seamless integration in various networks are challenges due to the deployment of various hardware and software while providing operations such as addressing and optimization at the architectural and protocol levels [46]. The challenge of device management is especially pronounced because of the heterogeneity among devices and associated services. Some other open issues are related to the development of lightweight and secure IoT device management frameworks which provide functionalities such as location awareness, mobility, low power consumption, support for various mobile OS, etc.

### 1.6.5 Modeling and simulation

Major challenges in developing IoT services are due to their complexity and heterogeneity in all parts of system architecture. The heterogeneity embraces both software and hardware. Such diversification is not trivial to handle and is exacerbated by an elemental peculiarity of the IoT: the very same software functionalities are expected to be deployable on different devices, each having only a limited set of common core characteristics. Moreover, things can be small or have limited resources; they can have limited battery capacity, storage resources, or computational capabilities. This adds complexity to deployment and redeployment of software functionalities across devices with different capabilities. This heterogeneity and the lack of standardized platform-agnostic software solutions make cross-platform development intractable [8, 47]. Therefore, IoT system modeling for finding eligible deployments is challenging. On the one hand, the availability of many diverse heterogeneous devices collaborating in the IoT represents an unprecedented opportunity to improve the quality of life, in addition to the quality of service, through collaboration among industrial and consumer devices. On the other hand, we must deal with new challenges at all levels to benefit from the significant advantages the IoT will unleash. Heterogeneity, runtime adaptability, reusability, interoperability, data mining, security, abstraction, automation, privacy, middleware, and architectures are just some of the aspects we need to consider at both design time and runtime and for which new software engineering approaches will be envisioned [8].

Simulation tools such as Opnet, NS-3, Cloudsim, and others can be used for understanding and modeling the IoT system. However, the complexity and heterogeneity of IoT scenarios complicate

these processes. This imposes the use of sophisticated, hybrid, and multi-level modeling and simulation techniques [48]. An overview of some other modeling and simulation challenges has been presented in [49]. For example, one of the main issues in existing simulation tools is the lack of integrated options to simulate network and Cloud infrastructure to obtain the overall performance of IoT systems. Also, there is a problem with simulating various protocols, security attacks, computing, and other IoT processes to obtain different results, such as network performance and energy consumption. Another important issue is enabling the simulation of IoT scenarios, including many heterogeneous devices with various traffic loads and types. This implies that the problem of simulating IoT scenarios is related to software tools and hardware performances that provide enormous resources such as CPU and RAM. According to previous considerations, a new enhancement of simulation tools should improve the ability to simulate small, medium, and large-scale IoT scenarios. Simulation and modeling tools need to support the dynamic nature of IoT, real-time requirements and increasing processing requirements. These scenarios include the deployment of heterogeneous technologies. There is a need for the continuous enhancement of simulation and modeling tools to address these shortcomings.

## 1.7 Conclusion

**T**HE Internet of Things creates new opportunities to develop innovative applications by leveraging existing and new technologies. In recent years, various consumer and industrial IoT applications have been developed and deployed. In this chapter, we have given an overview of the Internet of Things in general. We have seen the general concepts and the different application areas of the IoT. Then, we have presented the different IoT architectures and the enabling technologies used in this context. Finally, we have described the various challenges and issues to be resolved and which are seriously taken up by the scientific community. In what follows, we will focus on the modeling aspect of IoT systems, in particular, the use of Model Driven Engineering (MDE) as a mechanism for designing and analysing IoT applications.

---

## CHAPTER 2

---

# MODELING OF IoT SYSTEMS

### 2.1 Introduction

**S**IGNIFICANT challenges come across developers of the IoT due to their heterogeneous nature. As IoT systems are based on a set of heterogeneous, distributed and intelligent things, it is difficult to handle this diversification aggravated by an elementary peculiarity of the IoT: the very same software features should be deployable on different devices, each with only a limited set of basic common features. Moreover, things can be low devices or have limited resources (computational capabilities, storage resources, or battery capacity). This adds complexity to deployment and redeployment of software functionalities across devices with different capabilities. One promising approach to address these challenges and reduce the complexity of IoT development is Model-Driven Engineering (MDE), which is frequently used in many domains for software development.

The abundance of various hardware platforms available for IoT complicates their development. There is a need for a methodology that enables an efficiently increased level of abstraction to address such systems' complexity and heterogeneity problems. To this end, many researchers believe that MDE is a better solution to overcome challenges such as development complexity, heterogeneity, adaptability, and reusability, and they propose various applications of MDE for IoT development.

In this chapter, we will discuss the modeling and analysis of IoT systems using the MDE approach. At first, we will recall the essential concepts of MDE as well as the terminology we will use throughout this manuscript. Then, we will present some challenges of IoT systems development that can be addressed using MDE-based approaches. Finally, we will discuss some MDE-based approaches for modeling and analyzing IoT systems.

## 2.2 Model-Driven Engineering (MDE)

Model-Driven Engineering (MDE) [3, 50, 51] is a software engineering research area that places the model notion at the center of the development cycle. It focuses on abstract concerns around the models without considering the target technologies. In this paradigm, the source code is no longer considered the central element in the development process but rather an artifact derived from the modeling elements. MDE offers techniques for navigating between levels of abstraction. On the one hand, they are based on meta-modeling, which allows the different aspects of a modeling language to be defined and expressed. On the other hand, model transformation allows the manipulation of models and the mapping from one model to another.

### 2.2.1 Meta-modeling

The MDE approach aims to define a high level of abstraction of developed systems and automate the development process. It is mainly based on three concepts: the model, the meta-model, and the meta-meta-model. A *model* can be defined as a relevant abstraction of the system it models. It must be sufficient and necessary to answer specific questions in place of the system it represents. A model that describes using a modeling language must conform to a meta-model. The modeling language itself must be specified as a model called a *meta-model*. A meta-model is a higher-

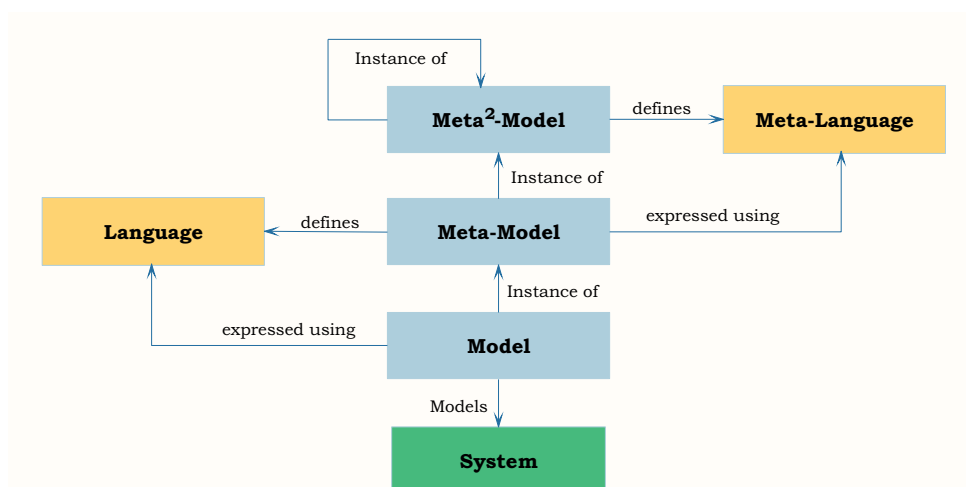


FIGURE 2.1: The Meta-modeling concepts.

order abstraction highlighting the concepts used to define the model. It models the language entities, their relationships as well as their constraints. As with the meta-model, which specifies

the modeling language and interprets models, the *meta-meta-model* has a language description in which the meta-model is expressed [3].

### 2.2.2 Model transformation

In addition to meta-modeling, model transformation is the central element of the MDE approach. It generates one or several target models from source models, where the source and target models conform to their meta-models. The transformation of a source model is done in two steps. The first step identifies the correspondences between the concepts of the source and target meta-models, which induces the existence of a transformation function applicable to all instances of the source meta-model. The second step involves applying the source model's transformation to generate the target model automatically by a program called the transformation engine or execution engine [50].

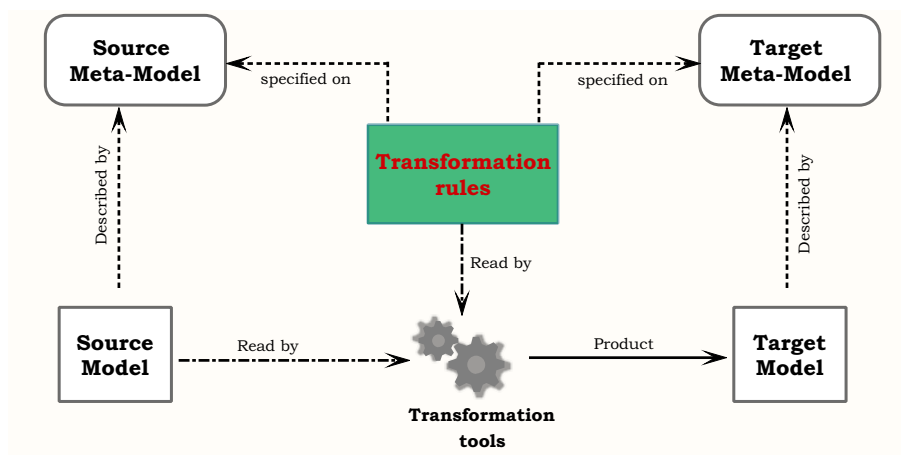


FIGURE 2.2: Basic concepts of model transformation.

### 2.2.3 Model Driven Architecture

OMG (Object Management Group) proposes the Model Driven Architecture (MDA) [52] approach as a framework for model-driven application development. MDA is built on the separation of concerns between an application's logic and the platforms on which it will run. It aims to describe models independently of the technical details of the execution platforms to allow the automatic generation of application code and obtain a significant gain in productivity. In the MDA approach, we can mainly distinguish four classes of models [52, 53]:

- *Computation Independent Model (CIM)*: models the requirements or needs of the system independently of any implementation. The CIM allows the system's vision in the environment where it will operate without going into the realization details or the treatments.
- *Platform Independent Model (PIM)*: it is an analysis and design model that expresses the systems' functioning independently of any implementation technology.
- *Platform Specific Model (PSM)*: is a code model that combines the PIM specifications with platform-specific details. The PSM is essentially used to generate executable code.
- *Platform Description Model (PDM)*: the PDM is the model that describes an execution platform. It provides a set of technical concepts representing the different parts of the platform and (or) the platform and (or) the services it provides.

#### **2.2.4 Domain-Specific Modeling Language**

In MDE-based approaches, the software artifacts are models created using modeling languages. A modeling language could be a general-purpose language or a specific one. A General Purpose Modeling Language (GPML) can express a wide range of systems and domains. When the models are created using the problem domain concepts, we talk about Domain-Specific Modeling [54]. A Domain-Specific Modeling Language (DSML) contains distinctive concepts that allow the description of targeted systems. It has a restricted expressiveness that focuses on a specific domain. The use of dedicated languages significantly facilitates the construction of software systems for the domains to which they are dedicated. DSMLs are generally small and must be easily manipulated, transformed, combined, etc. They are already successfully used in many fields, such as telecommunications, avionics, aerospace, and automotive industries. The interest of DSML is to benefit from the well-known advantages of domain-specific languages [55]:

- Allow solutions to be expressed with idioms at the level of abstraction of the addressed domain. As a result, domain experts can understand, validate, modulate, and often develop programs in dedicated languages.
- Facilitate code documentation.

- Improve quality, productivity, ability, maintainability, portability, and reusability.
- Enable domain-level validation. As long as elements of the language are safe, any sentence written with these elements can be considered safe.

Generally, a modeling language is defined by a set of all possible models conforming to the modeling language's abstract syntax, represented by one or more concrete syntaxes that satisfy a given semantics (see Figure 2.3) [3]. In other words, a modeling language ( $L_m$ ) is defined according to the tuple  $\{AS, CS^*, MA_{ac}^*, SD^*, M_{as}^*\}$  where  $AS$  is the abstract syntax,  $CS^*$  is the concrete syntax(es),  $MA_{ac}^*$  is the set of mappings from the abstract syntax to the concrete syntax(es),  $SD^*$  is the semantic domain(s), and  $M_{as}^*$  is the set of mappings from the abstract syntax to the semantic domain(s) [56].

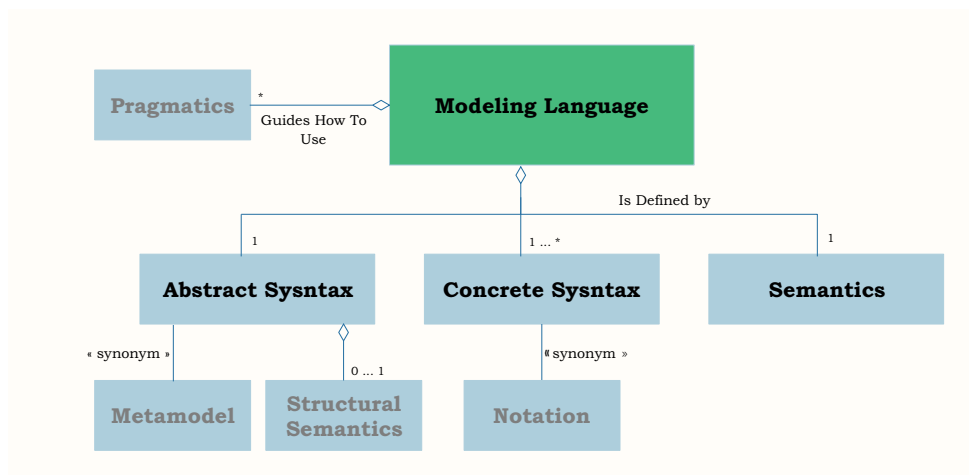


FIGURE 2.3: The Modeling Language definition [3].

### 2.2.4.1 Abstract syntax

The definition of a modeling language typically begins with capturing and identifying the concepts, abstractions, and relationships underlying the application domain. The result of this activity is the abstract syntax of the modeling language. The abstract syntax of a modeling language expresses, in a structural way, the set of its concepts and their relations. It is described using meta-modeling languages that offer the concepts and elementary relations that allow a meta-model to be described in modeling languages. A meta-model represents the abstract syntax of a modeling language through classes linked by different relations (associations, compositions, or specializations). Each

class represents a concept of the dedicated modeling language, i.e., a concept of the domain for which the language is designed.

Abstract syntax also includes “structural semantics” (or static semantics), mainly focused on establishing linking rules between its elements. For example, structural semantics makes it possible to define that an element of type A can be linked to other elements of type B according to this or that constraint. In general, structural semantics can be described in different ways: either by a declarative constraint language (e.g., the Object Constraint Language (OCL) [57]); by an informal natural language specification; or a mixed approach.

#### **2.2.4.2 Concrete syntax**

Concrete syntax provides users with the notations needed to express models. It can be textual, graphical, tabular or form-based, or a combination of these. The definition of concrete syntax allows each modeling language construct defined in the abstract syntax to be annotated with one (or more) concrete syntax decoration(s) that can be manipulated by the language used. In general, graphical notations are most appropriate for illustrating relationships between concepts, changing spatial or temporal distribution values, causal and temporal sequences between events, or data and control flows in process modeling scenarios. However, graphical models are not as scalable as textual or tabular models, which means that they are not the most appropriate for supporting large models; they are also not very applicable for writing or visualizing complex expressions or actions - for this, textual notations tend to be more appropriate. Finally, since a modeling language can have several concrete syntaxes, it would be possible to combine them for the benefit of its users. Several tools can be used to define concrete syntaxes. We mention mainly Sirius [58] for graphical syntaxes and Xtext [59] for textual syntaxes.

#### **2.2.4.3 Semantics**

The semantics of a language denotes in a precise and unambiguous way the meaning of the constructions of this language. It thus allows giving a precise meaning to the programs built from it. A semantic is said to be formal when expressed in a mathematical formalism and allows checking this definition’s coherence and completeness. Based on the concepts and models involved,



semantics can be of two types [3]: executable and non-executable. Non-executable semantics concern concepts that are not directly related to software execution, such as user requirements specification or deployment diagrams. On the other hand, executable semantics concern concepts directly related to the execution order of programs, such as those found in activity, sequence, and state machine diagrams. Most of the existing semantics frameworks are used to define the semantics of executable models (such as UML state machines). Some of these frameworks are “operational semantics” and “translational semantics” [3]:

- **Operational semantics:** it enables the description of the dynamic behavior of the constructs of a language. In the context of the MDE, it aims to express the behavioral semantics of the concepts of the abstract syntax, using an action language, to allow the execution of the models that conform to it. It gives an imperative vision by describing a program through a set of transitions between the states of the execution context. These concepts are devoid of semantics, but an action language makes it possible to express them and thus define the tools supporting the execution. Such a language makes it possible to describe the model’s evolution and produce one or more other states from a given state.
- **Translational semantics:** this is also called denotational semantics. Its principle is to rely on a rigorous formalism to express the semantics of a given language. A translation of the concepts of the original language is then made into this formalism. It is this translation that gives the semantics of the original language. In the context of the MDE, it is a question of expressing transformations towards another technical space, i.e., defining a bridge between the source and target technical spaces. These technological bridges allow using simulation, verification, and execution tools provided by the target technical spaces.

## 2.3 MDE to address the IoT development challenges

The availability of many diverse heterogeneous devices collaborating in the IoT represents an unprecedented opportunity to improve the quality of life, as well as the quality of service, through collaboration among industrial and consumer devices. However, to benefit from the IoT advantages, a whole host of new challenges must be addressed at all levels. Heterogeneity, runtime adaptability,

reusability, interoperability, data mining, security, abstraction, automation, privacy, middleware, and architectures are just some of the aspects we need to consider at both design time and runtime and for which new software engineering approaches will be envisioned. The MDE can help meet the technical challenges of IoT system development and runtime management. Next, we show how MDE techniques and tools can help tackle the challenges of developing IoT applications [8].

### **2.3.1 Heterogeneity**

Heterogeneity is common in IoT systems, where they differ in resources, protocols, hardware and software platforms, programming languages, etc. This heterogeneity and the lack of standardized software solutions make IoT systems development intractable. Thanks to modeling languages, MDE can provide a unique way to represent heterogeneous systems' many aspects in one place. Models defined through these languages can define software with concepts that do not necessarily depend on the underlying platform or technology. Moreover, models can become complex and challenging to grasp when heterogeneity is constantly present, even for experts. MDE offers powerful instruments to support the separation of concerns in multiview modeling—defining and managing models from different design viewpoints.

### **2.3.2 Large-scale and emergent properties**

IoT systems can reach the size of tens or hundreds of distributed things. They should be engineered to automatically scale to accommodate and to take advantage of an arbitrary number of devices. These systems' level of concurrency and complexity might lead them to expose emerging properties that represent unexpected behaviors stemming from both interaction between system parts and the system's interaction with its context. IoT applications' runtime evolution is a challenge. This is particularly difficult if the application operates on code-based artifacts. For example, imagine that a specific functionality of our surveillance system (such as management of suspicious behavior) is implemented for a specific physical device that becomes unavailable at a certain point. Reallocating the functionality to a different type of device would be difficult without modifying the functionality itself. Models@runtime techniques [60] use models and abstractions

of the runtime system and environment to effectively manage the complexity of evolving behaviors during execution.

### **2.3.3 Context awareness and uncertainty**

IoT systems are characterized by uncertainty and unexpected changes in their context. To be able to adapt these changes and thereby cope with uncertainty, things in the IoT system must be designed as adaptive systems. MDE proposes various ways to define adaptive systems and support adaptation under uncertainty. Researchers have proposed mechanisms that automatically generate alternative models to cope with different context conditions. Such mechanisms will identify functional and nonfunctional tradeoffs between the models, thereby dealing with functional and nonfunctional uncertainty [61].

### **2.3.4 Dynamic discoverability of resources**

New, unknown, or recovered devices can appear anytime in dynamic environments such as IoT systems. For the system to exploit them, it must have a mechanism that can dynamically discover the available resources and constraints. The system should be able to recognize, communicate with, and adjust the devices and their characteristics. The discoverability of resources and services provided by newly available devices in the IoT is fundamental. Service-oriented modeling and service-oriented architectures (SOAs) define the use of models and model transformations for identifying, specifying, realizing, composing and orchestrating services [62]. IoT systems can benefit from this mature discipline to exploit models for dynamic discoverability and realize new resources and services.

### **2.3.5 Reusability**

When developing an application very similar to previous ones from scratch, IoT system developers often become frustrated because they do not have the appropriate support for reusability. The lack of a software engineering methodology and comprehensive abstraction mechanisms for handling IoT systems complexity leads to countless similar, but not congruent, isolated solutions that

cannot be easily reused and combined. Systematic reusability is paramount to making IoT software development sustainable in the market, where expectations for new-generation devices grow incredibly. Reusability MDE is often combined with component-based software engineering to define reusable and replaceable self-contained model entities that can be integrated appropriately through architectures, connectors, and integration patterns to describe complex systems [63]. The entities can be modeled as different components, which can be manipulated through dedicated model transformation and analysis tools. Researchers have aimed to exploit the power of models and model transformations to guarantee the runtime preservation of quality attributes in isolation and combination [8].

### **2.3.6 Security and trust**

IoT applications rely on many interconnected components. The intrinsic complexity of such systems, such as the multitude of protocols and APIs, exacerbates security issues, which must be addressed. Thus, securing the decentralized architecture of IoT systems is critical. Security has been addressed mainly in MDE. Exciting is model-driven security [64], which defines system models and their security requirements and uses to generate complete and configured system architectures and access control infrastructures. Model-driven mechanisms for enforcing trust and managing privacy have been studied since MDE's birth. MDE can facilitate defining and enforcing user-friendly, precise, and adequate security and privacy specification policies. It also facilitates the understanding of how violations of such policies might affect the system's parts.

## **2.4 MDE-based approaches for modeling IoT systems**

Many research works have been done using the model-based approach to develop IoT applications. These works can be categorised according to the modeling activity they address, where it could be intended for one or more modeling activities such as development (meta-modeling, transformation, code generation, design process, and others), analysis, validation and verification, simulation, monitoring, adaptability, correctness, and others. This section presents a collection of MDE-based approaches for modeling and verifying IoT systems. We focus on approaches that provide code generation and formal verification tools.

### 2.4.1 Approaches with code generators

The UML4IoT [65] consists of a UML profile representing the basic constructs of the LWM2M protocol [66] and REST architectural paradigm for manufacturing environments. It has implicit execution semantics. UML4IoT provides a code generator of the needed IoT-compliant interface to integrate cyber-physical components into modern IoT manufacturing systems.

Ciccozzi et al. [67] have introduced MDE4IoT as an MDE framework to support the modeling and self-adaptation of emergent configurations of IoT systems. The authors aim to address many challenges, such as system heterogeneity and complexity, using high-level abstraction, collaborative development through separation of concerns and runtime adaptations that are supposed to be performed automatically by model transformations. The MDE4IoT framework consists of UML-based domain-specific profiles. It defines the behavioral aspect of software components using state machines and the Action Language for Foundational UML (ALF) [68]. ALF helps more accessible model validation and analysis and the generation of executable artefacts for heterogeneous targets.

The IoTLink [69] is an IoT application development toolkit that follows a model-driven approach. It enables end-users with minimal development experience to build IoT applications. The IoTLink toolkit comprises a graphical modeling language that encapsulates the heterogeneity and complexity of the IoT applications and a code generator. Using visual notations, IoT developers can specify application components in a platform-independent model, which then can be converted into a platform-specific model. The code generator can translate compliant models into Java source codes.

In [70], the authors have proposed COMFIT (Cloud and Model-based IDE for the Internet of Things) a development environment for the IoT that was built on MDA principles and cloud computing. COMFIT provides an MDA infrastructure (App Development Module) based on UML for designing IoT applications using high abstraction and App management and execution module cloud-based web, including simulators and compilers for developing IoT applications. Using App Development Module, the COMFIT models can be transformed into source code for

two operating systems, Contiki and TinyOS. COMFIT also supports the compilation or simulation of the generated code.

MontiThings [71] is a component and connector ADL (Architectural Description Language) for modeling IoT applications. It describes the behavior of IoT applications using components that exchange data with each other. MontiThings consists of a graphical and textual notation and includes a code generator for the C++ language. It is an extension of the MontiArc [72] language based on the Focus calculus [73]. Consequently, the MontiArc models can be verified [74].

In [75, 76], the authors have presented an MDD framework for IoT applications that addresses the lack of division of roles and the heterogeneity of devices in IoT applications. It presents the IoTsuite toolkit that aims to quickly develop IoT applications by providing automation. IoTsuite supports the automation of tasks at the different phases of IoT application development. In the current implementation, IoTsuite targets Android and JavaSE-enabled devices and MQTT middleware.

ThingML [2] is an approach for developing IoT applications. It consists of a textual DSL and a set of tools. The DSL ThingML is aligned with UML by applying concepts of components, statecharts, and communication by asynchronous messages. It also includes a complete action language to describe the behavior of components. The tools encompass a set of compilers targeting a large set of platforms and communication protocols. It has a wide coverage of target platforms (microcontrollers to servers). The ThingML code generation framework provides a plugin mechanism that can support a wide range of communication protocols such as UART, I2C, MQTT, WebSocket, ROS, and others.

Ihirwe et al. [77] have presented the CHESSIoT component-based modeling approach to support the design and analysis of industrial IoT systems. CHESSIoT provides a means to perform event-based modeling for code generation purposes. The CHESSIoT specifications will be transformed into ThingML models to take advantage of ThingML's code generators. In [78], the authors have proposed an approach to transform CAPS models into ThingML language. It aims to generate code from the CAPS specifications, where CAPS is an architecture-driven modeling framework for IoT systems development. CyprIoT [79] is a framework to model and control

network-based IoT systems. It uses a part of ThingML language to model IoT things and their behaviors. It also uses and extends its code generator framework.

### 2.4.2 Approaches with formal verification tools

Much research has been proposed to verify and analyze the IoT application using rewrite logic and its Maude language. In [80], the authors have presented a transformation from Complex Event Processing (CEP) [81] patterns into RT-Maude [82] specifications in order to help developers to verify and to analyze the program's properties. According to the authors, CEP is becoming essential in diverse contexts, such as IoT systems. In [83], the authors present an approach for analyzing Industrial IoT applications. They have proposed transforming the underlying IEC 61499 standard specifications into Business Process Model Notation (BPMN) models. The BPMN models enable quantitative analysis of process models. This analysis is achieved by transforming the business processes to Maude's specifications. Abbas et al. [84] have proposed an extension of BPMN 2.0 to model IoT applications and a based-Accleo tool to transform the extended BPMN 2.0 diagrams into Maude code for analysis purposes. The Maude specifications obtained are used for model simulation, analysis, and verification. Duran et al. [85] have proposed an approach for supporting the reconfiguration of rule-based IoT applications. The proposed approach enables a comparison of two versions of an application (before and after reconfiguration) to check if several functional and quantitative properties are satisfied. The analysis techniques have been implemented using formal languages. Where Maude language and its tools have been used to check the functional properties, the LNT language [86] and the CADP [87] analysis tools have been used to analyze quantitative properties. These techniques have been integrated into the WebThings platform [88].

On the other hand, many works propose analyzing IoT applications using different formal methods. Costa et al. [89] have presented an MDE-based method to design and analyze IoT applications. The authors have proposed the SysML4IoT modeling language for IoT systems. SysML4IoT is a SysML profile based on the IoT-A Reference Model [90]. After that, they defined a mapping from SysML4IoT models into NuSMV model checker input language [91] using the Accleo framework to verify the developed system's quality of service properties. Xu et al. [92]

have proposed extending ThingML to allow designers to model performance variations that are affected by uncertain external environments. The obtained models are transformed into a Network of Priced Timed Automata (NPTA) [93] for quantitative quality of service analysis of IoT applications using UPPAAL-SMC. Oquendo et al. [94] consider IoT systems a System-Of-System (SoS) class. They have suggested using SosADL language to describe the software architecture of IoT systems. SosADL is an SoS Architecture Description Language (ADL) based on  $\pi$ -Calculus theory. In the literature, SosADL designs have been transformed into several languages, such as DEVS [95] for simulation and UPPAAL for model checking. In [96], the authors have proposed an MDE-based approach to address security problems in IoT application. It proposes a pre-execution verification method for IoT applications to meet security and safety requirements. The authors introduce a domain-specific modeling language to describe IoT applications and a code generator to convert the models into Lustre programs that can be checked using the Kind 2 [96] model-checker.

### 2.4.3 Synthesis

Thanks to the biographical study on MDE-based approaches proposed for modeling IoT systems, it is clear that these methods have great diversity and heterogeneity in terms of techniques used, tools and even objectives. This is due to the heterogeneity in the IoT regarding software, hardware or different enabling technologies. It will not be easy to compare these approaches, but a comparison can be made based on some requirements for approaches to generating operational source code. In [7, 51], the authors stated a set of requirements that a modeling framework must meet to enable accurate representation and simulation of systems in general. In addition, we propose two other requirements (the R2 and R4 elements below) as specific requirements for the code generation framework. In the following, we introduce each requirement and we analyse its fulfillment by these IoT modeling approaches.

- **Well-Defined Notation(R1):** aiming to enable the representation of IoT systems. The first requirement that a modeling framework must meet is to provide a well-defined notation. It must cover the elements of its meta-model and provide a tool to support the specification of IoT systems using the DSL [7].



- **Supported platforms/languages (R2)**: in a heterogeneous environment like IoT systems in which system roles may attach to devices with different platforms. It is beneficial to build frameworks that allow generating code for multiple target platforms/languages. In addition, provide mechanisms to address the diversity of communication protocols, where IoT systems are distributed systems based on these communication protocols.
- **Extensibility and customization (R3)**: the third requirement that a modeling framework shall address is a clear definition of how to extend and customize it to allow representing other aspects that are not covered by the DSL or to efficiently and easily customize parts of the code generation process according to the developed applications' peculiarities.
- **Generated code quality (R4)**: code generation is not popular among practitioners. This bad reputation is typically based on experiences with tools producing code with low readability, hard to integrate with existing systems and other components and very hard to maintain and (or) evolve. It is complicated to judge the quality of the generated code, and this may require expert opinions in the target language/platform, real-world case studies or through V&V (verification and validation) or simulation tools. It is also necessary to evaluate the generated codes in terms of memory usage, execution time, readability, and traceability.
- **Explicit Execution Semantics (R5)**: to execute and simulate the model allowing answering questions at design-time, the DSL must provide the execution semantics explicitly, avoiding applying translational approaches.

Through the analysis of the research works done in the field of IoT systems modeling in general and those specifically interested in code generation, we propose a comparative table (see Table 2.1) based on the requirements mentioned above. The analysis of how the surveyed modeling frameworks fulfill the abovementioned requirements is presented below.

Despite the diversity of approaches and contributions, we note that no single solution meets all the criteria considered. The requirement R1 "Well-Defined Notation" is met by all the modeling frameworks studied. The majority provide a graphical notation based on UML representing the domain concepts. Some of these works relied on textual notation, such as [2], while others adopted

TABLE 2.1: A comparative study of approaches to modeling IoT systems.

Framework	R1	R2	R3	R4	R5
UML4IoT [65]	✓	✗	✗	* <sup>1</sup>	✗
MDE4IoT [67]	✓	✗	✗	* <sup>1</sup>	✓
IoTLink [69]	✓	✗	✗	* <sup>1</sup>	✗
COMFIT [70]	✓	*	✗	* <sup>1,2</sup>	✗
MontiThings [71]	✓	✗	✗	* <sup>1</sup>	✓
IoTSuite [75, 76]	✓	*	✗	* <sup>1</sup>	✗
ThingML [2]	✓	✓	✓	* <sup>1,3</sup>	✗

✓ = requirement fulfilled; ✗ = requirement not fulfilled; \* = requirement partially fulfilled;  
<sup>1</sup>: Case studies ; <sup>2</sup>: Simulation ; <sup>3</sup>: Industrial projects.

a mixed graphical and textual notation [71]. Regarding the second requirement (R2) - "Supported platforms/languages", the result is that, except ThingML, none of the studied approaches provides a code generator that supports multiple languages/platforms and communication protocols. ThingML supports three programming languages and nearly ten platforms and provides a plugin mechanism to deal with a wide range of communication protocols. The IoTSuite and COMFIT frameworks partially fulfil this requirement, as IoTSuite targets Android and JavaSE-enabled devices while COMFIT supports devices running on the TinyOS and Contiki operating systems.

Concerning the third requirement (R3) – “Extensibility and customization,” none of the surveyed approaches provides clear specifications of extensibility rules to extend the DSLs. In addition, if we exclude ThingML, none of the studied modeling frameworks provides a clear definition of extending and customizing the code generation process according to the developed applications’ peculiarities. The code generation framework of ThingML has a modular structure that allows for the customization of some extension points. It identifies ten different extension points, where each extension point is basically an interface (or abstract class) in the code generation framework with a set of methods responsible for generating the code associated with a given model element. For requirement R4, the studied frameworks have demonstrated the quality of the code through numerous case studies. The COMFIT framework supports the simulation of generated code. For its part, ThingML has been used and evaluated in commercial and industrial projects. However, there is still a need to provide means to verify the operability of the generated code while studying

its memory consumption and execution time, especially considering that many IoT devices have limited memory and energy capacities, which require optimized code.

Finally, regarding the fifth requirement (R5) –“Explicit Execution Semantics” – we identify that only MDE4IoT [67] and MontiThings [71] provide explicit execution semantics. MDE4IoT is based on the ALF action language to express the behavioral semantics of the system. MontiThings has formal semantics based on Focus calculus. Moreover, all the other surveyed approaches provide DSLs with implicit execution semantics; thus, to execute and simulate the system behavior, it would be necessary to deploy the components in the platforms and verify the various design alternatives at runtime. Alternatively, it would require translating the system description into a formal language to execute it.

## 2.5 Conclusion

In this chapter, we have discussed modeling IoT systems using MDE-based approaches. First, we have presented in brief the basic concepts of MDE. Then, we have presented and identified some challenges of modeling IoT systems, and we have also seen the different mechanisms provided by MDE to address these challenges. Finally, we have presented some model-based approaches for modeling and analysis of IoT systems with a comparative framework for these approaches. Through this comparison, it was found that ThingML is a promising approach to modeling IoT systems independently of the target platforms. The ThingML approach provides a code generation framework that supports multiple languages, platforms and communication protocols. This generation framework is also customizable to adapt the specificities of the applications or to support new platforms or languages. In the next chapter, we will present the ThingML approach in detail.

---

## CHAPTER 3

---

---

# THE THINGML APPROACH

### 3.1 Introduction

IoT systems are complex assemblies of heterogeneous components, frameworks, and services that are reused, evolved, customized, and composed to create and evolve applications. To address this complexity, MDE must provide solutions that adapt to this reality, focus on specific problems, and integrate into the software development process. It can be beneficial for many purposes, such as increasing productivity by automatically generating code from models. ThingML [2] is an MDE-based approach to address the challenges of heterogeneity in modeling IoT applications. It comprises a Domain-Specific Language (DSL) which combines well-proven software modeling constructs for the design of IoT systems and advanced code generation framework. The code generation framework can transform the ThingML models into operational code in various programming languages (Java, Javascript, and C / C ++), targeting various hardware platforms (from micro-controllers to powerful servers) and multiple communication protocols. Therefore, the ThingML approach is especially beneficial for applications that include heterogeneous platforms and communication channels. In this chapter, we will present the ThingML approach. We start by presenting the ThingML DSL, and the main concepts of this language. Thereafter, we introduce the code generation framework. Finally, we present some lacks and limits of the ThingML approach.

### 3.2 ThingML Domain-Specific Language

ThingML is a UML profile conceived to model IoT applications as a Domain-Specific Modeling textual Language (DSML) [2, 9]. Its textual syntax is defined using the Xtext framework [59] based on the Eclipse Modeling Framework (EMF) [97]. Xtext allows the development of textual DSLs.

It uses an Extended Backus-Naur Form (EBNF)-like language to define the DSL grammar of the developed language. Listing 3.1 represents an excerpt from the ThingML grammar expressed in the EBNF language. This excerpt shows part of the concrete textual syntax of the state machine that consists of states. A state can be a simple, final, or composite state. It can include properties and -internal- transitions, and perform actions on entry and exit. A transition must have a target, and it can have a triggering event, a guard condition, and actions to be performed during the transition.

```

/*****
*     STATE MECHINES
*****/
State returns State:
  StateMachine | FinalState | CompositeState |
  'state' name=ID ( annotations+=PlatformAnnotation )* '{'
    (properties+=Property)*
    (
      ('on' 'entry' entry=Action)? &
      ('on' 'exit' exit=Action)? &
      (properties+=Property | internal+=InternalTransition |
       outgoing+=Transition)*
    )
  '>';
Handler:
  Transition | InternalTransition
;
Transition returns Transition:
  'transition' (name=ID)? '->' target=[State|ID]
  ( annotations+=PlatformAnnotation )*
  ('event' event=Event)?
  ('guard' guard=Expression)?
  ('action' action=Action)?;

InternalTransition returns InternalTransition:
  {InternalTransition}
  'internal' (name=ID)?
  ( annotations+=PlatformAnnotation )*
  ('event' event=Event)?
  ('guard' guard=Expression)?
  ('action' action=Action)?;

```

LISTING 3.1: An excerpt of the ThingML grammar expressed in EBNF [98].

The Xtext framework uses the EBNF grammar to automatically generate a comprehensive text editor and an Ecore-based meta-model of the developed language. The textual editor provides developer help features such as syntax highlighting, error detection markers, and auto-completion.

### 3.2.1 Meta-model

ThingML is an open-source project; its meta-model, editors, and associated toolset of code generation are available in [98]. The ThingML DSL mainly relies on two structures [2]: *Thing* and *Configuration*, where Thing represents the application components, and Configuration describes the instantiation of these components and their interconnection. Moreover, the component behaviors are modeled using *Statecharts* and an imperative platform-independent action language.

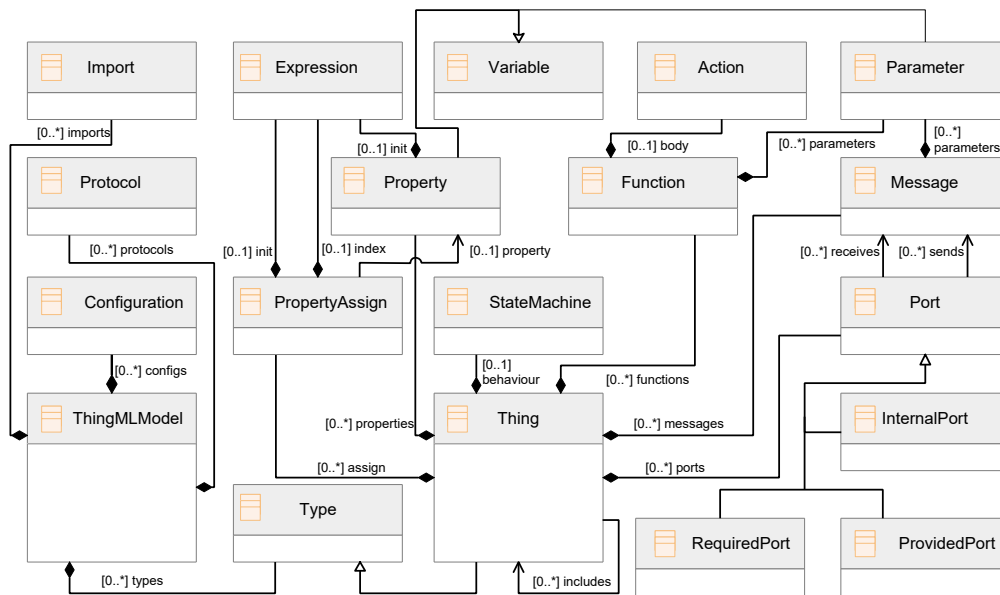


FIGURE 3.1: An excerpt of the ThingML meta-model.

Figure 3.1 represents an excerpt of the ThingML meta-model. It shows the main ThingML model concepts and the relationships between these concepts. In short, as stated earlier, a ThingML model, described by the *ThingMLModel* class in this meta-model, can be expressed as a combination of things and configurations. The *Thing* class includes (composition relation) several components such as messages, functions, properties, state machines, and ports. Each of these components is described by a class in the meta-model.

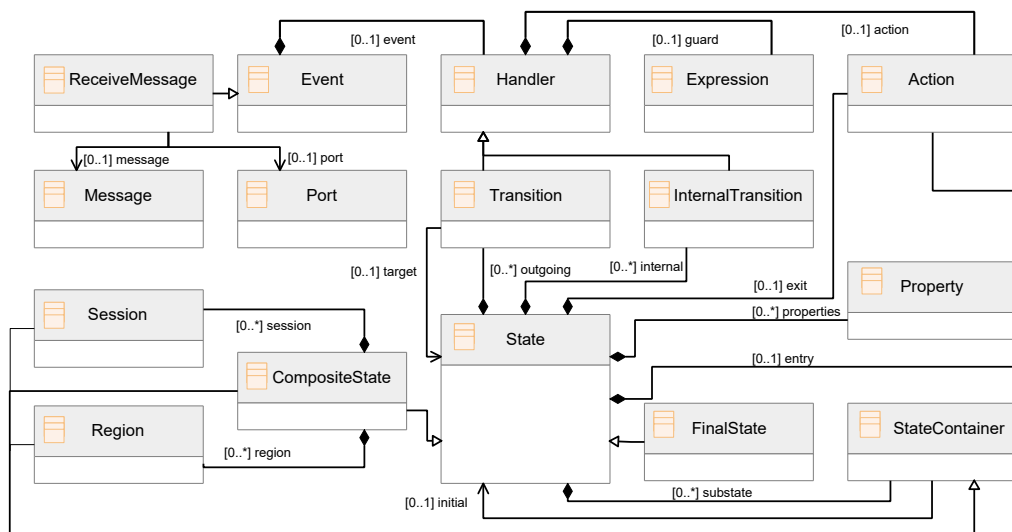


FIGURE 3.2: An excerpt of the ThingML meta-model (the state machine part).

Figure 3.2 shows the state machine part in the ThingML meta-model. The *State* class presents the main class; it contains action on entry, action on exit, properties, and transitions. The states may also contain composite states that can be sequential or concurrent. A transition has a target

state and can contain an event, a guard condition, and an action. An event is a message that arrives via a port.

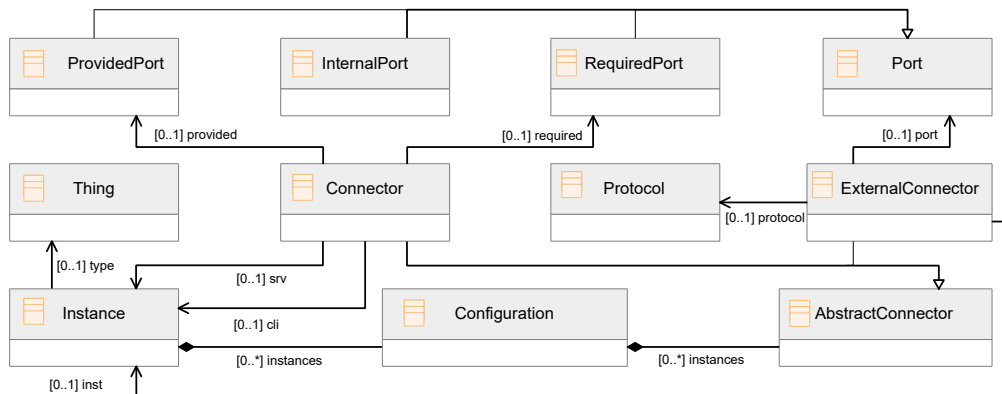


FIGURE 3.3: An excerpt of the ThingML meta-model (the configuration part).

Figure 3.3 presents the configuration part in the ThingML meta-model. A configuration can include instances and connectors. A connector has a client instance (defined by the *cli* relation) and a server instance (defined by the *srv* relation). It also has a provided port (defined by the *provided* relation) and a required port (defined by the *required* relation). The type relation links the two classes, *Instance* and *Thing*.

### 3.2.2 Thing

The main component in ThingML is the *thing* construct. A thing, called process or component in other approaches, is mainly a software component but can represent a software wrapper of a hardware component, for example a light-emitting diode, a piezoelectric buzzer, an algorithm, or an entire program. The things are entirely modular, for example if a thing LED is created, this LED can be reused for all LEDs used in the application, circuit, and even other applications. In other words, the generic behavior of a LED is defined once and becomes available to be used in all applications. A Thing can include *messages*, *ports*, *functions*, *properties*, and a *state machine*. The internal behavior of the thing is defined as a state machine within the thing component. Communication between things can only be made by asynchronous messages carried via ports. Therefore, messages can only be sent and received through ports. These messages may contain parameters of any data type supported by ThingML. Finally, the properties define local variables that are only accessible by their thing's state machines and functions.

If a thing has one or more ports that it provides, these ports can be defined as a fragmented thing or defined in the thing itself. A fragment defines a thing that cannot be instantiated but is included in other things. The fragment is created by the keywords “*thing fragment*” and are followed by a name that ends with “*Msgs*” (see Listing 3.2). Inside the fragment, all messages that the port must handle must be defined. With all messages defined, this fragment can now be used as an interface for things; this means that several different things can include this fragment.

```
1 // Thing fragment definition (this is a comment)
2 thing fragment LedMsgs {
3     // Definition of messages
4     message led_ON();
5     message led_OFF();
6 }
```

LISTING 3.2: Specification of a thing fragment and messages

The communication between things in ThingML happens through the use of ports. A port can send and receive messages to and from other ports. A thing can provide a port, which becomes available for other things to use. Ports can also be required, meaning a thing can state that it uses another port. The communication through ports is done with asynchronous message passing. The messages sent through the port are the main way of triggering transitions and internal events, making the state machine change states and make the program go on. Listing 3.3 (Lines 10-12 and 18-20) presents an example of a required port and a provided port. These ports are taken from the blink program provided in ThingML [99].

```
7 // Thing definition
8 thing Blink includes LEDMsgs {
9     // Required port definition
10    required port led {
11        sends led_ON, led_OFF
12    }
13    ...
14 }
15 // Thing definition
16 thing LED includes LEDMsgs {
17    // Provided port definition
18    provided port ctrl {
19        receives led_ON, led_OFF
20    }
21    ...
22 }
```

LISTING 3.3: Declaration of ports in ThingML



The ThingML language specifies the dynamic behavior of application components by a mix of state machines, a platform-independent action language, and target languages [2]. Where ThingML provides a set of annotations that enable designers to use the target languages and benefit from existing libraries.

### 3.2.3 The platform-independent action language

ThingML describes the arithmetic and Boolean expressions, sending and receiving messages, declaring local variables or functions, and calling functions with a platform-independent action language. Note that the action language supports structured programming by *if (-else)* conditional action and by *while* and *for* iterative actions (loops). Table 3.1 presents a simplified excerpt of the syntax of this action language described in the Backus-Naur Form (BNF) (see [98] for the complete grammar).

TABLE 3.1: The syntax of platform-independent action language in BNF.

Exp	::= Byte   Char   String   Var   AExp   BExp
AExp	::= Int   Float   AVar   - AExp   AExp bin_op AExp
BExp	::= Bool   BVar   BExp bool_op BExp   <b>not</b> BExp   AExp rel_op AExp
bin_op	::= +   -   *   /   %
rel_op	::= ==   !=   <=   <   >=   >
bool_op	::= <b>and</b>   <b>or</b>
Para	::= ID : DataType
Action	::= <b>do</b> Action <b>end</b>   Action Action   <b>var</b> Para = Exp   Var = Exp   AVar = AExp   BVar = BExp   ID ! ID [ ( parameters ) ]   AVar ++   AVar --   <b>while</b> ( BExp ) Action   <b>for</b> ( Para [, Para ] <b>in</b> Array ) Action   <b>if</b> ( BExp ) <b>then</b> Action [ <b>else</b> Action ]   <b>return</b> Exp   <b>print</b> Exp   <b>println</b> Exp   <b>error</b> Exp   <b>errorln</b> Exp   FunctionCallStatement

### 3.2.4 State machine

Conforming to UML statecharts, the state machine reacts according to events corresponding to incoming messages on the ports and the local properties' values. Their structure can include *states* (*atomic* or *composite*), *transitions*, and *parallel regions*. The state machine can run actions or call functions in three ways: entering the states, exiting the states, or during the transitions. The transitions are the only way of changing the state machine from one state to another. They fire

when a message arrives via ports, and their guard conditions are evaluated to be true. Parallel regions are used to describe the orthogonal state mechanism (a.k.a. concurrent states). An orthogonal state is a composite state containing several concurrent substates called regions. Each region represents an execution flow.

```
7 // Thing behavior declaration
8 statechart Blink init ON {
9     state OFF {
10        on entry do
11            led!led_OFF()
12            timer!timer_start(0, 1000)
13        end
14        transition -> ON event e : timer?timer_timeout
15    }
16    state ON {
17        on entry do
18            led!led_ON()
19            timer!timer_start(0, 800)
20        end
21        transition -> OFF event e : timer?timer_timeout
22    }
23 }
```

LISTING 3.4: Definition of a state machine in the ThingML language

### 3.2.5 Configuration

A configuration consists of instances and connectors describing a concrete application. The instance inherits all parent Thing characteristics, such as messages, ports, properties, and behavior. The connector represents a link between two ports, a *required port* on the first end and a *provided port* on the second end, where the asynchronous messages are routed between these two ends.

```
7 // Configuration definition
8 configuration BlinkApp {
9     // Declaration of instances
10    // blink is an instance of the "Blink" thing
11    instance blink : Blink
12    // led is an instance of the "LED" thing
13    instance led : LED
14    // Connector declaration
15    connector blink.led => led.ctrl // a required port => a provided port
16 }
```

LISTING 3.5: Definition of a configuration in the ThingML language

### 3.3 Code generation framework

One of the selling points of Model-Driven Software Engineering (MDSE) is the increased productivity offered by automatically generating code from models [2]. The ThingML approach includes a modeling language and tool designed for supporting code generation and a multi-platform code generation framework. It focuses on the customizability of its code generators while providing the abstraction developers need to improve productivity [100]. The approach does not aim at replacing programming or hiding source code but instead at helping developers produce better source code more efficiently. ThingML is implemented in an open-source tool providing a family of code generators targeting heterogeneous platforms. Where its code generation framework has been used to generate code in 3 different languages (C/C++, Java, and Javascript), targeting around ten different target platforms (ranging from tiny 8bit microcontrollers to servers) and ten different communication protocols (see Table 3.2). The ThingML code generation framework also provides a plugin mechanism that can support a wide range of communication protocols such as UART, I2C, MQTT, Websocket, ROS, and others. It has been evaluated through several case studies and it is used to develop a commercial ambient assisted living system [2]. The ThingML approach is currently being used by the Norwegian company Tellu [101] for the development of a new range of eHealth and fall detection systems called Safe@Home [102] to be deployed in elderly homes.

TABLE 3.2: Platforms supported by the code generation framework [2].

Platform	Memory	Type	Target language
Avr 8bits <sup>2</sup>	2-8 KB	Micro Controller	C/C++
TI MSP430	8 KB	Micro Controller	C/C++
ARM Cortex PSoC 4	32KB	Embedded Processor	C/C++
Espruino (ARM)	48KB	Embedded Processor	javascript(JS)
MIPS(Atheros AR9331)	64 MB	Embedded Processor	C/C++,JS,java
Raspberry Pi	0.5-1GB	Embedded Processor	C/C++,JS,java
Intel Edison	1 GB	Embedded Processor	C/C++,JS,java
x86	GBs	Processor	C/C++,JS,java
Linux/Windows	GBs	Cloud	C/C++,JS,java

The structure of this framework makes it highly customizable, allowing the developer to

efficiently and easily customize parts of the code generation process according to the developed applications' peculiarities [2, 9]. This modular structure allows for the customization of some extension points while all the others can be reused as-is. Figure 3.4 presents the ten different extension points we have identified. Each extension point is an interface (or abstract class) in the code generation framework with a set of methods responsible for generating the code associated with a given model element.

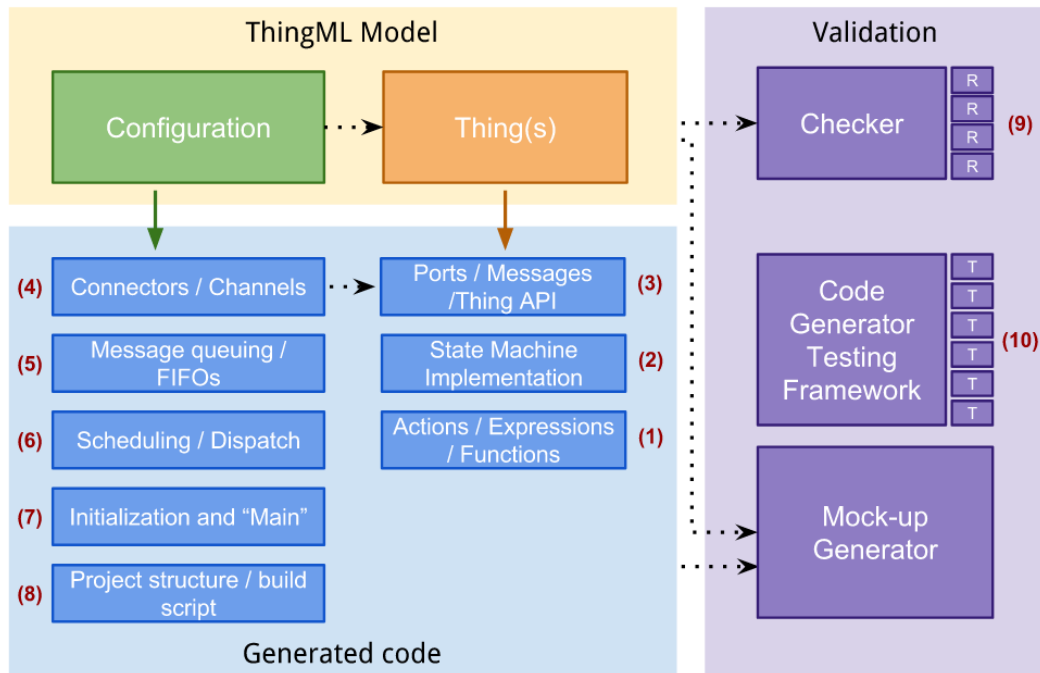


FIGURE 3.4: ThingML framework extension points [2].

### 3.4 Lacks and limits of the ThingML approach

ThingML is a tool-supported model-driven software engineering approach targeting the heterogeneity and distribution challenges associated with developing IoT systems. ThingML is based on a domain-specific modeling language integrating state-of-the-art concepts for modeling IoT systems and comes with a set of compilers targeting a large set of platforms and communication protocols. However, the ThingML approach may have some limitations, which can be summarized as follows:

- (I) ThingML does not have rigorous semantics to support formal reasoning about system designs. Consequently, detecting unwanted behaviors becomes extra complicated, notably

for mission-critical IoT systems where reliability is necessary because failure is potentially catastrophic. To address this limitation, we propose a tool-based approach to transform the ThingML designs into Maude's Rewriting Logic language, which enables rigorous analysis and verification of ThingML designs (see Chapter 5).

- (II) The ThingML DSL provides a textual syntax to describe applications IoT in a platform-independent way. It describes the dynamic behavior of components using a mix of state charts, communication by asynchronous messages, a platform-independent action language, and target languages. Therefore, these specifications can include many details that decrease their legibility. In this context, we develop a hybrid modeling editor for the ThingML language (see Chapter 6). The hybrid editors present the best modeling solutions that combine textual notations with graphical notations and accumulate their advantages. It facilitates the modeling process and helps to clarify and better understand textual models. On the other hand, the ThingML approach lacks tools to test and analyse the generated codes from specifications before deployment on devices. We adopt a simulation approach using Proteus software in the second contribution (see Chapter 6). It enables rapid prototyping of the application hardware circuit and test and evaluates the generated code source on this circuit.

### 3.5 Conclusion

In this chapter, we have presented the ThingML approach. We have focused on its DSL and its code generation framework. We have explained the basic concepts of the ThingML DSL as well as the main characteristics of the code generation framework. Finally, we went over the lacks and limits of the ThingML approach. We have found that ThingML is a promising approach to modeling IoT systems, especially their code generation framework, which generates an operational code for several languages/platforms. However, it lacks explicit execution semantics to execute and simulate the model to analyze and answer questions at design time. In the next chapter, we will describe the basic concepts of Rewriting Logic, its purpose in providing a unifying semantic framework for ThingML models and its power to describe their structural and behavioral aspects.

---

## CHAPTER 4

---

# REWRITING LOGIC AND MAUDE

### 4.1 Introduction

**G**IVEN the increasing complexity of systems along with the safety and proper functioning constraints associated, developing these systems increasingly calls for modeling, verification and validation activities. Modeling makes it possible to separate the different concerns in the development cycle by providing developers with modeling languages to express precisely the necessary information. To ensure the reliability and dependability of the system, especially in the early stages of its design, it is necessary to use analysis techniques to check the models against the expected properties. Many verification methods exist in the literature have proven their effectiveness. These methods aim to discover errors during the system development process. Formal methods are effective techniques to achieve this purpose. Rewriting Logic provides a powerful formal method that can formally represent a wide range of languages and systems. It also provides powerful verification tools, including simulation and model checking.

In the first part of this chapter, we will present the verification methods that can be used within the framework of a system engineering approach based on models. We will focus mainly on formal verification methods. The second part of the chapter will be devoted to the presentation of the Rewriting Logic and its language Maude. In particular, we will detail the implementation of the executable operational semantics in Maude.

## 4.2 Verification techniques

### 4.2.1 Test

We mention testing among the means used to improve the quality of systems and ensure their proper functioning. The aim is to verify that a certain number of scenarios respect the system's specifications to be developed. It can be used to verify different properties and requirements, either functional or non-functional (such as reliability, performance, and security requirements). Testing consists of stimulating the system with test inputs and comparing the obtained behavior with the expected behavior [103]. A test procedure is perfect if it confronts the system with all possible inputs. This completeness is usually not possible. Dijkstra had already commented on this fact in 1976: "the test can only show the presence of bugs (errors), but it can never demonstrate the absence of bugs". Tests do not totally allow the validation of the final system. However, some tools can be used to generate test sets from a specification to improve the coverage and effectiveness of these tests.

### 4.2.2 Simulation

Another way to validate systems is to simulate the dynamic behavior of the system and thus obtain an execution trace. This simulation can be performed in batch from a pre-defined scenario or interactively. This last solution allows the user to build the execution trace from the events he injects progressively. The user can also see his model evolve throughout the execution and thus visually control the system's behaviour for a given execution. Simulation allows for improving the understanding of a system without having to manipulate it, either because it is not yet defined or available, or because it cannot be directly manipulated due to cost, time, resources or risk. The simulation is therefore performed on a model of the system.

Simulation is generally defined in three stages. The first step consists in generating a representation of the workload, i.e. the set of inputs to be applied to the studied system. This representation can be a trace (or a scenario) describing an actual workload or more synthetic and generated by a heuristic or a stochastic function. The second step consists in simulating the model

from the workload defined as the input to produce the results. Finally, the third step consists in analyzing the results of the simulation to gain a better understanding of the considered system.

### 4.2.3 Formal verification techniques

Generally, there are two main families of techniques to formally verify a system's correctness. First, theorem-proving techniques are mathematical proofs in the classical sense of the term, where the verification of properties is done by deduction from a set of axioms and rules of inference. The second family of techniques is called model-checking and decides if a system behavior model satisfies a given property (expressed in temporal logic) by exploring the model state space.

In the theorem-proving technique, the system and the properties sought are expressed as formulas in mathematical logic. This logic is described by a formal system that defines a set of axioms and deduction rules. Theorem proving is the process of finding the proof of a property from the system's axioms. This can be done with the help of an interactive proof assistant, which is designed to help the user construct a formal axiomatic proof. The steps during the proof involve the axioms, rules, the definitions and lemmas that were eventually derived. Theorem proving can be used with infinite state spaces using techniques like structural induction. Its primary disadvantage is that the verification process is usually slow, error-prone, labor-intensive and requires very specialized users with much expertise.

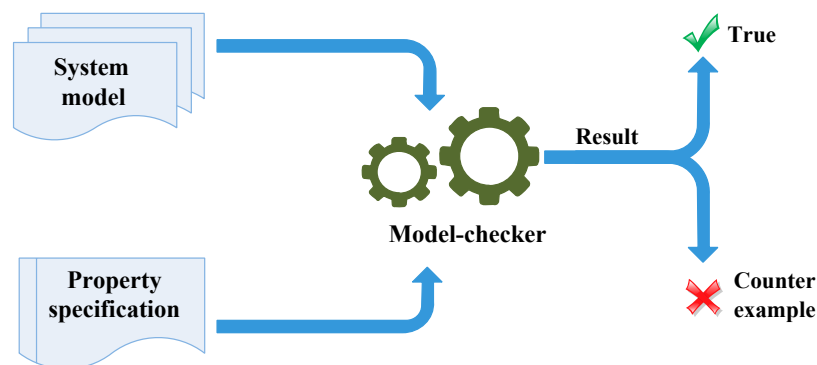


FIGURE 4.1: Model Checking Process.

Model checking is a formal verification technique to determine if a system satisfies a set of properties. Solving the model-checking problem is done using a software tool called *model checker*. Figure 4.1 shows that a model checker typically supports two specification levels, *system* and



*property*. The system specification level, provided by the system model, formalizes the system's behavior to be analyzed. The property specification level in which we specify some property (or properties) that we want to check about the analyzed system. Based on a partial or exhaustive exploration of the model's state space, the model checker either outputs a claim that the property is true or provides a counterexample reporting the inconsistency.

### 4.3 Rewriting Logic

Rewriting Logic was proposed by J. Meseguer [11] as a unified logic that generalizes equational logic and term rewriting for concurrency. It is a general semantic framework in which many (programming or modeling) languages and systems can be naturally specified and analyzed. We present in this section the basic ideas about rewriting logic, the Maude system, and specifying programming language semantics in Maude.

#### 4.3.1 Rewrite theory

**Definition 3.1.** A **signature** of the membership equational logic is defined as a triple  $\Sigma = (K, F, S)$ , where:

- $K$  is a set of Kinds;
- $F$  is a set of  $K^* * K$  function operations, where each symbol  $f \in F_{k_1 \dots k_n}$  is denoted by  $f : k_1 \dots k_n \rightarrow k$ ;
- $S = S_k$  a  $K$ -kinded family of disjoint sets of sorts.

**Definition 3.2.** A membership equational theory is a pair  $(\Sigma, E)$  with:

- $\Sigma$  is a signature of the membership equational logic;
- $E$  is a set of -possibly conditional-  $\Sigma$ -equations ( $t = t'$  if *cond*) and -possibly conditional-  $\Sigma$ -memberships ( $t : s$  if *cond*) for  $t, t' \in T_\Sigma(X)_k$  and  $s \in S_k$ .  $T_\Sigma(X)_k$  denotes the set of  $\Sigma$ -terms with kind over the set  $X$  of kinded variables. *cond* is a condition of the general form  $\wedge_i p_i = q_i \wedge \wedge_j w_j : s_j$ .

**Definition 3.3.** A rewrite theory is defined as a triple  $\mathcal{R} = (\Sigma, E, R)$ , where

- $(\Sigma, E)$  is an equational theory (it can be many-sorted, order-sorted, or a membership equational theory [104]);
- $R$  is a set of labeled -possibly conditional- rewriting rules applied modulo the equations  $E$ .

The equational theory  $(\Sigma, E)$  describes system states as the algebraic data type  $T_{\Sigma/E}$ , and the rewriting rules  $R$  describe the dynamic behavior of concurrent systems. A rewriting rule has the form  $r : t \rightarrow t' \text{ if } Cond$  with  $r$  a label and  $t, t'$  terms. It indicates that the term  $t$  is transformed into  $t'$  if the condition  $Cond$  is satisfied where a term represents the described system's state or partial state. A rule's condition can have a conjunction of rewrites, equations, and memberships, with the general form  $(\wedge_i u_i = u'_i) \wedge (\wedge_j v_j : s_j) \wedge (\wedge_l w_l \rightarrow w'_l)$ .

### 4.3.2 Deduction rules

The rewrite theory is viewed as an executable specification or a prototype of the concurrent system that it formalizes. Computation in a concurrent system is a sequence of transitions (rewrite rules) executed from a given initial state. It corresponds to proof or deduction in the rewrite logic. This deduction is intrinsically concurrent and allows correct reasoning on the system's evolution from one state to another. Given a rewriting theory  $\mathcal{R} = (\Sigma, E, R)$ , we say that the sequence  $[t] \rightarrow [t']$  is provable in  $\mathcal{R}$ , and we write  $\mathcal{R} \vdash [t] \rightarrow [t']$  if and only if  $[t] \rightarrow [t']$  is obtained by finite application of the following deduction rules [11]:

- **Reflexivity.** For each term  $[t] \in T_{\Sigma/E}(X)$ ,  $\frac{}{[t] \rightarrow [t]}$ , where  $T_{\Sigma/E}(X)$  is the set of  $\Sigma$ -terms with variables built on the  $\Sigma$ -signature and  $E$ -equations.
- **Congruence.** For each function  $f \in \Sigma_n, n \in \mathbb{N}$ ,  $\frac{[t_1] \rightarrow [t'_1] \dots [t_n] \rightarrow [t'_n]}{[f(t_1, \dots, t_n)] \rightarrow [f(t'_1, \dots, t'_n)]}$
- **Unconditional replacement.** For each unconditional rule  $r : [t(x_1, \dots, x_n)] \rightarrow [t'(x_1, \dots, x_n)] \in R$ ,  $\frac{[w_1] \rightarrow [w'_1] \dots [w_n] \rightarrow [w'_n]}{[t(\bar{w}/\bar{x})] \rightarrow [t(\bar{w}'/\bar{x})]}$  Knowing that  $t(\bar{w}/\bar{x})$  denotes the simultaneous substitution of  $x_i$  by  $w_i$  in  $t$ , with  $1 \leq i \leq n$ .
- **Replacement.** For each rule  $r : [t(\bar{x})] \rightarrow [t'(\bar{x})] \text{ if } [u_1(\bar{x})] \rightarrow [v_1(\bar{x})] \wedge \dots \wedge [u_k(\bar{x})] \rightarrow [w_1] \rightarrow [w'_1] \dots [w_n] \rightarrow [w'_n]$   
 $[v_k(\bar{x})] \in R$ ,  $\frac{[u_1(\bar{w}/\bar{x})] \rightarrow [v_1(\bar{w}/\bar{x})] \dots [u_k(\bar{w}/\bar{x})] \rightarrow [v_k(\bar{w}/\bar{x})]}{[t(\bar{w}/\bar{x})] \rightarrow [t(\bar{w}'/\bar{x})]}$

➔ **Transitivity.**  $\frac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$

## 4.4 Maude language

Rewriting logic has been implemented in different logical languages. The *Maude* language [15, 105] is a widely used implementation of rewriting logic [106]. It is a simple, expressive and efficient language based on the equational membership logic and rewriting logic; it also supports executable specification and programming [107]. The basic units of specification or programming in Maude are called *modules*. The basic types of modules in Core-Maude are *functional modules* and *system modules* [108]. Functional modules to implement membership equational theories. System modules implement rewriting theories and define the dynamic behavior of a system.

### 4.4.1 Functional module

The implementation of the functional modules is based on equational theories [109]. These modules allow the definition of data types and operators (operations on these data). An equational simplification materializes the rewriting within these modules. Equational simplification is rewriting an initial expression until no equation is applicable. The result is called the *canonical form*, which is the same whatever the order of execution of the equations. The keywords that introduce a functional module are:

```
fmod ModuleId is Module body endfm
```

*ModuleId* is the identifier of the functional module, the module body defines data types and their functions by means of a membership equational theory of the form  $(\sum, M \cup E \cup A)$ , where:

- (i)  $\sum$  is a signature defining the *sorts*, *subsort*, and the *operations* used in the theory. The sorts present the data types; they are declared according to the syntax:

```
sort SortId .
```

We can order the data types by indicating the kinds that have a relationship between them using the keyword *subsort* (or *subsorts*) according to the syntax:

```
subsort Sort1 < Sort2 .
```

This means that *Sort1* is a subsort of *Sort2*. The operations used to declare the constants and function symbols used in the theory. An operation with its arguments (sorts) is declared in the general form:

```
op OpId : Sort1 Sort2 ... SortN -> ResultSort .
```

If an operation's argument set is empty, that operator is named a *constant* of *ResultSort*.

- (ii) *E* is a set of -possibly conditional- equations used as simplification rules to evaluate the terms to their canonical form. Reducing with the *E* equations is performed modulo *A*. Unconditional equations are declared using the *eq* keyword according to the following general schema:

```
eq Term-1 = Term-2 .
```

The terms *Term-1* and *Term-2* must both have the same sort. Conditional equations are declared using the *ceq* keyword according to the following general schema:

```
ceq Term-1 = Term-2 if EqCondition .
```

A condition can be a single equation, membership, or conjunction of equations and memberships.

- (iii) *A* is a set of equational axioms (such as associativity, commutativity, and identity) satisfied by some of the function symbols in  $\Sigma$ .
- (iv) *M* is a collection of -possibly conditional- memberships. Unconditional. Membership axioms specify terms as having a given sort. They are declared with the *mb* and *cmb* keywords according to the following form:

```
mb Term : Sort . --- Unconditional Memberships
cmb Term : Sort if EqCondition . --- Conditional Memberships
```

Variables can be declared in modules using the *var* or *vars* keywords or introduced directly into equations and membership tests in the form of an *X: SortId* expression that declares a variable named *X* of sort *SortId*.

```
1 mod NAT-ADD is
2   sorts Nat NzNat .
3   subsort NzNat < Nat .
4   op 0 : -> Nat [ctor] .
5   op s_ : Nat -> Nat [ctor] .
6   op _+_ : Nat Nat -> Nat .
```

```

7   vars N M : Nat .
8   cmb N : NzNat if N =\= 0 .
9   eq 0 + N = N .
10  eq s N + M = s (N + M) .
11  endm

```

LISTING 4.1: Example of a functional module

Listing 4.1 shows an example of a functional module named *NAT-ADD* introducing two sorts, *Nat* to represent natural numbers and *NzNat* to represent non-zero natural numbers (lines 2, 8). We declared the sort *NzNat* as a sub-sort of *Nat* (in line 3). This module shows an alternative way to define natural numbers using the *s* (successor) operation (line 5). Thus, only one base number exists (the constant "0") (line 4), and the other numbers are defined using the successor operation (line 10).

#### 4.4.2 System module

A system module specifies a rewrite theory of the form  $\mathcal{R} = (\Sigma, M \cup E \cup A, R)$ . It extends the functional module by introducing a set of -possibly conditional- rewriting rules which define the system behaviors. In other words, rewriting rules specify the concurrent local transitions performed in the system. A system module is declared as follows:

```
mod ModuleId is Module body endm
```

Where *ModuleId* is the identifier of the system module, the module body contains the same elements as those of the equational theory except for the rewriting rules.

➔ **Unconditional rewriting rules:** The system's dynamism can be modeled using rewriting rules. Each rule has the following form:

```
r1 [label] : t => t' .
```

➔ **Conditional rewriting rules:** The rewrite rules of this category will be executed if their conditions are evaluated to be true. These rules are declared as follows:

```
crl [label] : t => t' if C .
```

We consider a concurrent vending machine system to buy apples and cakes, where the user can insert dollars and quarters [105]. A cake costs a dollar, and an apple three quarters. When the user buys an apple, the machine takes a dollar and returns a quarter. The machine can change four quarters into a dollar. The system module shown in Listing 4.2, called *VENDING-MACHINE*, presents a specification of the vending machine's behavior.

```

1  mod VENDING-MACHINE is
2  sorts Coin Item Marking .
3  subsorts Coin Item < Marking .
4  op __ : Marking Marking -> Marking [assoc comm id: null] .
5  op null : -> Marking .
6  ops $ q : -> Coin .
7  ops a c : -> Item .
8  var M : Marking .
9  rl [add-q] : M => M q .
10 rl [add-$] : M => M $ .
11 rl [buy-c] : $ => c .
12 rl [buy-a] : $ => a q .
13 rl [change] : q q q q => $ .
14 endm

```

LISTING 4.2: Example of a system module

This specification introduces the  $\$$  and  $q$  constants to represent dollar and quarter coins, respectively (line 6), and the  $a$  and  $c$  constants to represent apples and cakes, respectively (line 7). The operations performed by the machine are specified using the rewriting rules. These are the following operations:

- ➔ Insert a quarter coin in the machine, described by the *add-q* rewriting rule (line 9).
- ➔ Insert a dollar coin in the machine, described by the *add-\$* rewriting rule (line 10).
- ➔ Buy a cake that costs 1 \$, described by the *buy-c* rewriting rule (line 11).
- ➔ Buy an apple that costs three quarters, it described by the *buy-a* rewriting rule (line 12).
- ➔ Change four quarters into a dollar, described by the *change* rewriting rule (line 13).

### 4.4.3 Simulation and analysis in Maude

In Maude, simulating a behavior involves transforming the initial state to another by applying one or more rewriting rules. Therefore, behavior means a sequence of rewriting steps. Maude

offers three main ways to simulate and analyze the modules: rewriting, searching, and LTL model checking [15].

#### 4.4.3.1 Rewriting and search

The *reduce* command (abbreviated as *red*) allows an initial term to be reduced by applying the equations and membership axioms in a given module. While the *rewrite* command (abbreviated as *rew*) and the *frewrite* (fair rewrite) command (abbreviated as *frew*) perform a *single rewrite sequence* from a given initial term. They allow an initial term to be rewritten using the specified module's rules, equations, and membership axioms. Note that no rule will be applied if an equation can be applied. In the case of the *rewrite* command, the default interpreter applies the rewriting rules using a rule-fair top-down strategy and stops when the number of rule applications reaches the given bound. The *frewrite* command rewrites a term using a rule- and position-fair strategy that makes it possible for some rules to be applied that could be “starved” using the leftmost, outermost rule fair strategy of the *rewrite* command.

Unlike the *rewrite* and *frewrite* commands, which explore only one possible behavior (sequence of rewrites), the *search* command allows analysis of all possible sequences of rewrites from an initial state (term). It searches if states corresponding to given patterns and satisfying certain conditions can be accessed from the initial term. The execution of this command performs a deep traversal of the computational tree (reachability tree) generated during this search to detect invariant violations in systems with infinite states [15].

#### 4.4.3.2 The Maude's LTL model-checker

The Maude system provides an efficient model checker, a powerful formal method to verify the specification properties [16]. In Maude's LTL model checking, the system is defined using Maude modules, and the properties to be checked are described using Linear Time Logic (LTL) [16]. We first define a set of Atomic Propositions (AP) to formulate properties in LTL. Then, we use the logical operators (the traditional operators of propositional calculus and temporal operators) to define the LTL formulas inductively as follows [105]:

- $\top \in \text{LTL formula.}$

- if  $p \in \text{AP}$  then  $p \in \text{LTL formula}$ .
- if  $\psi$  and  $\varphi \in \text{LTL formulas}$  then  $\neg\psi$ ,  $\varphi \vee \psi$ ,  $o\psi$ , and  $\varphi \mathcal{U} \psi \in \text{LTL formulas}$ .

Where  $\mathcal{U}$  is the until operator, and  $o$  is the next operator. In addition to these fundamental operators, other logical and temporal operators can be defined in terms of these connectives. Following are the additional temporal operators [105].

- *Eventually*:  $\diamond \varphi = \top \mathcal{U} \varphi$
- *Henceforth*:  $\Box \varphi = \neg \diamond \neg \varphi$
- *Release*:  $\varphi \mathcal{R} \psi = \neg((\neg \varphi) \mathcal{U} (\neg \psi))$
- *Unless*:  $\varphi \mathcal{W} \psi = (\varphi \mathcal{U} \psi) \vee (\Box \varphi)$
- *Leads-to*:  $\varphi \rightsquigarrow \psi = \Box(\varphi \rightarrow (\diamond \psi))$
- *Strong implication*:  $\varphi \Rightarrow \psi = \Box(\varphi \rightarrow \psi)$
- *Strong equivalence*:  $\varphi \Leftrightarrow \psi = \Box(\varphi \leftrightarrow \psi)$

## 4.5 Executable operational semantics in Maude

The semantics of a programming language focuses on giving meaning to programs. It removes any ambiguity in the definition of a programming language. There are different formal methods to define these semantics. Operational semantics [110] give meaning to programs in terms of computational steps. More precisely, it is interested in how states are changed when executing instructions. Operational semantics inductively defines a program's evaluation relation (inference rule). This relation describes a system of transitions between different states of the programs. Depending on the nature of transitions, there are two main approaches to operational semantics [111]:

- *Structural operational semantics (small-step semantics)* [110]: also called computation semantics [112]. In this approach, the inference rules describe all the elementary computation steps of a program's execution.
- *Big-step semantics (or natural semantics)* [113]: in this approach, the inference rules link a program to its final result without specifying the computation steps that led to this result. Sometimes it is called the evaluation semantics [112].



For more details about the operational semantics, we direct the reader to M. Hennessy's book [112] which provides a clear introduction to the subject.

### 4.5.1 Syntax definition

We begin our description of how to implement operational semantics in Maude with a language of arithmetic and Boolean expressions (Exp4) [112]. Exp4 is a language with arithmetic and Boolean expressions, if-then-else, and local variable declarations (let). Figure 4.2 shows the abstract syntax of this language with obvious intuitive meaning.

#### 1. Syntactic categories

$e$	$\in$	$Exp$	$be$	$\in$	$BExp$
$op$	$\in$	$Op$	$bop$	$\in$	$BOp$
$n$	$\in$	$Num$	$x$	$\in$	$Var$
$bx$	$\in$	$BVar$			

#### 2. Definitions

$op$	$::=$	$+ \mid - \mid *$
$bop$	$::=$	$And \mid Or$
$e$	$::=$	$n \mid x \mid e' op e'' \mid let\ x = e' in\ e'' \mid If\ be\ Then\ e' Else\ e''$
$be$	$::=$	$bx \mid T \mid F \mid be' bop be'' \mid Not\ be' \mid Equal(e, e')$

FIGURE 4.2: Abstract syntax for Exp4

This syntax is implemented in the following functional module EXP4-SYNTAX. Note that the signature structure corresponds to the grammar structure defined by the language's syntax in Figure 4.2.

```

1  fmod EXP4-SYNTAX is
2    pr QID .
3    sorts Exp BExp Num Boolean Var BVar Op BOp .
4    subsorts Num Var < Exp .
5    subsorts Boolean BVar < BExp .
6
7    op V : Qid -> Var .
8    ops + - * : -> Op .
9    op 0 : -> Num .
10   op s : Num -> Num .
11
12   op ___ : Exp Op Exp -> Exp [prec 20] .
13   op If_Then_Else_ : BExp Exp Exp -> Exp [prec 25] .
14   op let_=in_ : Var Exp Exp -> Exp [prec 25] .
15
16   op BV : Qid -> BVar .

```

```

17   ops And Or : -> BOp .
18   ops T F : -> Boolean .
19
20   op ___ : BExp BOp BExp -> BExp [prec 20] .
21   op Equal : Exp Exp -> BExp .
22   op Not_ : BExp -> BExp [prec 15] .
23   endfm

```

We use the predefined quoted identifiers, of sort *Qid*, for representing variable identifiers in the language *Exp4*. Instead of declaring this sort as a subsort of *Var*, since *Qid* is also used to represent Boolean variables, we have constructors *V* and *BV* that transform the *Qids* to values of sorts *Var* and *BVar*, respectively. We use the natural numbers in Peano notation as arithmetic constants, with constructors *0* and *s*.

```

1   fmod AP is
2     pr EXP4-SYNTAX .
3
4     op Ap : Op Num Num -> Num .
5     vars n n' : Num .
6     eq Ap(+, 0, n) = n .
7     eq Ap(+, s(n), n') = s(Ap(+, n, n')) .
8     eq Ap(*, 0, n) = 0 .
9     eq Ap(*, s(n), n') = Ap(+, n', Ap(*, n, n')) .
10    eq Ap(-, 0, n) = 0 .
11    eq Ap(-, s(n), 0) = s(n) .
12    eq Ap(-, s(n), s(n')) = Ap(-, n, n') .
13
14    op Ap : BOp Boolean Boolean -> Boolean .
15    var bv : Boolean .
16    eq Ap(And, T, bv) = bv .
17    eq Ap(And, F, bv) = F .
18    eq Ap(Or, T, bv) = T .
19    eq Ap(Or, F, bv) = bv .
20  endfm

```

In another functional module, *AP*, we define an operation *Ap* for applying a binary operator to two already evaluated arguments. A third functional module *ENV* is used to define environments that associate values to variables, either arithmetic or Boolean.

```

1   fmod ENV is
2     pr EXP4-SYNTAX .
3
4     sorts Value Variable .
5     subsorts Num Boolean < Value .
6     subsorts Var BVar < Variable .
7     sort ENV .

```

```

8
9   op mt : -> ENV .
10  op _= : Variable Value -> ENV [prec 20] .
11  op __ : ENV ENV -> ENV [assoc id: mt prec 30] .
12  op _'(_') : ENV Variable -> Value .
13  op _'[_/_' ] : ENV Value Variable -> ENV [prec 35] .
14  op remove : ENV Variable -> ENV .
15
16  vars X X' : Variable .
17  var V : Value .
18  var ro : ENV .
19  eq (X = V ro)(X') = if X == X' then V else ro(X') fi .
20  eq ro [V / X] = remove(ro, X) X = V .
21  eq remove(mt, X) = mt .
22  eq remove(X = V ro, X') = if X == X' then ro else X = V remove(ro,X') fi .
23  endfm

```

Operations  $mt$ ,  $\_ = \_$ , and  $\_ \_$  (in the module  $ENV$ ) are used to build empty environments, singleton environments, and union (with overriding) of environments, respectively. The operation  $\_ (\_)$  is used to look up the value associated with a variable in an environment and is defined recursively by an equation. The operation  $\_ [\_ / \_]$  is used to modify the binding between a variable and a value in an environment, and it is defined by adding to the left a new binding to the environment. The last equation removes repetitions.

### 4.5.2 Big-step semantics

The Big-step semantics for Exp4 is given using two relations:  $\Longrightarrow_A$  and  $\Longrightarrow_B$ , corresponding respectively to arithmetic and Boolean expressions. The judgments in this semantics will have the form:  $\langle \rho \vdash e \Longrightarrow_A v \rangle$  and  $\langle \rho \vdash be \Longrightarrow_B bv \rangle$ . Where  $\rho$  is an environment that stores the value of each variable,  $e$  (resp.  $be$ ) is an arithmetic expression (resp. boolean expression) of the language, and  $v$  (resp.  $bv$ ) is the value at which the expression  $e$  (resp.  $be$ ) evaluates. The semantics rules of this language are shown in Figures 4.3 and 4.4.

In this type of semantics, it is usual that rules such as the rule  $OpR$ . This rule expresses that to evaluate the expression  $e \text{ op } e'$  in the environment  $\rho$ , one must evaluate both  $e$  and  $e'$ . To obtain their values  $v$  and  $v'$ , respectively, the total result being the application (with the function  $Ap$ ) of the binary operator  $op$  to the results  $v$  and  $v'$ .

Rule CR	$\rho \vdash n \Longrightarrow_A n$
Rule VarR	$\rho \vdash x \Longrightarrow_A \rho(x)$
Rule IfR	$\frac{\rho \vdash be \Longrightarrow_B T \quad \rho \vdash e \Longrightarrow_A v}{\rho \vdash \text{If } be \text{ Then } e \text{ Else } e' \Longrightarrow_A v} \quad \frac{\rho \vdash be \Longrightarrow_B F \quad \rho \vdash e' \Longrightarrow_A v'}{\rho \vdash \text{If } be \text{ Then } e \text{ Else } e' \Longrightarrow_A v'}$
Rule OpR	$\frac{\rho \vdash e \Longrightarrow_A v \quad \rho \vdash e' \Longrightarrow_A v'}{\rho \vdash e \text{ op } e' \Longrightarrow_A \text{Ap}(\text{op}, v, v')}$
Rule LocR	$\frac{\rho \vdash e \Longrightarrow_A v \quad \rho[v/x] \vdash e' \Longrightarrow_A v'}{\rho \vdash \text{let } x = e \text{ in } e' \Longrightarrow_A v'}$

FIGURE 4.3: Evaluation semantics for arithmetic expressions:  $\Longrightarrow_A$ 

Rule BCR	$\rho \vdash T \Longrightarrow_B T$	$\rho \vdash F \Longrightarrow_B F$
Rule BVarR	$\rho \vdash bx \Longrightarrow_B \rho(bx)$	
Rule EqR	$\frac{\rho \vdash e \Longrightarrow_A v \quad \rho \vdash e' \Longrightarrow_A v}{\rho \vdash \text{Equal}(e, e') \Longrightarrow_B T}$	$\frac{\rho \vdash e \Longrightarrow_A v \quad \rho \vdash e' \Longrightarrow_A v'}{\rho \vdash \text{Equal}(e, e') \Longrightarrow_B F}$ <p style="text-align: center;"><i>if</i> <math>v \neq v'</math></p>
Rule BOpR	$\frac{\rho \vdash be \Longrightarrow_B bv \quad \rho \vdash be' \Longrightarrow_B bv'}{\rho \vdash be \text{ bop } be' \Longrightarrow_B \text{Ap}(\text{bop}, bv, bv')}$	
Rule NotR	$\frac{\rho \vdash be \Longrightarrow_B T}{\rho \vdash \text{Not } be \Longrightarrow_B F}$	$\frac{\rho \vdash be \Longrightarrow_B F}{\rho \vdash \text{Not } be \Longrightarrow_B T}$

FIGURE 4.4: Evaluation semantics for Boolean expressions:  $\Longrightarrow_B$ 

The EVALUATION module has the rewrite rules representing the evaluation semantics for Exp4, both for arithmetic and Boolean expressions. A semantics rule of the form  $\frac{P_1 \rightarrow Q_1 \dots P_n \rightarrow Q_n}{P_0 \rightarrow Q_0}$  is transformed into a conditional rewriting rule of the form  $P_0 \rightarrow Q_0 \text{ if } P_1 \rightarrow Q_1 / \dots / P_n \rightarrow Q_n$ , where the conclusion becomes the main rewriting rule, and the premises become the rule condition that includes rewrites [107]. First, the elements on both sides of the arrow in a judgment must be represented as terms in Maude to represent the semantic rules in Maude. In this semantics, we have an environment and expression on the left. A term of the sort *Statement* represents these two elements. On the right, we can have arithmetic or Boolean expression. Notice the use of the sort *Statement* is to ensure that both sides of the rewrite rules will have a common sort.

```

sort Statement .
subsorts Num Boolean < Statement .

op _|-_ : ENV Exp -> Statement [prec 40] .
op _|-_ : ENV BExp -> Statement [prec 40] .

```

The axioms (semantics rules without premises) are translated as (unconditional) rewrite rules, where the transition, in conclusion, becomes the rewrite rule.

```

var ro : ENV .      var n      : Num      .      var x      : Var      .      var bx     : BVar .
var op  : Op   .      vars e e'  : Exp     .      vars be be' : BExp .      var v v'  : Num   .
var bop : BOp  .      var bv bv' : Boolean .

rl [CR]      : ro |- n  => n .
rl [VarR]    : ro |- x  => ro(x) .

rl [BCR1]   : ro |- T  => T .
rl [BCR2]   : ro |- F  => F .
rl [BVarR]  : ro |- bx => ro(bx) .

```

The rest of the semantic rules (with premises) are translated to conditional rewrite rules where the main rewrite corresponds to the transition in conclusion, and the rewrites in the conditions correspond to the transitions in the premises. Conditions are ordered to be checked sequentially from left to right, and therefore information can flow from one condition to the next; this happens in the rule LocR below, where the value of  $v$  is obtained in the first condition and is later used in the second.

```

cr1 [OpR]   : ro |- e op e' => Ap(op,v,v')   if ro |- e => v /\      ro |- e' => v' .
cr1 [IfR1]  : ro |- If be Then e Else e' => v if ro |- be => T /\      ro |- e => v .
cr1 [IfR2]  : ro |- If be Then e Else e' => v' if ro |- be => F /\      ro |- e' => v' .
cr1 [LocR]  : ro |- let x = e in e' => v'     if ro |- e => v /\      ro[v / x] |- e' => v' .

```

The EVALUATION module is admissible and directly executable in Maude. For example, The following command evaluates the product of two numbers  $s(s(s(0)))$  and  $s(s(s(s(0))))$  in the Maude implementation presented above.

```

Maude> rew mt |- s(s(s(0))) * s(s(s(s(0)))) .
rewrite in EVALUATION : mt |- s(s(s(0))) * s(s(s(s(0)))) .
rewrites: 22 in 0ms cpu (0ms real) (~ rewrites/second)
result Num: s(s(s(s(s(s(s(s(s(s(0))))))))))

```

## 4.6 Conclusion

In this chapter, after introducing an overview of verification methods, we have seen some basic concepts of Rewriting Logic, constituting a semantics framework for specifying languages and concurrent systems. We have also presented the Maude language, its different modules, and its main ways of simulation and analysis that can be performed. Finally, we have presented an implementation of executable an semantics in Maude. The concepts presented in this chapter constitute a necessary background for understanding our contribution in this thesis's context.

---

## CHAPTER 5

---

### MDE-BASED FORMAL APPROACH

#### 5.1 Introduction

THINGML is a promising approach to modeling IoT systems. However, it does not have rigorous semantics for formal reasoning about system designs. Consequently, detecting unwanted behaviors becomes extra difficult, notably for mission-critical IoT systems where reliability is a requisite need because failure is potentially catastrophic. In this chapter, we will present an MDE-based formal approach to define and implement formal semantics of ThingML language using rewriting logic and its Maude language. In this sense, ThingML and Maude language have complementary characteristics which can be applied jointly. Using ThingML allows designers to model their IoT applications and to benefit from a set of code generators for various platforms, whereas Maude allows these designs to be analyzed and verified.

#### 5.2 General overview

Figure 5.1 gives a general overview of the proposed MDE-based formal approach [17]. The designers model the system's functionality according to the ThingML meta-model. After that, they transform obtained specifications into Maude models using the proposed transformation rules implemented on the Acceleo framework [19]. Finally, the resulting Maude modules will be used by the designers to verify system properties expressed in LTL logic. The model checker returns a *true* when the specification is found to meet these properties. In this case, designers run the code generation for the desired platform. Otherwise, the model checker provides a *counterexample* that can be used to make corrections.

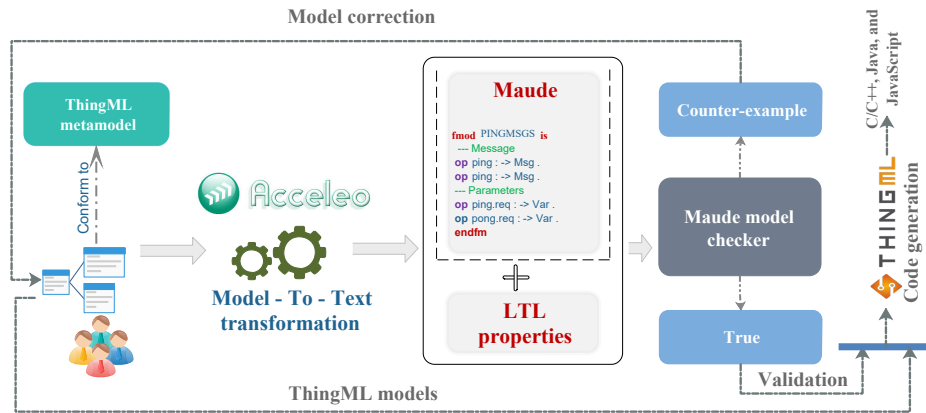


FIGURE 5.1: The workflow of the proposed MDE-based formal approach.

### 5.3 Formalization of ThingML constructs

This section will explain how to express a ThingML program in Maude. By this formalization, we intend to allow the analysis of ThingML specifications through the analysis results obtained from the equivalent Maude specifications. Our formalization uses the structures declared in predefined module *CONFIGURATION* that permit the modeling of object-based systems. This predefined module provides basic sorts and constructors to represent the essential concepts of object, message, and configuration. The objects are described as record-like structures of the form  $\langle O : C \mid att_1 : v_1, \dots, att_n : v_n \rangle$ , where  $O$  is an object identifier,  $C$  is a class identifier,  $att_i$  are identifiers of attributes, and  $v_i$  are the current values of these attributes. The configuration has the structure of a multi-set of objects and messages that evolves by concurrent rewriting [105].

In our formalization, ThingML instances are translated into *Maude objects*, including their execution environment. The execution environment is implemented in Maude using an *attribute* that includes two parts: *Store* and *Action*. The environment attribute allows objects to run their actions and manage their information. The evaluation semantics for the ThingML action language is given through big-step semantics in Maude language to evaluate the expressions and execute the actions. Additionally, a rewriting rules-based semantics is defined to run the state machine, which can change the object status based on events (or messages) that have arrived via ports. Likewise, a semantics of message routing between objects through connectors is defined using Maude rewriting rules. Connectors are translated into objects with two attributes, *client* and *server*—knowing that the client and server are objects having the *buffer* attribute that temporarily



stores the sent messages. The main ideas of our formalization can be summarized in Table 5.1 [17]. The details of our formalization will be presented in the following subsections.

TABLE 5.1: Summary of the correspondences between the main ThingML and Maude constructs.

Construct	ThingML specification	Corresponding Maude code
<i>Thing</i>	<code>thing fragment F-T {</code> ... <code>}</code>	<code>mod F-T is</code> ... <code>endm</code>
	<code>thing T includes F-T {</code> ... <code>}</code>	<code>mod T is</code> <code>pr F-T .</code> <code>op T : -&gt; ThingId [ctor] .</code> --- The class identifier ... <code>endm</code>
<i>Messages</i>	<code>message M() ;</code>	<code>op M : -&gt; MsgId [ctor] .</code>
	<code>message M(par:DataType) ;</code>	<code>op M : -&gt; MsgId [ctor] .</code> <code>op par : -&gt; Var [ctor] .</code> <code>eq parmsg(M) = par .</code>
<i>Ports</i>	<code>provided port P { ... }</code>	<code>op P : -&gt; PortId [ctor] .</code>
	<code>required port P { ... }</code>	<code>op P : -&gt; PortId [ctor] .</code>
	<code>internal port P { ... }</code>	<code>op P : -&gt; PortId [ctor] .</code>
<i>properties</i>	<code>property Pr : DataType = 0</code>	<code>op Pr : -&gt; Var [ctor] .</code>
	<code>readonly property Pr :</code> <code>    DataType = 0</code>	<code>op Pr : -&gt; Var [ctor] .</code>
<i>Platform-independent language</i>	- Data types (Char, String, Boolean, UInt8, UInt16, integer, Float ... )	- Maude's predefined sorts (Bool, Int, Nat, Float, String)
	- The Arithmetic, Boolean and relational operations	- New operations corresponding to Maude's predefined operations (with the same properties) - An evaluation semantics of expression language implemented in Maude
	- Actions and Functions	- Operations - An evaluation semantics implemented in Maude
<i>statechart</i>	<code>statechart SC init S0 {</code> <code>state S0 {</code> <code>on entry Act-ent-S0</code> <code>on exit Act-exi-S0</code> <code>transition Tran -&gt; S1</code> <code>event P ? M</code> <code>guard Cond</code> <code>action Act-T</code> } <code>}</code>	<code>op S0 : -&gt; AtomicStateId [ctor] .</code> ---( These actions will be used in the rewriting rules corresponding to the transitions) <code>cr1 [Tran] :</code> <code>&lt; I : T   environment: &lt; noAction,</code> <code>(status: S0) ; st ; (P ? M) ; st' &gt;&gt;</code> <code>=&gt;</code> <code>&lt; I : T   environment: &lt;Act-exi-S0 ; Act-T ;</code> <code>goto (S1) ; Act-ent-S1 , (status: noState) ; st ;</code> <code>st' &gt;&gt; if Cond .</code>
	<code>state S1 {...}</code>	<code>op S1 : -&gt; AtomicStateId [ctor] .</code>
	<code>composite state CS {...}</code>	<code>op CS : -&gt; CompositeStateId [ctor] .</code>

<i>Configuration</i>	<b>configuration</b> Config	<b>mod</b> Config <b>is</b>
	{	<b>op</b> Config : -> Configuration .
	<b>instance</b> I1 : T1 <b>instance</b> I2 : T2	<b>pr</b> T1 . <b>pr</b> T2 . <b>op</b> I1 : -> InstanceId [ctor] . --- An object identifier <b>op</b> I2 : -> InstanceId [ctor] . <b>eq</b> Config = < I1 : T1   environment: < Act1, (status: S0-1);(Pr = 0);...>> < I2 : T2   environment: < Act2 , (status: S0-2);(Pr = 0);... >>
<b>connector</b> I1.P1 ==> I2.P2	<b>connector</b>   client: < T1 . P1 : RequiredPort   buffer: noMsg > --> server: < I2 . P2 : ProvidedPort   buffer: noMsg > .	
}	<b>endm</b>	

### 5.3.1 Thing

We propose to transform a *Thing* component specification into a system module in which we declare: the class identifier corresponding to Thing, messages, ports, properties, states, and the rewriting rules defining the behavior of this *Thing*. To declare the Thing class's objects with the *environment* attribute, we introduce the following structure [17]:

```

sorts ThingId Statement Store Action .
subsort ThingId < Cid . --- Cid : Class identifier
--- Declaration of the environment attribute
op environment:_ : Statement -> Attribute [ctor gather (&)] .
op <_,_> : Action Store -> Statement [ctor] .

```

The *environment* attribute represents the key structure to implement the evaluation semantics of the ThingML action and expression language. It acts like a memory consisting of two parts:

- *Action*: includes a sequence of actions to be executed. The proposed evaluation semantics allows giving meaning to these actions (i.e., execute these actions in the desired order). Details of these semantics will be found in Section 5.3.3.
- *Store*: stores the status, sent messages, events, and different variables' values. It serves as a storage memory allowing the reading, writing and updating of the information it contains. Therefore, the content of this memory is modified according to the actions that will be executed.

In [107], the authors have propose implementing the *Store* concept in Maude. However, their implementation is limited to arithmetic and Boolean variables. To take into consideration all Thing information, we extend Store’s functionality to enable storing and managing the thing’s status, messages, properties, parameters, and events. The *THINGML-STORE* module represents the implementation of these functionalities. It includes operators and equations that ensure different functions such as reading, writing, and modifying variables (arithmetic, Boolean, and string), status, messages, and events.

```

--- `(_=_)` : use to associate a value to their variable
op `(_=_)`      : Variable Value -> Store [prec 20] . --- SS of a variable
op `(_)`       : Event          -> Store [prec 20] . --- SS of a event
op `(_Via_)`   : Msg PortId     -> Store [prec 20] . --- SS of a message
op `(status:_)` : Status        -> Store [prec 20] . --- SS of the status
op `;_`       : Store Store -> Store [assoc id: mt prec 30] . --- Union of SSs

--- The next operators are used to add or update the SEs of the:
op `[_/_]` : Store Value Variable -> Store [prec 35] . --- variable
op `[_/_]` : Store VaList VarList -> Store [prec 35] . --- var list
op `[_|_]` : Store Msg PortId     -> Store [prec 35] . --- message
op `[_]`   : Store Status         -> Store [prec 35] . --- status
--- `(_)` : returns a value (or a list) of the variable (or a list)
op `(_)` : Store Variable -> Value .
op `(_)` : Store VarList -> VaList .
--- The next part of the module represents the implementation of the operations defined above

```

### 5.3.2 Messages and ports

We map messages, parameters, and properties to operators. We declare the parameters and the properties as operators of the *Var* sort (variable). For the messages, we propose the following Maude code where these messages can include parameters [17].

```

sorts MsgId MsgSet .
op `(_)` : MsgId -> Msg [ctor] .
op `(_)` : MsgId ExpList -> Msg [ctor] .
subsort Msg < MsgSet .
op noMsg : -> MsgSet [ctor] .
op `;_` : MsgSet MsgSet -> MsgSet [ctor comm assoc id: noMsg] .
op parmsg : MsgId -> VarList .

```

Ports are endpoints for channels transporting asynchronous messages between thing instances. We propose the following structure to describe the ThingML ports as port class objects having

the *buffer* attribute that temporarily stores the sent messages through their port. As appears in this code, there are three classes of ports: *ProvidedPort*, *RequiredPort*, and *InternalPort*.

```

sorts Port PortId PortName .
subsort Port < Cid .    --- Cid : Class identifier
subsort PortName < Oid .  --- Oid : Object identifier
op _._ : InstanceId PortId -> PortName [ctor] .
--- Declaration of port classes
ops ProvidedPort RequiredPort InternalPort : -> Port .
--- class ProvidedPort | buffer: MsgSet .
--- class RequiredPort | buffer: MsgSet .
--- class InternalPort | buffer: MsgSet .
--- Declaration of the buffer attribute
op buffer:_ : MsgSet -> Attribute [ctor gather (&)] .

```

The operators defined in the *THINGML-STORE* module allow saving a sent message as a singleton store (*m Via port*) in the emitting object's environment. The implementation of message routing will be discussed in Section 5.3.5 after demonstrating the connector concept.

### 5.3.3 Platform-independent language

This section shows the syntax and the semantics to execute the ThingML action language in Maude. The first step is to define the formal semantics of the action and expression language. For this purpose, we use the notation of evaluation semantics presented in [18]. Then, we implement these semantics in the Maude language based on the work presented in [107]. We describe only a few examples illustrating the principle operations (See Appendices A and B for more details).

#### 5.3.3.1 Expressions

We define the evaluation semantics of language expressions in terms of the relations:  $\Rightarrow_A$ ,  $\Rightarrow_B$ , and  $\Rightarrow_C$  corresponding respectively to the arithmetic, Boolean, and relational expressions. An evaluation relation is given by the notation  $\Rightarrow : \langle e, st \rangle \mapsto v$ , where it takes a pair consisting of an expression and a memory and returns a value, the result of evaluating the expression in this memory [18]. Figure 5.2 [17] describes the Boolean expressions' evaluation rules ( $\Rightarrow_B$ ). *Ap* (*bop*, *bv*, *bv'*) denotes the Boolean operator's application (denoted *bop*) to both Boolean values *bv* and *bv'*.

$$\text{Value\_R} : \frac{}{\langle bv, st \rangle \Rightarrow_B bv} \quad (1)$$

$$\text{BVar\_R} : \frac{}{\langle bx, st \rangle \Rightarrow_B st(bx)} \quad (2)$$

$$\text{BOp\_R} : \frac{\begin{array}{l} \langle be, st \rangle \Rightarrow_B bv \\ \langle be', st \rangle \Rightarrow_B bv' \end{array}}{\langle be \text{ bop } be', st \rangle \Rightarrow_B \text{Ap}(bop, bv, bv')} \quad (3)$$

FIGURE 5.2: Evaluation semantics for Boolean expression.

In Maude, to evaluate the expressions, we first map the data types of the ThingML language to Maude's predefined sorts. Then, we define new arithmetic and logical operations corresponding to Maude's predefined operations, where we keep the same properties [17].

```

--- "And" and "Or" operations corresponding to the "and" and "or" operations of the BOOL module
ops And Or   : -> Bop .
op _And_     : BExp BExp -> BExp [ctor assoc comm prec 55] .
op _Or_      : BExp BExp -> BExp [ctor assoc comm prec 59] .

```

Next, we implement the evaluation rules using the rewriting logic as follows:

```

op <_,_> : BExp Store -> Statement . --- corresponding to rule =>B : < e , st > -> v
--- To make sure that both parties of a rewriting rule take the same sort.
subsorts Bool Int Nat Float String Store < Statement .
vars be be' : BExp . var bx : BVar . vars bv bv' : Bool .
--- The rewriting rule corresponding to the evaluation rule (1)
rl [Value-R] : < bv, st > => bv .
--- The rewriting rule corresponding to the evaluation rule (2)
--- st(_) : is previously defined in the THINGML-STORE module above.
rl [BVar-R] : < bx, st > => st(bx) .
--- The rewriting rules implementing the evaluation rule (3)
cr1 [BOp-R-And] : < be And be', st > => Ap(And,bv,bv') if < be, st > => bv /\ < be', st > => bv' .
cr1 [BOp-R-Or] : < be Or be', st > => Ap(Or,bv,bv') if < be, st > => bv /\ < be', st > => bv' .

```

Finally, we use the *Ap* operation that enables to apply a binary operator to two already evaluated arguments [107]. It allows switching between the defined operations and the corresponding Maude's predefined operations. The latter enables the execution of the arithmetic and logical operations concretely.

```

op Ap : BOp Bool Bool -> Bool .
eq Ap(And,bv,bv') = bv and bv' .
eq Ap(Or,bv,bv') = bv or bv' .

```

Similarly, we describe and implement the other arithmetic, logical and relational operations.

### 5.3.3.2 Action language

We give the action evaluation semantics in terms of the relation  $\Rightarrow_D$  described by the notation:  $\Rightarrow_D : \langle A, st \rangle \mapsto st'$ , where it takes a pair consisting of the  $A$  action and the  $st$  memory and returns the  $st'$  modified memory. Figure 5.3 shows the evaluation rules for three actions  $goto()$ ,  $!_!$  (send action), and conditional action.

$$Goto\_R : \frac{}{\langle goto(s), st \rangle \Rightarrow_D st(s)} \quad (4)$$

$$Send\_R : \frac{}{\langle p!msg(), st \rangle \Rightarrow_D st[msg()|-p]} \quad \frac{}{\langle p!msg(vl), st \rangle \Rightarrow_D st[msg(vl)|-p]} \\ \frac{\langle xl, st \rangle \Rightarrow_D vl}{\langle p!msg(xl), st \rangle \Rightarrow_D st[msg(vl)|-p]} \quad (5)$$

$$If\_R : \frac{\langle be, st \rangle \Rightarrow_D true \quad \langle A, st \rangle \Rightarrow_D st'}{\langle If\ be\ Then\ A\ Else\ A', st \rangle \Rightarrow_D st'} \quad \frac{\langle be, st \rangle \Rightarrow_D false \quad \langle A', st \rangle \Rightarrow_D st'}{\langle If\ be\ Then\ A\ Else\ A', st \rangle \Rightarrow_D st'} \quad (6)$$

FIGURE 5.3: Evaluation semantics for some ThingML actions

After defining the semantics, all that remains is implementing it in the Maude language. For that, we first translate the ThingML actions into Maude operators according to the following structure [17]:

```

sort Action .
op noAction   :                -> Action .
op _:=_      : Var Exp         -> Action .
op _;_       : Action Action -> Action [assoc] .
op !_!_     : PortId Msg      -> Action .
op _++      : Var             -> Action .
op _--      : Var             -> Action .
op If_Then_ : BExp Action     -> Action .
op print`(_`) : SString      -> Action .
op If_Then_Else_ : BExp Action Action -> Action .
op While_Do_  : BExp Action     -> Action .
op goto`(_`) : Status         -> Action .

```

Then, we transform the semantics evaluation rules into rewriting rules enabling the actions placed in the environment attribute to be executed in the desired order.

```

var be : BExp . vars st st' : Store . vars A A' : Action .
var msg : MsgId . var p : PortId . var xl : VarList .
var vl : ValList . var s : Status .
--- The rewriting rule corresponding to the evaluation rule (4)

```

```

--- st[_] : is previously defined in the THINGML-STORE module above
rl [GoTo-R] : < goto(s) , st > => st[s] .
--- The rewriting rules corresponding to the evaluation rule (5)
rl [Send-R3] : < p ! msg() , st > => st [ msg() |- p ] .
rl [Send-R2] : < p ! msg(v1) , st > => st [ msg(v1) |- p ] .
cr1 [Send-R1] : < p ! msg(x1) , st > => st [ msg(v1) |- p ] if < x1 , st > => v1 .
--- The rewriting rules implementing to the evaluation rule (6)
cr1 [If-R1] : < If be Then A Else A' , st > => st' if < be , st > => true /\ < A , st > => st' .
cr1 [If-R2] : < If be Then A Else A' , st > => st' if < be , st > => false /\ < A' , st > => st' .

```

Action  $goto(\_)$  is not included in the action language of ThingML. However, we have proposed to define it to implement state transition. Performing this action changes the object status to the specified state in the parameter [17]. We explain the reason for adding  $goto(\_)$  action in the following subsection.

### 5.3.4 State machine

The state machine comprises a set of states (i.e., atomic, composite, or regions) and transitions to change the system state according to events that arrived via ports. First, to represent the different types of states, we propose the following structure in Maude [17]:

```

sorts StateId CompositeStateId AtomicStateId .
subsorts CompositeStateId AtomicStateId < StateId .
sort Status .
subsort AtomicStateId < Status .
op _`(_` ) : CompositeStateId Status -> Status [ctor] .
op noState : -> Status [ctor] .
op _||_ : Status Status -> Status [ctor assoc id: noState] .

```

As a reminder, an object status is specified by a singleton store ( $status: \_$ ) in the environment attribute of that object. In addition, an event is described as a singleton store in the receiving object's environment. For this purpose, we declare the *Event* sort and  $\_? \_$  operator to describe the events.

```

sort Event .
op _?_ : PortId MsgId -> Event [ctor] .

```

Finally, we transform the transitions into rewriting rules. To illustrate the mapping process, we present the transformation of a typical transaction. The following description in ThingML shows a  $T$  transition links two states,  $S$  and  $R$ . The  $S$  state has an action block executed on exit,

and the  $R$  state has an action block executed on entry. The  $T$  transition represents a change in the system state from  $S$  to  $R$ , triggered upon the arrival of a  $Msg$  message via the  $P$  port if the  $C$  condition is verified.

```

1  state S {
2      on exit do S-Exit-Acts end
3      transition T -> R event P?Msg
4          guard C
5          action do T-Acts end
6  }
7  state R {
8      on entry do R-Entry-Acts end
9  }
```

The transformation produces the following rewriting rule:

```

crl [S-To-R] : < I : Thing | environment: < noAction , (status: S) ; st ; (P ? Msg) ; st' > >
=> < I : Thing | environment: < S-Exit-Acts ; T-Acts ; goto(R) ; R-Entry-Acts , (
status: noState) ; st ; st' > >
if < C , st ; st' > => true .
```

Therefore, when the  $I$  instance changes its status from  $S$  to  $R$ , it first goes to the *noState* state (i.e., the instance is changing its status). We introduce the *noState* state to freeze the instance status on this transitional state until the end of the execution of the exit actions of the  $S$  state (*S-Exit-Acts*) and the  $T$  transition actions (*T-Acts*). When the execution of these actions is finished, the instance goes to the  $R$  state using the *goto(R)* action. Finally, the instance executes the entry actions of the  $R$  state (*R-Entry-Acts*) [17]. The rule condition is a reachability relation of the form  $\langle be, st \rangle \Rightarrow true$ , where  $be$  is a Boolean expression, and  $st$  is the instance environment, which will be evaluated using the semantics of expressions presented in Section 5.3.3.1.

### 5.3.5 Configuration

We transform a ThingML configuration into a system module in which we declare a Maude *configuration*. The latter consists of objects corresponding to ThingML instances and connectors declared in the ThingML configuration. We map ThingML instances into objects of the corresponding class, and we transform the connectors according to the following proposed structure [17]:



```

sort InstanceId .
subsort InstanceId < Oid . --- Oid : Object identifier
sort Connector .
subsort Connector < Object .
op connector | client:_--> server:_ : Object Object -> Connector [ctor] .

```

This structure defines the connector as an object with two attributes, *client* and *server*, objects of *RequiredPort* and *ProvidedPort* classes respectively. It describes the connection between two ThingML instances as a channel for carrying messages. Figure 5.4 shows the message routing steps from the sender instance to the receiver instance. When sending a message, the emitting object (corresponding to the emitting ThingML instances) stores this message with the port used as a singleton store (*msg Via port*) in its *environment* attribute. Then, the message is moved to the *connector* using the following rewriting rules [17]:

```

--- To move the messages from instance environment to the connector (client buffer)
rl [Env2CliBuff] : < I : T | environment: < A , st ; (M Via Po) ; st' > > connector | client:
    < I . Po : RequiredPort | buffer: MS > --> server: 0
=> < I : T | environment: < A , st ; st' > > connector | client:
    < I . Po : RequiredPort | buffer: (MS ; M) > --> server: 0 .

--- To move the messages from instance environment to the connector (server buffer)
rl [Env2SerBuff] : < I : T | environment: < A , st ; (M Via Po) ; st' > > connector | client: 0
    --> server: < I . Po : ProvidedPort | buffer: MS >
=> < I : T | environment: < A , st ; st' > > connector | client: 0
    --> server: < I . Po : ProvidedPort | buffer: (MS ; M) > .

```

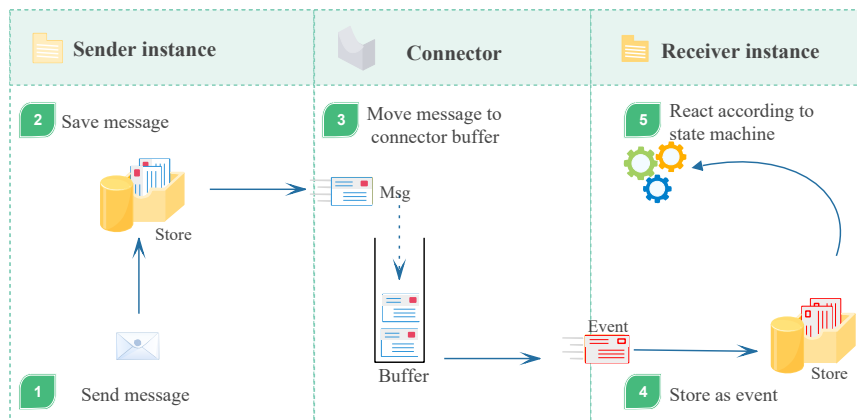


FIGURE 5.4: The message routing

After that, the message travels by the *connector* to be eventually stored in the receiving object environment as an *event*. The following rules enable this progression, where two cases can be

distinguished. In the first case, the sent message does not include any parameters. Consequently, only the event ( $Po ? MsgId$ ) will be stored in the object environment.

```

--- To move the messages from connector ( server buffer) to instance environment (message
    without parameters)
rl [BuffSer2EnvR1] : < I : T | environment: < A , st > > connector | client: < I . Po :
    RequiredPort | ATTS > --> server: < P : ProvidedPort | buffer: ((msgId ()) ; MS) >
=> < I : T | environment: < A , st ; ( Po ? msgId ) > > connector | client: < I . Po :
    RequiredPort | ATTS > --> server: < P : ProvidedPort | buffer: MS > .

--- To move the messages from connector ( client buffer) to instance environment (message
    without parameters)
rl [BuffCli2EnvR1] : < I : T | environment: < A , st > > connector | client: < P : RequiredPort
    | buffer: ((msgId ()) ; MS) > --> server: < I . Po : ProvidedPort | ATTS >
=> < I : T | environment: < A , st ; ( Po ? msgId ) > > connector | client: < P : RequiredPort |
    buffer: MS > --> server: < I . Po : ProvidedPort | ATTS > .

```

In the second case, the sent message includes parameters. Therefore, all events ( $Po? MsgId$ ) and the list of parameters ( $st [vl / (parmsg (msgId))]$ ) will be stored in the object environment. The parameters are saved as variables to be used later in processing.

```

--- To move the messages from connector ( server buffer) to instance environment (message with
    parameters)
rl [BuffSer2EnvR2] : < I : T | environment: < A , st > > connector | client: < I . Po :
    RequiredPort | ATTS > --> server: < P : ProvidedPort | buffer: ((msgId (vl)) ; MS) >
=> < I : T | environment: < A , (st [ vl / (parmsg(msgId)) ]) ; ( Po ? msgId ) > > connector |
    client: < I . Po : RequiredPort | ATTS > --> server: < P : ProvidedPort | buffer: MS > .

--- To move the messages from connector ( client buffer) to instance environment (message with
    parameters)
rl [BuffCli2EnvR2] : < I : T | environment: < A , st > > connector | client: < P : RequiredPort
    | buffer: ((msgId (vl)) ; MS) > --> server: < I . Po : ProvidedPort | ATTS >
=> < I : T | environment: < A , (st [ vl / (parmsg(msgId)) ]) ; ( Po ? msgId ) > > connector |
    client: < P : RequiredPort | buffer: MS > --> server: < I . Po : ProvidedPort | ATTS > .

```

## 5.4 ThingML2Maude: A translator tool of ThingML models to Maude

This section presents the *ThingML2Maude* tool [17], a model-to-text translator to map the ThingML models into Maude code. ThingML2Maude is a code generator based on the Aceleo framework that automatically generates Maude specifications from ThingML models. It is implemented on the Eclipse modeling framework on which the Xtext editor of ThingML works. Once the designer have finished designing IoT systems using ThingML's Xtext-based editor, it opens

the ThingML2Maude Acceleo project to translate their models into Maude code. Thereafter, it sets the execution parameters like input and output files. The input files include the ThingML designs, and the output files are the files where the resulting Maude code will be generated. The last step is to start the run of the ThingML2Maude tools with a simple click to generate the Maude specifications (see Figure 5.5 [17]).

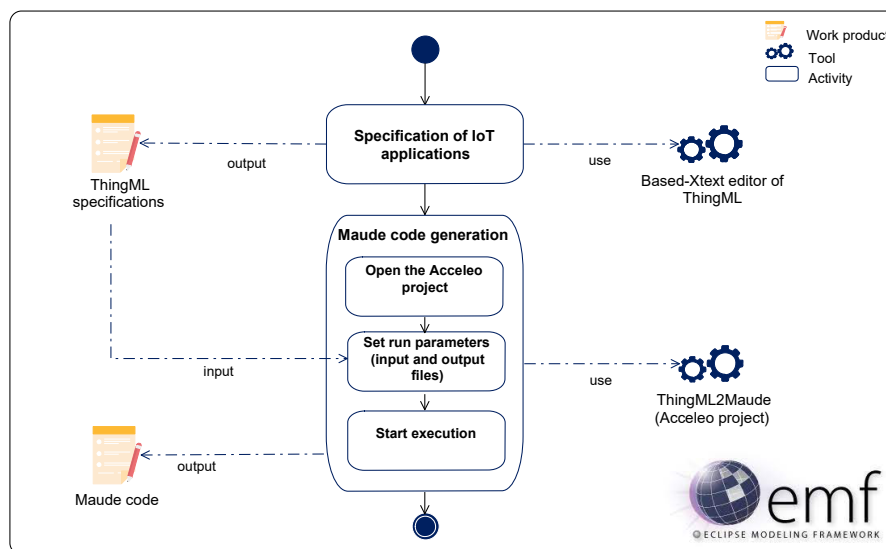


FIGURE 5.5: Automatic transformation process.

Acceleo [19] is a template-based technology that allows the implementation of transformation rules. Each rule transforms an element from the meta-model to the corresponding code. The Acceleo projects require a single source meta-model to generate text from models. We propose around thirty templates that implement dozens of transformation rules to browse the models conforming to the ThingML meta-model and generate the corresponding Maude code. For example, Listing 5.1 [17] describes the *genConfiguration* template that takes as parameters a configuration. This template transforms the instances and the connectors into corresponding Maude code where three other templates, namely *InitState()*, *InitAction()*, and *InitProperty()*, are called to initialize status, actions, and properties, respectively.

Listing 5.2 [17] describes another template that transforms all messages of a thing into the corresponding Maude code. This template takes as parameters a Thing and recalls another template named *ParaMsgVar* used to transform the parameters of messages.

```

9  [template public genConfiguration(aConfiguration : Configuration)]
10 mod [aConfiguration.name.toUpper()] is
11 [if (aConfiguration.instances -> size()) > 0 ]
12 [for (aInstance : Instance | aConfiguration.instances )]
13   pr [aInstance.type.name.toUpper()] .
14 [//for]
15 [//if]
16 [if (aConfiguration.instances -> size()) > 0 ]
17 [for (aInstance : Instance | aConfiguration.instances )]
18   op [aInstance.name/] : -> ThingId .
19 [//for]
20 [//if]
21 op [aConfiguration.name/] : -> Configuration .
22 eq [aConfiguration.name/] =
23 [if (aConfiguration.instances -> size()) > 0 ]
24 [for (aInstance : Instance | aConfiguration.instances )]
25   < [aInstance.name/] : [aInstance.type.name/] |
26     environment: < [aInstance.type.InitProperty()] /]
27     [aInstance.type.behaviour.InitAction()] /] , (status:
28     [aInstance.type.behaviour.InitState()] /] > >
29 [//for]
30 [//if]
31 [if (aConfiguration.connectors -> size()) > 0 ]
32 [for (aConnector : Connector | aConfiguration.eAllContents(Connector))]
33   connector |
34     client: < [aConnector.cli.name/] . [aConnector.required.name/] :
35             RequiredPort | buffer: noMsg >
36   -->
37     server: < [aConnector.srv.name/] . [aConnector.provided.name/] :
38             ProvidedPort | buffer: noMsg >
39 [//for]
40 [//if] .
41 endm
42 [//template]

```

LISTING 5.1: Example of Acceleo Template: the genConfiguration template.

```

17 [template public message(aThing : Thing) post(trim())]
18 --- Messages
19 [for (msg : Message | aThing.messages )]
20   op [msg.name/] : -> MsgId [ctor] .
21 [//for]
22
23 --- Parameters of messages
24 [for (msg : Message | aThing.messages )]
25   [msg.ParamMsgVar()] /]
26 [//for]
27 [//template]

```

LISTING 5.2: Example of Acceleo Template: the message template.

## 5.5 Case study

To illustrate the proposed approach's practical usefulness, we consider a ThingML design extracted from the HEADS research project (Heterogeneous and Distributed Services for the Future Computing Continuum) [114, 115]. The *PingPong* design [116] presents a fully platform-independent ThingML program (only uses ThingML statements). It shows the primary constructs of the ThingML language. This design demonstrates how to use two components to exchange asynchronous messages. The behavior of these two components is described by state machines that

react according to arrived events. These events correspond to incoming messages that are sent by the other component. Therefore, the problem of ensuring consistency between the state machines of the components may arise. For this reason, the PingPong design is a good and simple example to explain our approach for checking the consistency and correctness of ThingML specifications [17].

### 5.5.1 Specification

The *PingPong* design includes four components: the *PingServer* and *PingClient* things, *PingMsgs* as a thing fragment, and the *PingConfig* configuration. In this model, the *PingClient* component sends a *ping* parameterized message to the *PingServer* component that responds with a *pong* message once receiving the *ping* message. The *PingClient* status is changed based on the value of local properties and the *pong* message parameter. Finally, the *PingConfig* configuration represents a concrete application composed of two instances, *client* and *server*, and one connector.

Listing 5.3 describes the *PingMsgs* thing fragment with *ping* and *pong* parameterized messages. The *ping* and *pong* messages are used in implementing the behavior of *PingClient* and *PingServer* things. They include a parameter with the *UInt8* data type of ThingML. This parameter describes the number of *ping* or *pong* messages sent.

```
1  thing fragment PingMsgs
2  {
3      message ping(req : UInt8);
4      message pong(req' : UInt8);
5  }
```

LISTING 5.3: ThingML implementation of PingMsgs thing fragment

Listing 5.4 shows the ThingML specification of the *PingServer* thing, including the *PingMsgs* thing fragment, a provided port, and a statechart defining their behavior. The *pingservice* port enables the *PingServer* thing to exchange messages with other things. It allows sending the *pong* message and receiving the *ping* message. The *PingServerMachine* statechart that implements the *PingServer* thing behavior includes two states *Waiting* and *Pong*. It initially enters the *Waiting* state. It passes from the *Waiting* state to the *Pong* state when a *ping* message arrives through the *pingservice* port. Upon entering the *Pong* state, *PingServer* sends a *pong* message via the

*ping* service port, and then passes again to the *Waiting* state. Figure 5.6 presents graphically the statechart of the *PingServer* thing [17].

```

6  thing PingServer includes PingMsgs
7  {
8      provided port pingservice
9      {
10         sends pong
11         receives ping
12     }
13     statechart PingServerMachine init Waiting
14     {
15         property count : UInt8 = 0
16         state Waiting
17         {
18             transition -> Pong
19                 event m : pingservice?ping // ? : Receive
20                 //the ping message on pingservice port
21                 action count = m.req
22         }
23         state Pong
24         {
25             on entry pingservice!pong(count)
26             // ! : Send message pong on pingservice port.
27             on exit print "Send Pong", count, "..."
28             transition -> Waiting
29         }
30     }
31 }

```

LISTING 5.4: ThingML implementation of PingServer thing.

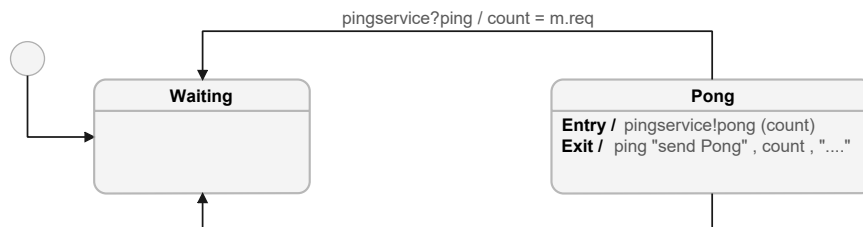


FIGURE 5.6: The statechart of the PingServer thing.

Listing 5.5 presents the ThingML specification of the *PingClient* thing, including the *PingMsgs* thing fragment, two properties, a required port, and a statechart defining their behavior.

```

32  thing PingClient includes PingMsgs {
33      readonly property count_max : UInt8 = 5
34      property counter: UInt8 = 1

```

```

35     required port pingservice
36     {
37         receives pong
38         sends ping
39     }
40     statechart PingClientMachine init Ping
41     {
42         state Ping
43         {
44             on entry do
45                 print "Send Ping", counter, "..."
46                 pingservice!ping(counter)
47             end
48             transition -> Waiting
49         }
50         state Waiting
51         {
52             transition -> Ping
53                 event e : pingservice?pong
54                 guard e.req' == counter and counter < count_max
55                 action do println "[OK]"
56                     counter = counter + 1
57                 end
58             transition -> Stop
59                 event e : pingservice?pong
60                 guard e.req' != counter
61             transition -> OK
62                 event e : pingservice?pong
63                 guard e.req' == counter and counter >= count_max
64         }
65         final state OK
66         {
67             on entry
68                 println "[OK] Bye."
69         }
70         final state Stop
71         {
72             on entry
73                 println "[Error]"
74         }
75     }
76 }

```

LISTING 5.5: ThingML implementation of PingClient thing.

In this specification, we declare a required port that enables the *PingClient* thing to exchange messages with other things. This port allows sending the *ping* message and receiving the *pong* message. As shown in the specification, the types of sent messages (resp. received messages) via the required port correspond to the types of received messages (resp. sent messages) through the provided port. The *PingClientMachine* statechart that implements the *PingClient* thing behavior

includes four states (*Waiting*, *Ping*, *OK*, and *Stop*), where *OK* and *Stop* are the final states. The *PingClient* thing initially enters the *Ping* state. Upon entering the *Ping* state, *PingClient* sends a *ping* message via the *pingservice* port. Then passes to the *Waiting* state, which has three transitions. These transitions fire when a *pong* message arrives through the *pingservice* port, and their guard conditions are satisfied. Therefore, the *PingClient* status passes from the *Waiting* state to the *Ping*, *OK*, or *Stop* state, following the guard conditions. Figure 5.7 presents graphically the statechart of the *PingClient* thing [17].

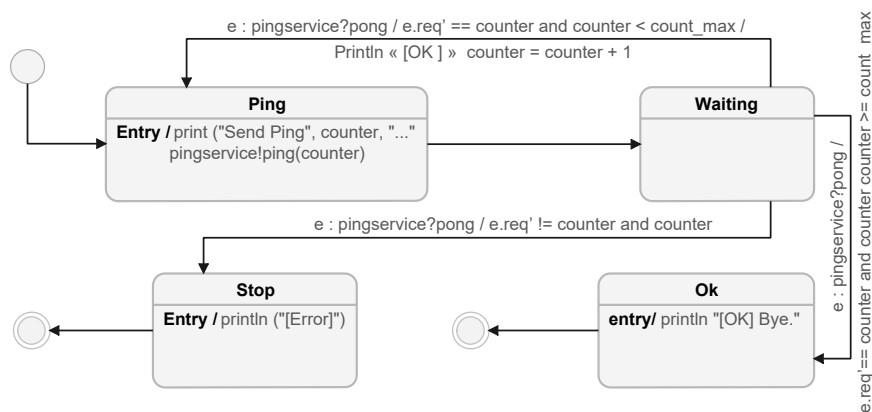


FIGURE 5.7: The statechart of the *PingClient* thing.

Listing 5.6 describes the ThingML implementation of the *PingConfig* concrete application. The *PingConfig* configuration comprises two instances and a connector. The *client* is an instance of the *PingClient* thing, and the *server* is an instance of the *PingServer* thing. The connector links these instances via their ports, where the required port of the client instance is at the first end, and the provided port of the server instance is at the second end.

```

77 configuration PingConfig
78 {
79     instance client: PingClient
80     instance server: PingServer
81     connector client.pingservice => server.pingservice
82 }

```

LISTING 5.6: ThingML implementation of *PingConfig* configuration.



### 5.5.2 Transformation

To verify and analyze this design, we transform it into the corresponding Maude code. We use the *ThingML2Maude* tool to realize this transformation. The following four modules present the Maude specification resulting from the automatic transformation. First, the *PINGMSGSGS* system module [17] represents the specification of the *PingMsgs* thing fragment in Maude. This module contains the *pong* and *ping* operators of the *MsgId* sort (message identifiers) corresponding to the messages declared in the *PingMsgs* thing fragment. It also contains the *req* and *req'* operators of the *Var* sort (variable) corresponding to the message parameters. Finally, it includes two equations that determine the parameters of the *pong* and *ping* messages.

```

1  mod PINGMSGSGS is
2    pr THINGML-MSG-SEMANTIC .
3    --- Messages
4    ops pong ping : -> MsgId [ctor] .
5    --- Parameters of messages
6    ops req req' : -> Var [ctor] .
7    eq parmsg(ping) = req .
8    eq parmsg(pong) = req' .
9  endm

```

LISTING 5.7: System module corresponding to PingMsgs thing fragment.

Second, two other system modules describe *PingServer* and *PingClient* things in the Maude language. They are named *PINGSERVER* and *PINGCLIENT* [17].

```

1  mod PINGSERVER is
2    pr PINGMSGSGS .
3    --- Operetors
4    op PingServer : -> ThingId [ctor] .
5    op count : -> Var [ctor] .
6    ops Waiting Pong : -> AtomicStateId [ctor] .
7    op pingservice : -> PortId [ctor] .
8    --- Variables
9    vars st st' : Store .   vars A A' : Action .
10   var I : InstanceId .   var VL : VaList .
11   --- Rewriting rules
12   rl [Waiting-To-Pong] : < I : PingServer | environment: < noAction , ( status: Waiting )
13     ; st ; ( pingservice ? ping ) ; st' > > =>
14     < I : PingServer | environment: < count := req ; goto(Pong) ; pingservice ! pong (
15     count)) , (status: noState) ; st ; st' > > .
16
17   rl [Pong-To-Waiting] : < I : PingServer | environment: < noAction , ( status: Pong ) ;
18     st > > =>

```

```

16     < I : PingServer | environment: < print( "Send Pong count..." ) ; goto( Waiting ) , (
      status: noState ) ; st > > .
17   endm

```

LISTING 5.8: System module corresponding to PingServer thing.

In these modules, we declare the *PingServer* and *PingClient* things as classes having the *environment* attribute containing information about the current status, variable values, actions, and messages. The classes are declared using operators of the *ThingId* sort, where the *ThingId* sort is a subsort of the *Cid* sort (Class identifier). The *PINGSERVER* and *PINGCLIENT* modules also declare the states, variables, and ports of things as operators of the sorts *AtomicStateId*, *Var*, and *PortId*, respectively. Finally, they include the rewriting rules (possibly conditional) corresponding to transitions of the thing statechart.

```

1  mod PINGCLIENT is
2    pr PINGMSGS .
3    --- Operators
4    op PingClient : -> ThingId [ctor] .
5    ops count-max counter : -> Var [ctor] .
6    ops Ping Waiting OK Stop : -> AtomicStateId [ctor] .
7    op pingservice : -> PortId [ctor] .
8    --- Variables
9    vars st st' : Store .   vars A A' : Action .
10   var I : InstanceId .   var VL : VaList .
11   --- Rewriting rules
12   rl [Ping-To-Waiting] : < I : PingClient | environment: < noAction , (status: Ping) ; st
      > > => < I : PingClient | environment: < goto(Waiting) , (status: noState) ; st >
      > .
13
14   crl [Waiting-To-Ping] : < I : PingClient | environment: < noAction , (status: Waiting)
      ; st ; (pingservice ? pong) ; st' > > => < I : PingClient | environment: < print("[
      OK]") ; counter := counter .+ 1 ; goto(Ping) ; print("Send Ping counter...") ; (
      pingservice ! ping(counter)) , ( status: noState ) ; st ; st' > >
15   if < ((req' .= counter) And (counter .< count-max)) , st ; st' > => true .
16
17   crl [Waiting-To-Stop] : < I : PingClient | environment: < noAction , (status: Waiting)
      ; st ; (pingservice ? pong) ; st' > > => < I : PingClient | environment: < goto(
      Stop) ; print("[Error]") , ( status: noState ) ; st ; st' > >
18   if < req' .!= counter , st ; st' > => true .
19
20   crl [Waiting-To-OK] : < I : PingClient | environment: < noAction , (status: Waiting) ;
      st ; (pingservice ? pong) ; st' > > => < I : PingClient | environment: < goto(OK
      ) ; print("Bye.") , ( status: noState ) ; st ; st' > >
21   if < ((req' .= counter) And (counter .>= count-max)) , st ; st' > => true .
22   endm

```

LISTING 5.9: System module corresponding to PingClient thing.

Finally, the *PINGCONFIG* module [17] describes the configuration specification. In this module, the transformation products two operators (*client* and *server*) of the *InstanceId* sort and an operator *PingConfig* of the *Configuration* sort. The *PingConfig* configuration contains two objects and a connector. We initialize the *environment* attribute of objects according to the parent thing's specification, where the initialization includes the initial state, entry actions, and variable values. The connector is described as an object with two attributes, *client* and *server*, objects of *RequiredPort* and *ProvidedPort* classes, respectively. We initialize the *buffer* attribute of the ports by *noMsg* value (no message in buffer).

```

1  mod PINGCONFIG is
2    pr PINGCLIENT .
3    pr PINGSERVER .
4
5    ops client server : -> InstanceId [ctor] .
6    op PingConfig : -> Configuration .
7    eq PingConfig = < client : PingClient | environment: < count-max := 5 ; counter := 1 ;
   print("Send Ping counter...") ; (pingservice ! ping(counter)) , (status: Ping ) > >
   < server : PingServer | environment: < count := 0 ,(status: Waiting ) > > connector
   | client: < client . pingservice : RequiredPort | buffer: noMsg > --> server: <
   server . pingservice : ProvidedPort | buffer: noMsg > .
8  endm

```

LISTING 5.10: System module corresponding to PingConfig configuration.

After obtaining the generated Maude specification, it remains to verify and analyze the developed models, which will be the next section's subject.

### 5.5.3 Simulation and analysis

In Maude, simulating a behavior involves transforming the initial state to another by applying one or more rewriting rules. Therefore, behavior means a sequence of rewriting steps. Maude offers three main ways to simulate and analyze the modules: rewriting, search, and LTL model checking [15].

#### 5.5.3.1 Rewriting

Maude's *rewrite* and *fair rewrite* commands simulate one system behavior starting with a given initial state. The following command simulates our Maude specification from the initial state

*PingConfig* [17]. The bracketed number that appears in the first command provides an upper limit for the allowed number of rules that will be applied.

```
Maude> frew [27] PingConfig .
frewrite in PINGCONFIG : PingConfig .
rewrites: 130 in 4ms cpu (2ms real) (32500 rewrites/second)
result Configuration: (connector | client: < client . pingservice : RequiredPort | buffer:
↳ ping(2) > --> server: < server . pingservice : ProvidedPort | buffer: noMsg >)< client :
↳ PingClient | environment: < noAction ,(status: Waiting) ; (count-max = 5) ; (req' = 1) ;
↳ (counter = 2) > > < server : PingServer | environment: < noAction, (status: Waiting) ; (
↳ req = 1) ; (count = 1) > >
```

FIGURE 5.8: Execution result of the fair rewrite command.

We interpret the results as follows: after applying twenty-seven rewriting rules, the *client* instance sent two *ping* messages (*counter* = 2) and received a single *pong* message (*req'* = 1), where it awaits the following *pong* message (*status: Waiting*). On the other side, the *server* instance awaits the following *ping* message (*status: Waiting*), where it received one *ping* message (*req* = 1) and sent one *pong* message. Moreover, *no actions* to run for both client and server instances. The connector contains the *ping* (2) message in the client *buffer* (i.e., there is an undelivered message).

```
Maude> frew PingConfig .
frewrite in PINGCONFIG-PREDS : PingConfig .
rewrites: 560 in 8ms cpu (9ms real) (70000 rewrites/second)
result Configuration: (connector | client: < client . pingservice : RequiredPort | buffer:
↳ noMsg > --> server: < server . pingservice : ProvidedPort | buffer: noMsg >)< client :
↳ PingClient | environment: < noAction,(status: OK) ; (count-max = 5) ; (counter = 5) ; (req'
↳ = 5) > > < server : PingServer | environment: < noAction,(status: Waiting) ; (req = 5) ;
↳ (count = 5) > >
```

FIGURE 5.9: Execution result of the rewriting to a terminal state.

Figure 5.9 [17] shows rewriting to a terminal state. In this state, the *client* instance sent five *ping* messages (*counter* = 5) and received five *pong* messages (*req'* = 5), where it passes to the *Ok* final state. On the other side, the *server* instance received five *ping* messages (*req* = 5) and sent five *pong* messages. Moreover, *no actions* to run for both client and server instances. The connector has no messages in the client and server *buffer* (i.e., all messages are delivered).

### 5.5.3.2 Search

All the system state space reachable from an initial state is explored in different ways to find the states satisfying a given search pattern. The following search command shows that no state with *counter* > *count-max* can be reached starting with the initial state *PingConfig* in the module *PINGCONFIG* (i.e., the number of sent ping messages never exceeds the allowed maximum number).

```
Maude> search PingConfig =>* < client : PingClient | environment: < A:Action, st:Store ;
↳ (counter = N:Nat) ; st':Store > > C:Configuration such that N:Nat > 5 .
search in PINGCONFIG : PingConfig =>* C:Configuration < client : PingClient | environment: <
↳ A:Action, st:Store ; (counter = N:Nat) ; st':Store > > such that N:Nat > 5 = true .
No solution.
states: 566 rewrites: 13362 in 184ms cpu (185ms real) (72619 rewrites/second)
```

FIGURE 5.10: Result of the search command on states with *counter* > *count-max*.

On the other hand, if we search one state in which the *counter* = 5, the Maude system finds many solutions satisfying this condition. The following code describes the search command and the Maude system response.

```
Maude> search [1] PingConfig =>* < client : PingClient | environment: < A:Action, st:Store ;
↳ (counter = N:Nat) ; st':Store > > C:Configuration such that N:Nat = 5 .
search in PINGCONFIG : PingConfig =>* C:Configuration < client : PingClient | environment: <
↳ A:Action, st:Store ; (counter = N:Nat) ; st':Store > > such that N:Nat = 5 .
Solution 1 (state 446)
states: 447 rewrites: 10599 in 148ms cpu (153ms real) (71614 rewrites/second)
C:Configuration --> (connector | client: < client . pingservice : RequiredPort | buffer: noMsg
↳ > --> server: < server . pingservice : ProvidedPort | buffer: noMsg >) < server :
↳ PingServer | environment: < noAction, (status: Pong) ; (req = 4) ; (count = 4) > >
A:Action --> noAction
st:Store --> (status: Ping) ; (count-max = 5) ; (req' = 4)
N:Nat --> 5
st':Store --> (ping(5) Via pingservice)
```

FIGURE 5.11: Result of the search command on a state with *counter* = 5.

### 5.5.3.3 Linear Temporal Logic Model Checking

This section discusses how to verify the correctness of a set of properties of the specification. Properties can be specific to a particular system, like those related to system states. In contrast, other properties can be generalized to several systems or programs, like the absence of the deadlock

in the system. Maude uses a model checker to check whether all system behaviors satisfy a property. Maude's model checking is based on Linear Temporal Logic (LTL) to specify models' properties. First, we define a set of atomic propositions in a new module called *PINGCONFIG-PREDS* that implements the predicates presented in Table 5.2 [17].

```

1  mod PINGCONFIG-PREDS is
2    inc PINGCONFIG .
3    inc SATISFACTION .
4    inc MODEL-CHECKER .
5    subsort Configuration < State .
6    --- Variables
7    var N : Nat . vars st st' : Store . var A : Action . var C : Configuration .
8    var P : PortId . var M : MsgId . var I : InstanceId . var T : ThingId .
9    var Ms : Msg . var Po : Port . var O : Object .
10   --- Predicates declaration
11   ops Ping Pong : InstanceId Nat -> Prop . --- Ping and Pong predicates
12   eq < I : T | environment: < A, st ; (ping(N) Via P) ; st' > > C |= Ping(I, N) = true .
13   eq < I : T | environment: < A, st ; (pong(N) Via P) ; st' > > C |= Pong(I, N) = true .
14
15   op Msg-In-Env : InstanceId -> Prop . --- The Msg-In-Env predicate
16   eq < I : T | environment: < A , st ; (Ms Via P) ; st' > > C |= Msg-In-Env(I) = true .
17
18   op Msg-In-Buffer : -> Prop . --- The Msg-In-Buffer predicate
19   ceq (connector | client: < I . P : Po | buffer: Ms > --> server: O) C
20     |= Msg-In-Buffer = true if Ms /= noMsg .
21   ceq (connector | client: O --> server: < I . P : Po | buffer: Ms >) C
22     |= Msg-In-Buffer = true if Ms /= noMsg .
23
24   op Event-In-Env : InstanceId -> Prop . --- The Event-In-Env predicate
25   eq < I : T | environment: < A , st ; (P ? M) ; st' > > C |= Event-In-Env(I) = true .
26
27   op Action-In-Env : InstanceId -> Prop . --- The Action-In-Env predicate
28   ceq <I : T | environment: <A , st > > C |= Action-In-Env(I) = true if A /= noAction .
29
30   op Waiting : InstanceId -> Prop . --- The Waiting predicate
31   eq < I : T | environment: < A , ( status: Waiting) ; st > > C |= Waiting(I) = true .
32  endm

```

LISTING 5.11: PINGCONFIG-PREDS system module.

TABLE 5.2: Description of the atomic propositions

Predicate	Description
<i>Ping</i>	True when the instance sends the ping message
<i>Pong</i>	True if the instance responded with the pong message
<i>Msg-In-Env</i>	This predicate describes the pending messages in the instance environment
<i>Msg-In-Buffer</i>	Returns true if there are messages in the connector buffers
<i>Action-In-Env</i>	Return true if actions are being run
<i>Waiting</i>	The instance in the Waiting state

We define the formulas of the propositional LTL from this set of atomic propositions. LTL formulas are composed of atomic propositions and logical operators that include the traditional operators of propositional calculus (not:  $\sim$ , and:  $\wedge$ , or:  $\vee$ , implies:  $\rightarrow$ , and equivalent:  $\leftrightarrow$ ) and temporal operators (eventually:  $\langle \rangle$ , always:  $\square$ , ... ).

**Property 1.** The first property to check is: “If the client sends a ping message, the server will respond by a pong message”. This property is specific to this case study, it is linked to the message defined in the *PingPong* design. The following command checks whether the specification will satisfy this property:

```
Maude> red modelCheck(PingConfig, [] (Ping(client , N:Nat) -> <> Pong(server, N:Nat))) .
reduce in PINGCONFIG-PREDS : modelCheck(PingConfig, [] (Ping(client, N) -> <> Pong(server, N)))
↪ .
rewrites: 12961 in 184ms cpu (186ms real) (70440 rewrites/second)
result Bool: true
```

FIGURE 5.12: Verification result of the 1st property.

The result shows that the model checker returns the Boolean value *true*, which means that this property is satisfied. Otherwise, the model checker will provide a counterexample.

**Property 2.** The second property: “At the end of the execution, all the messages in the system will be consumed”. In other words, the connector buffers are empty and no events, and no messages in the instance environments.

```
Maude> red modelCheck(PingConfig, <>( [] ~ (Msg-In-Buffer \\/ Msg-In-Env(client) \\/
↪ Msg-In-Env(server) \\/ Event-In-Env(client) \\/ Event-In-Env(server) ))) .
reduce in PINGCONFIG-PREDS : modelCheck(PingConfig, <> [] ~ (Event-In-Env(server) \\/
↪ (Event-In-Env(client) \\/ (Msg-In-Env(server) \\/ (Msg-In-Buffer \\/ Msg-In-Env(client)))))) .
rewrites: 13368 in 196ms cpu (196ms real) (68204 rewrites/second)
result Bool: true
```

FIGURE 5.13: Verification result of the 2nd property.

**Property 3.** Next, we check if “All actions in the system will be executed”. Properties 2 and 3 are general properties that can be checked in any system. However, the actual formulas of these properties are system specific because they contain the system instances’ identifiers.

```
Maude> red modelCheck(PingConfig, [] (Action-In-Env(client) -> <> ~ Action-In-Env(client)) /\
  ↪ [] (Action-In-Env(server) -> <> ~ Action-In-Env(server))) .
reduce in PINGCONFIG-PREDS : modelCheck(PingConfig, [] (Action-In-Env(client) -> <> ~
  ↪ Action-In-Env(client)) /\ [] (Action-In-Env(server) -> <> ~ Action-In-Env(server))) .
rewrites: 14109 in 192ms cpu (195ms real) (73484 rewrites/second)
result Bool: true
```

FIGURE 5.14: Verification result of the 3rd property.

**Property 4.** Finally, we verify if “Both client and server instances will never hang in a state where their status is in *Waiting*”. That is the absence of the deadlock in the system.

```
Maude> red modelCheck(PingConfig, ~ <>( [] (Waiting(client) /\ Waiting (server)))) .
reduce in PINGCONFIG-PREDS : modelCheck(PingConfig, ~ <> [] (Waiting(client) /\
  ↪ Waiting(server))) .
rewrites: 12966 in 180ms cpu (182ms real) (72033 rewrites/second)
result Bool: true
```

FIGURE 5.15: Verification result of the 4th property.

The result of previous commands is always the Boolean value *true*, which means that the *PingPong* design satisfies all preceding properties. To show the practical utility of the proposed approach to finding problems with ThingML designs. We introduce a bug in the *PingPong* specification and show how the analysis exposes it. Bugs in ThingML designs can be caused by many reasons, such as consistency between things statecharts or forgetting to describe a transition or action in things statecharts. In our case study, we modify the *PingServer* thing implementation, where we remove the entering action from the *Pong* state. Listing 5.12 shows the new ThingML specification of the *PingServer* thing that contains a bug [17].

```
5  thing PingServer includes PingMsgs {
6    provided port pingservice {
7      sends pong
8      receives ping }
9    statechart PingServerMachine init Waiting {
10     property count : UInt8 = 0
11     state Waiting{
12       transition -> Pong  event m : pingservice?ping
13                           action count = m.req }
14     state Pong{
15       on exit print "Send Pong", count, "..."
16       transition -> Waiting }
17  }
```

LISTING 5.12: A new ThingML implementation of *PingServer* thing.



After transforming the new ThingML specification into the corresponding Maude code, we verify the absence of the deadlock (Property 4). The following command checks whether the new specification will satisfy this property.

```
Maude> red modelCheck(PingConfig,~ <>( [] (Waiting(client) /\ Waiting (server)))) .
reduce in PINGCONFIG-PREDS : modelCheck(PingConfig, ~ <> [] (Waiting(client) /\
  ↪ Waiting(server))) .
rewrites: 86 in 0ms cpu (3ms real) (~ rewrites/second)
result ModelCheckResult: counterexample({(connector | client: < client . pingservice :
  ↪ RequiredPort | buffer: noMsg > --> server: < server . pingservice : ProvidedPort | buffer:
  ↪ noMsg >) < client : PingClient | environment: < count-max := 5 ; counter := 1 ; print("Send
  ↪ Pingcounter...") ; (pingservice ! ping(counter)) ; noAction,(status: Ping) > > < server :
  ↪ PingServer | environment: < count := 0 ; noAction,(status: Waiting) > >, 'Act-R}
{(connector | client: < client . pingservice : RequiredPort | buffer: noMsg > --> server: <
  ↪ server . pingservice : ProvidedPort | buffer: noMsg >) < client : PingClient | environment:
  ↪ < counter := 1 ; print( "Send Pingcounter...") ; (pingservice ! ping(counter)) ;
  ↪ noAction,(status: Ping) ; (count-max = 5) > > < server : PingServer | environment: < count
  ↪ := 0 ; noAction,(status: Waiting) > >, 'Act-R}

  ... (Several system states are displayed here)

{(connector | client: < client . pingservice : RequiredPort | buffer: noMsg > --> server: <
  ↪ server . pingservice : ProvidedPort | buffer: noMsg >) < client : PingClient | environment:
  ↪ < noAction,(status: Waiting) ; (count-max = 5) ; (counter = 1) > > < server : PingServer |
  ↪ environment: < noAction,(status: Waiting) ; (req = 1) ; (count = 1) > >,deadlock})
```

FIGURE 5.16: Verification result of the 4th property on the modified specification.

The results show that the property is not satisfied with the modified specification. According to the counter-example path, the client and server instances will hang in a state where their status is in Waiting. We can correct our model based on the detailed execution trace provided by the model checker (counter-example).

## 5.6 Conclusion

In this chapter, we have proposed an approach that allows the automatic translation of IoT systems designs described using ThingML into a Maude specification. We have defined Maude structures to describe all ThingML components and their behavioral aspects. We have defined a big-step semantics (evaluation semantics) for actions and functions described by the ThingML action language. Then, we have implemented these semantics in the Maude language. Our approach aims to jointly apply the ThingML and Maude languages to integrate and benefit from their advantages. The transformation gives precise semantics to the ThingML language and

---

benefits from the Maude environment to analyze and verify the obtained Maude specifications. Experimental results using a case study show that our approach can generate an executable specification in the Maude environment, effectively allowing the simulation of ThingML designs and verifying IoT systems' properties. Using Maude's LTL model checker allows us to check the desirable (or undesirable) properties that must be guaranteed within a system under development. Linear Time Logic (LTL) is used to define these properties.

---

## CHAPTER 6

---

# A SIMULATION-BASED MDE APPROACH

## 6.1 Introduction

THE ThingML approach is based on a Domain-Specific Language (DSL) and a code generation framework. The ThingML textual DSL allows describing IoT applications in a platform independent way. In ThingML models, the dynamic behavior of components is described using a mix of state charts, communication by asynchronous messages, a platform-independent action language, and target languages. Therefore, these specifications include many details that decrease their legibility and comprehension since they are expressed only in textual form. On another side, The ThingML code generation framework allows the generation of application source code in several languages from ThingML specifications, where several hardware platforms support the application's source code. However, the ThingML approach does not allow rapid prototyping and experimentation to expedite the evaluation and testing of generated codes before deploying in IoT devices.

In this context, we propose an MDE and simulation-based approach to quickly develop and test IoT applications. It can help users to quickly create and test IoT applications. More precisely, we develop a hybrid (graphical-textual) modeling editor for ThingML to facilitate the development process. We also adopt a simulation approach using Proteus software to evaluate the generated codes.

## 6.2 General overview

Figure 6.1 gives a general overview of our approach [20]. We use the ThingML language to describe IoT applications using the developed hybrid editor. This hybrid editor facilitates and

speeds the modeling process and helps to clarify and better understand textual models. After that, we transform the obtained specifications into a source code using the ThingML generator code. For the analysis purposes, we finally use the Proteus software for rapid prototyping of the application hardware circuit, and then simulating the previously generated source code on this circuit.

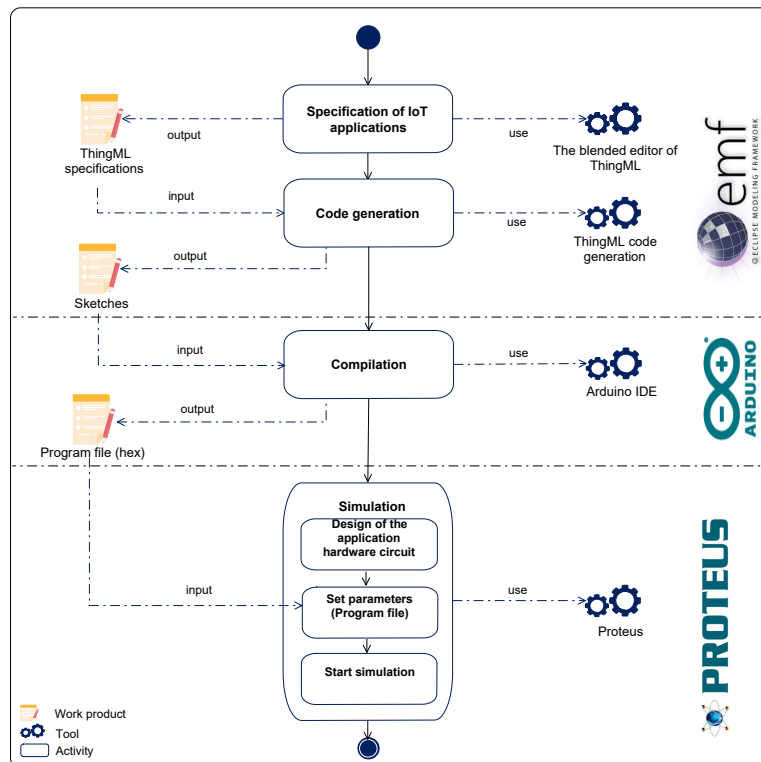


FIGURE 6.1: The workflow of the simulation based-approach.

Our approach is structured according to the following steps [20]:

- **Step 1.** Design and modeling of IoT applications using the ThingML language.
- **Step 2.** The code generation is done using the ThingML code generation framework. In this work, we will use the Arduino hardware platform. The resulting source code is called *sketches*.
- **Step 3.** Compiling the *sketches* code into *object code* that the Arduino runs (an extension file ".hex"). Arduino IDE is used to perform this compilation.
- **Step 4.** The simulation will be performed with Proteus software. It consists of three sub-operations: design of the hardware circuit of the application, the definition of the parameters

(Program file), and launching the simulation.

The following section presents the underlying technologies used in this work.

## 6.3 Underlying technologies

This section provides an overview of the Eclipse modeling project, the Arduino platform, and the Proteus software utilized in this work.

### 6.3.1 Eclipse Modeling Project (EMP)

The Eclipse Modelling Project (EMP) [117] is a collection of frameworks and tools for the Model Driven Engineering on the Eclipse platform. In short, they provide a wide range of solutions for various aspects of model driven development, from language definition, generative development of language editors to code generation as well as model verification and validation. In the following, some of the tools from Eclipse Modelling Project that have been used in this work are introduced.

#### 6.3.1.1 Eclipse Modeling Framework (EMF)

The EMF project [97] is an open-source modeling framework that forms the basis for building all the frameworks and tools in the Eclipse modeling project. It uses the object-oriented *meta-modeling* language *Ecore* to define the developed modeling languages' abstract syntax (*meta-model*). To define the concrete syntax and build editors' workbenches for the modeling language, the designers can use many EMF-based frameworks like *Sirius* and *Xtext*.

#### 6.3.1.2 Sirius Framework

The Sirius framework [58] is an open-source Eclipse project that enables the creation of graphical DSL editors. It is built on the Eclipse Modelling technologies, like EMF and GMF. The purpose is to give a generic workbench for model-based architecture engineering that could simplify production, reduce design time, and increase productivity when building a graphical editor [118]. In contrast to GMF, which generates a massive quantity of very complex code, with Sirius, no code is generated from the specification but interpreted. Consequently, the changes to the specification

immediately come into effect without the need to run a new eclipse configuration. Sirius is also integrated with other technologies that give more strength to the tool, like the *Sirius-Xtext integration*.

### 6.3.1.3 Xtext Framework

Xtext [59] is an EMF component that enables the building of text-based DSLs. It defines a DSL grammar using an Extended Backus-Naur Form (EBNF)-like language, which may be used to produce a metamodel and associated infrastructure such as a parser and linker. Xtext provides a comprehensive text editor with developer-help features such as syntax highlighting, error indicators, and code completion.

### 6.3.2 Arduino platform

In the field of electronics, Arduino [22] is a well-known open-source platform. It has been intended to be user-friendly for those without prior knowledge of electronics. Arduino enables the creation of things capable of controlling a motor, turning on a light, and sending alerts, among other functions. It is mainly based on two components: hardware and software.

- *Arduino hardware*: the Arduino board can control and respond to the components connected to it. These components may be sensors or actuators (lights and LEDs, relays, displays, motors) that allow them to communicate with the outside world.
- *Arduino software*: the Arduino board may readily be programmed utilizing Arduino integrated development environment (IDE). The IDE allows users to create software programs known as *sketches* using a simplified version of C++. Then, it converts those programs to object code compatible with Arduino hardware.

### 6.3.3 Proteus software

Developed by Labcenter Electronics, Proteus [21] is a software suite used primarily to design electrical schematics. It comprises packages such as Proteus PCB Design for the printed circuit

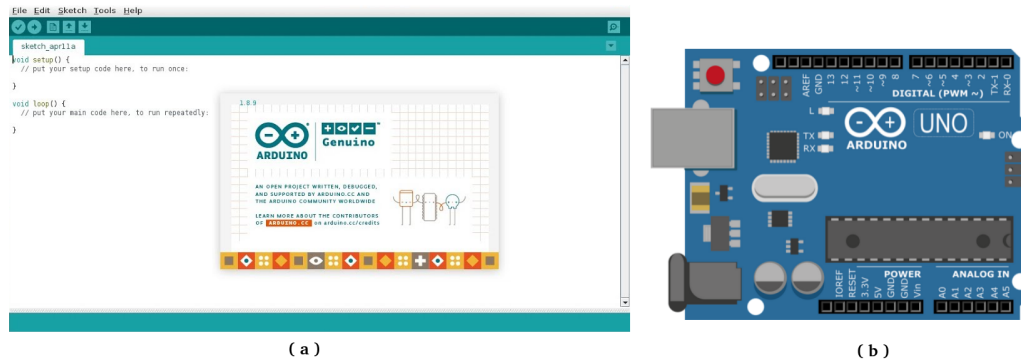


FIGURE 6.2: (a) Arduino IDE. (b) Arduino Uno board.

and Proteus Virtual System Modeling (VSM) for the simulation. The VSM package is used to simulate circuits with microprocessors. It enables rapid prototyping of hardware and firmware designs. Proteus VSM enables the simulation of the interaction between software running on a microcontroller and any analog or digital electronics connected to it. Also, it simulates the execution of object code, just like a real chip. Besides, Proteus VSM supports several microprocessor families (PIC16, PIC18, AVR, Arduino, ...).

## 6.4 The hybrid graphical-textual modeling editor

Several notations can be used as concrete syntaxes for DSL. It can be textual, graphical, tabular, form-based, or a combination of these. Each of these notations provides benefits that are unavailable in the other notations. The combination of multiple notations permits accumulating the benefits of each notation, and it may reveal a variety of advantages [119]. Nevertheless, the modeling frameworks traditionally relied on one specific editing notation. Using a single type of notation has the disadvantage of restricting the tools available for developing and manipulating models that may be required. On the other hand, a modeling framework based on multi-notations might reveal many advantages and provide a better performance regarding single-notation modeling [119]. Hybrid (or blended) modeling can be defined as manipulating a common underlying model resource using several editors based on different notations [119]. Flexible separation of concerns, enhanced human comprehension, better communication, multi-view modeling based on different notations, the ability to manipulating the models outside a modeling environment, and speedier-modeling tasks are among the features. The study presented in [119] shows the potential benefits of hybrid modeling for more details.

The ThingML approach is based on a textual Domain-Specific Language (DSL), where the dynamic behavior of components is described using a mix of state charts, communication by asynchronous messages, a platform-independent action language, and target languages. Therefore, these specifications can include many details that decrease their legibility. In this context, we develop a hybrid modeling editor for the ThingML language. The hybrid editors present the best modeling solutions that combine textual notations with graphical notations and accumulate their advantages [119]. It is well known that graphical specification is better suited for describing the system components and their relationships. States and transitions in the state machine are specified graphically, whereas actions in states and guards in transitions are specified using a textual expression language. Thus, each system aspect will be described using the most appropriate view (textual/graphical). Therefore, the hybrid editor for the ThingML language can speed up modeling tasks and facilitate understanding. We use well-known frameworks and tools under the Eclipse platform to achieve this goal, such as Sirius and Xtext.

#### **6.4.1 Xtext-based editor**

ThingML is an open-source project [98] presented as textual SDL to develop IoT applications. Its syntax is defined using the Xtext framework [59], which uses an EBNF-like language to define the DSL grammar. The Xtext framework uses the EBNF grammar to automatically generate a comprehensive text editor and a meta-model. The textual editor provides developer help features such as syntax highlighting, error detection markers, and auto-completion. Based on the generated meta-model, we will develop a graphical editor where we define a graphical representation for each meta-model class.

#### **6.4.2 Sirius-based editor**

The ThingML graphical editor [20] has been developed utilizing the Sirius framework. Starting from the generated metamodel, Sirius allows a model-based specification of visual concrete syntax organized in viewpoints. Graphical and textual editors are synchronized thanks to the Sirius and Xtext integration. We have developed three Viewpoints for viewing and editing Things, state



machines and configurations. Figure 6.3 shows the structure of the state machine Viewpoint. Other figures on the graphic editors are shown in subsection 6.5.1 (See Figures 6.4, 6.5, and 6.6).

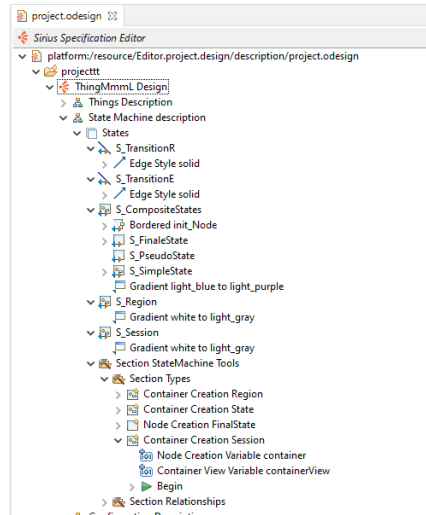


FIGURE 6.3: The structure of the state machine Viewpoint

## 6.5 Case study

This section presents the experimental results of the case study of the traffic light controller. We use ThingML version 2.0.0.202107160746 (available in [98] ) as code generator. We also conduct a series of simulations under the Proteus software (version 18.1). This case study aims to build a simple traffic light controller using the Arduino platform. We hope this case study will help the reader understand the fundamental design concepts of ThingML. Three LEDs (Light-Emitting Diodes) are used in this system: green, yellow, and red. They are linked to an Arduino Uno board. The traffic light works endlessly over time in the same way: it will remain green for 8 seconds, yellow for 3 seconds, and red for 5 seconds.

### 6.5.1 Specification

The ThingML language is used to model this system, where we have mainly defined two *Things* and a *Configuration*. Figure 6.4 [20] presents a graphical view of the *Traffic\_Light* application components. It shows the Things, their relationships, and their messages and ports. Listing 6.1 describes the *LED* Thing that presents the LED controller software. It comprises a provided port, two functions, and a state machine. The state machine describes an implementation of the behavior of this Thing and has two states: *ON* and *OFF*, with the *OFF* state being the initial. It

changes from the *OFF* to the *ON* state upon the event *ctrl?ledON* (the *ledON()* message arriving over the *ctrl* port). Similarly, when the event *ctrl?ledON* is triggered, the Thing changes state from *ON* to *OFF*.

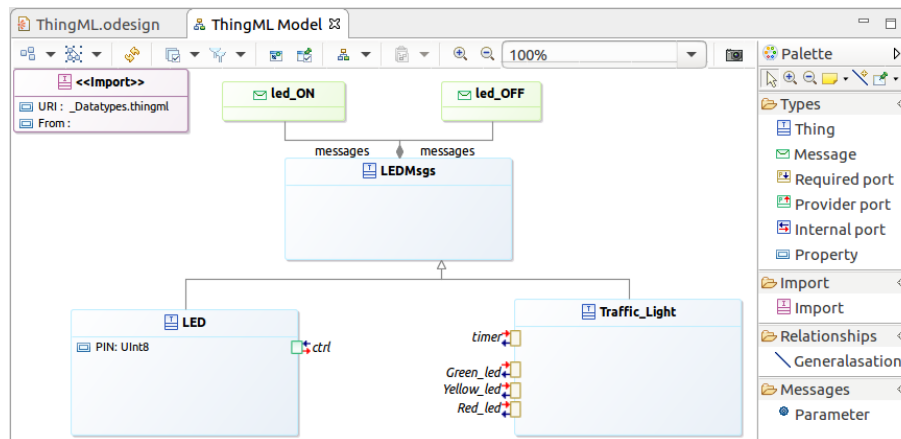


FIGURE 6.4: Graphical view of Things in the Traffic\_Light application.

```

1  thing LED includes LEDMsgs {
2    property PIN: UInt8 = 10
3    provided port ctrl { receives led_ON, led_OFF }
4    function setDigitalOutput(pin: UInt8) do
5      `pinMode(`&pin&`, OUTPUT);`
6    end
7    function digitalWrite(pin: UInt8, value : DigitalState) do
8      `digitalWrite(`&pin&`, `&value&`);`
9    end
10   statechart LED init OFF {
11     on entry setDigitalOutput(PIN)
12     state OFF {
13       transition -> ON event ctrl?led_ON
14         action digitalWrite(PIN, DigitalState:HIGH)    }
15     state ON {
16       transition -> OFF event ctrl?led_OFF
17       action digitalWrite(PIN, DigitalState:LOW)
18     }
19   }
20 }

```

LISTING 6.1: ThingML implementation of the LED thing.

*Traffic\_Light*'s second software component describes the implementation of the traffic light controller (see Listing 6.2). This Thing is comprised of four ports and a state machine. The ports will be used to connect to the LEDs. A state machine describes the Thing's behavior with three states: *RED*, *GREEN*, and *YELLOW*. It is initially in the *RED* state and switches from one state to another when the *timer?Timer.timeout* event is triggered (that is, at the end of the time allotted for each state). When the Thing enters or exits these states, it acts, on entering, it sends

the *led\_ON* message through the associated port, and on exit, it sends the *led\_OFF* message via the corresponding port.

```

1  thing Traffic_Light includes TimerMsgs , LEDMsgs {
2      required port timer { sends timer_start receives timer_timeout }
3      required port Red_led { sends led_ON led_OFF }
4      required port Green_led { sends led_ON led_OFF }
5      required port Yellow_led { sends led_ON led_OFF }
6      statechart Traffic_Light init RED {
7          state RED {
8              on entry do
9                  Red_led!led_ON()
10                 timer!timer_start(5000)
11             end
12             transition -> GREEN event timer?timer_timeout
13             on exit Red_led!led_OFF() }
14         state GREEN {
15             on entry do
16                 Green_led!led_ON()
17                 timer!timer_start(8000)
18             end
19             transition -> YELLOW event timer?timer_timeout
20             on exit Green_led!led_OFF() }
21         state YELLOW {
22             on entry do
23                 Yellow_led!led_ON()
24                 timer!timer_start(3000)
25             end
26             transition -> RED event timer?timer_timeout
27             on exit Yellow_led!led_OFF() } }
28     }
29 }

```

LISTING 6.2: ThingML implementation of the TrafficLight thing.

The graphical view of the state diagram of the *Traffic\_Light* Thing is presented in Fig. 6.5 [20].

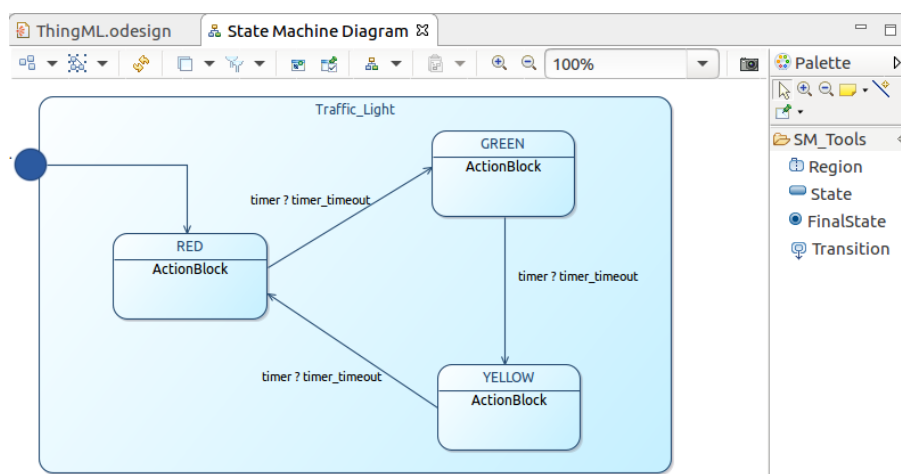


FIGURE 6.5: Graphical view of thing Traffic\_Light state chart.

The *Traffic\_Light\_App* configuration, shown in Listing 6.3, presents the concrete application. It consists of four instances; three *LED* and one *Traffic\_Light*. These instances are interconnected using four *connectors*. Each connector must link a required port with another provided port. Each instance's *PIN* property is initialized with a value corresponding to the genuine value of the Arduino board's *Digital Pic*.

```

1  configuration Traffic_Light_App {
2      instance traffic_light : Traffic_Light
3      instance red_led : LED
4      set red_led.PIN = 11
5      instance green_led : LED
6      set green_led.PIN = 12
7      instance yellow_led : LED
8      set yellow_led.PIN = 13
9      connector traffic_light.Red_led => red_led.ctrl
10     connector traffic_light.Green_led => green_led.ctrl
11     connector traffic_light.Yellow_led => yellow_led.ctrl
12     connector traffic_light.timer over Timer
13 }

```

LISTING 6.3: ThingML implementation of Traffic\_Light\_App configuration.

The graphical view of the *Traffic\_Light\_App* configuration is shown in Fig. 6.6 [20].

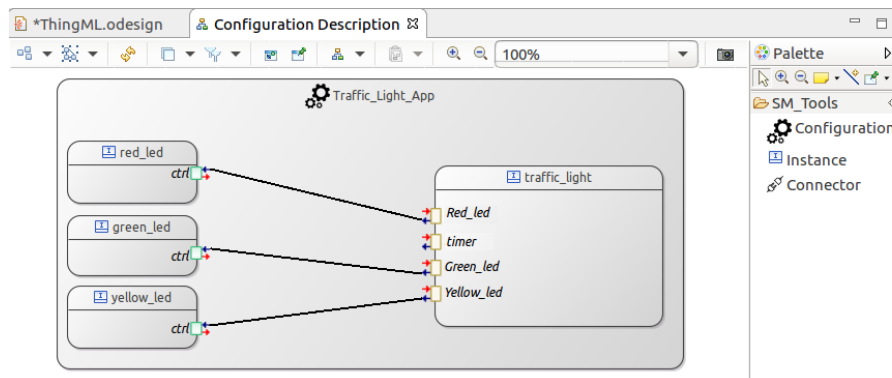
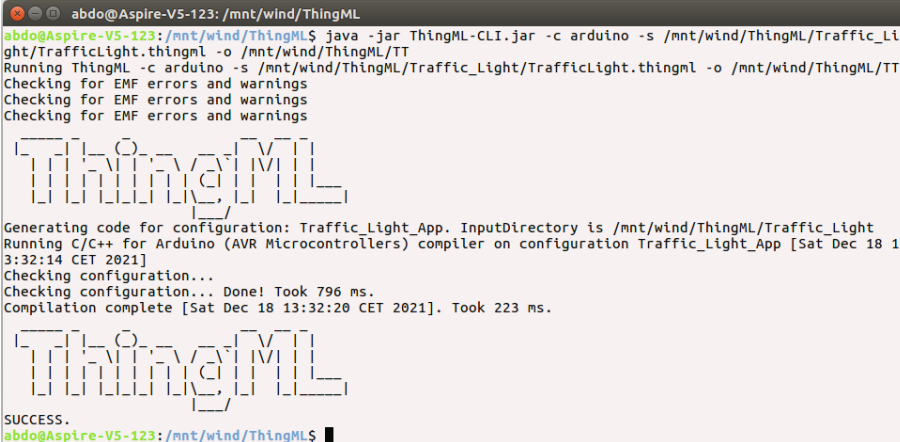


FIGURE 6.6: Graphical view of the Traffic\_Light\_App configuration.

### 6.5.2 Code generation

After the system modeling, the ThingML code generator will be used to generate the code source automatically. There are two methods, either by using the Eclipse platform or the Jar file. Fig. 6.7 shows the code generation process using the Jar file.



```

abdo@Aspire-V5-123: /mnt/wind/ThingML
abdo@Aspire-V5-123: /mnt/wind/ThingML$ java -jar ThingML-CLI.jar -c arduino -s /mnt/wind/ThingML/Traffic_Light/TrafficLight.thingml -o /mnt/wind/ThingML/TT
Running ThingML -c arduino -s /mnt/wind/ThingML/Traffic_Light/TrafficLight.thingml -o /mnt/wind/ThingML/TT
Checking for EMF errors and warnings
Checking for EMF errors and warnings
Checking for EMF errors and warnings

Generating code for configuration: Traffic_Light_App. InputDirectory is /mnt/wind/ThingML/Traffic_Light
Running C/C++ for Arduino (AVR Microcontrollers) compiler on configuration Traffic_Light_App [Sat Dec 18 13:32:14 CET 2021]
Checking configuration...
Checking configuration... Done! Took 796 ms.
Compilation complete [Sat Dec 18 13:32:20 CET 2021]. Took 223 ms.

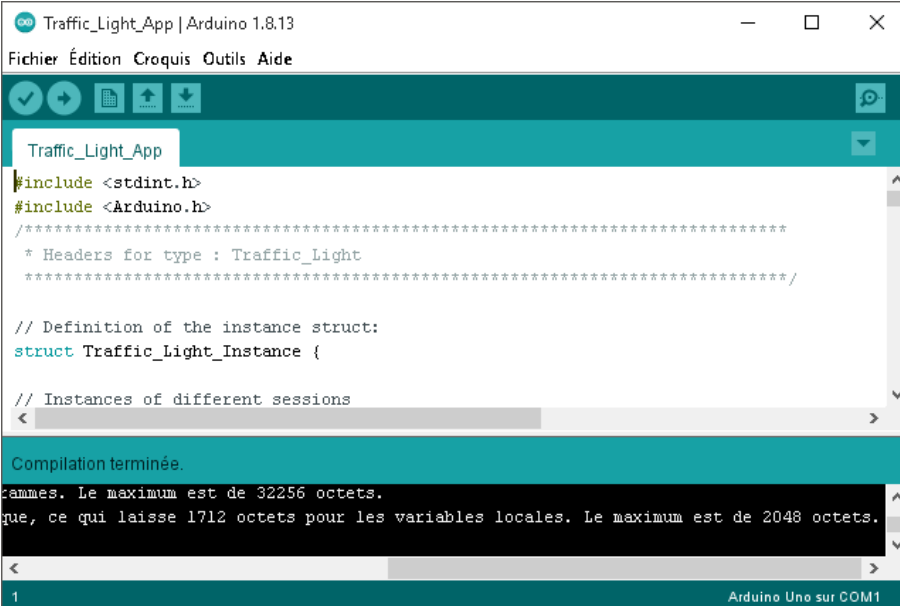
SUCCESS.
abdo@Aspire-V5-123: /mnt/wind/ThingML$

```

FIGURE 6.7: The code generation using Jar file.

### 6.5.3 Compilation of the sketches

The next step in the process is to compile the *sketches*. The compilation step is done with the Arduino IDE. Figure 6.8 presents the results of the compilation, showing that the sketches are compiled correctly. The result is an *object file* (with *.hex* extension).



```

Traffic_Light_App | Arduino 1.8.13
Fichier Édition Croquis Outils Aide

#include <stdint.h>
#include <Arduino.h>
/*****
 * Headers for type : Traffic_Light
 *****/

// Definition of the instance struct:
struct Traffic_Light_Instance {

// Instances of different sessions
<

Compilation terminée.
rammes. Le maximum est de 32256 octets.
que, ce qui laisse 1712 octets pour les variables locales. Le maximum est de 2048 octets.
1
Arduino Uno sur COM1

```

FIGURE 6.8: The compilation of traffic light specification.

### 6.5.4 Simulation

In the simulation step, we first build the application hardware circuit. The traffic light hardware circuit seen in Fig. 6.9 [20] was created using the Proteus program. It consists of an Arduino Uno board and three LEDs. The red, yellow, and green LEDs are connected to the 11, 12, and

13 digital pins of Arduino Uno, respectively. We then define the program file of the Arduino component by the object file obtained in the compilation step. Finally, we simulate the code obtained in the developed circuit.

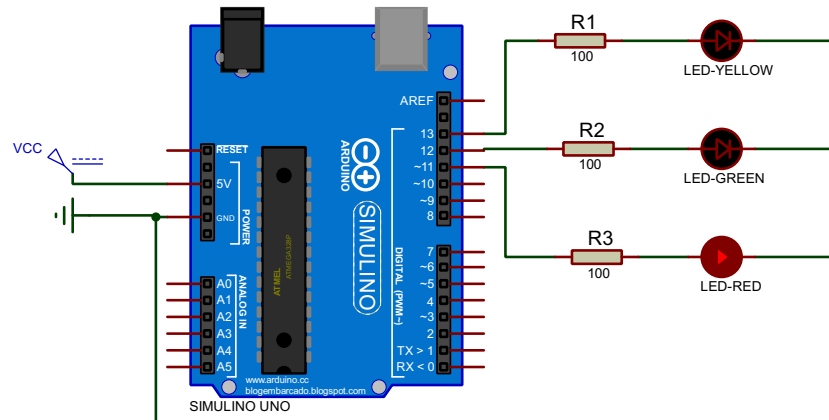


FIGURE 6.9: The traffic light hardware circuit

We have conducted a series of simulations that demonstrate that the source code generated by the ThingML code generation from our specification is executed correctly on the circuit developed in Proteus. Table 6.1 presents the simulation results for 10, 20, and 50 traffic light cycles (a cycle is the switching between Red, Green, Yellow, and then Red) [20].

TABLE 6.1: The simulation results

	10 cycles	20 cycles	50 cycles
Execution time	160 s	320 s	800 s
Result	executed correctly	executed correctly	executed correctly

## 6.6 Conclusion

In this chapter, we have presented an approach to design and simulate IoT applications. In this approach, we have used ThingML to design the applications and generate a source code from this specification using ThingML code generation, where we have developed a hybrid editor for the ThingML language. Then, we have used the Proteus software to build the application circuit and to simulate generated code on this circuit. The simulation results demonstrate that the source code generated is executed correctly on the circuit developed. Consequently, the users can test their applications without the availability of IoT devices.

# General conclusion

---

THE last few years have seen a strong development in the use of Internet of Things technologies, where a large number of user devices need to collaborate in order to perform a common task. The availability of many diverse heterogeneous devices collaborating in the IoT represents an unprecedented opportunity to improve the quality of life, along with the quality of service, through collaboration among industrial and consumer devices. However, to benefit from the IoT advantages, a whole host of new challenges must be addressed at all levels. MDE can help meet the technical challenges of IoT system development.

In this thesis, we are interested in proposing an approach for modeling and analyzing IoT applications based on Model-Driven Engineering (MDE). We have made a tour of the paradigm of the IoT, as well as the modeling of IoT systems using the MDE approach. We have shown the existing approaches in the literature by making a comparative study that allows discussing the advantages and disadvantages for each of them. We then discussed the two languages used, namely ThingML, as a semi-formal modeling language, and Maude, as a formal specification language. After that, we have proposed an MDE-based formal approach, which aims to jointly apply the ThingML and Maude languages to integrate and benefit from their advantages. It consists in using ThingML for modeling IoT applications and benefiting from a set of code generators for various platforms. This is followed by the automatic translation of ThingML specifications into Maude code to allow verification and analysis. We have defined Maude structures to describe all ThingML components and their behavioral aspects. We have defined big-step semantics (evaluation semantics) for actions and functions described by the ThingML action language. Then, we implemented these semantics in the Maude language. The transformation gives precise semantics to the ThingML language and benefits from the Maude environment to analyze and

verify the obtained Maude specifications. Experimental results using a case study show that our approach can generate an executable specification in the Maude environment, effectively allowing the simulation of ThingML designs and verifying IoT systems' properties. Using Maude's LTL model checker allows us to check the desirable (or undesirable) properties that must be guaranteed within a system under development. Linear Time Logic (LTL) is used to define these properties.

In a second contribution, we have developed a hybrid textual-graphical editor that facilitates the development process and addresses some of the shortcomings of using a textual editor. We have also suggested using Proteus software as a tool to build the hardware circuit of the application as well as to simulate and test the previously generated code before deployment on IoT devices.

In the current version of the formal approach, we have implemented the semantics of the ThingML action language using rewriting rules with reachability conditions. These rewriting rules may need more computational resources, especially for significant case studies. For this reason, we plan to study the scalability of our formal framework. For that, we propose to introduce the real-time aspect using RT-Maude, which will enable us to analyze more extensive real case studies, we will try to streamline the semantics following different approaches such as presented in [120]. To wrap ups, we will try to make a critical comparison of the results of the applied approaches to choose the most appropriate. In addition, we plan to develop a formal tool based on Maude for the ThingML language that can hide formal details from designers. Such a tool allows the automatic generation of Maude LTL expressions and automatically displays the interpretation of the analysis results in the source models.



# Bibliography

---

- [1] Alem Čolaković and Mesud Hadžialić. Internet of Things (IoT): A review of enabling technologies, challenges, and open research issues. *Computer networks*, 144:17–39, 2018.
- [2] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. ThingML: a language and code generation framework for heterogeneous targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint-Malo, France, October 2-7, 2016*, pages 125–135, 2016.
- [3] Alberto Rodrigues Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.
- [4] Strategyanalytics, Global Connected and IoT Device Forecast Update, [Online]. Available: <https://www.strategyanalytics.com/access-services/devices/connected-home/consumer-electronics/reports/report-detail/global-connected-and-iot-device-forecast-update/> [Accessed June-2022].
- [5] Ala I. Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys and Tutorials*, 17(4):2347–2376, 2015.
- [6] Eleonora Borgia. The Internet of Things vision: Key features, applications and open issues. *Computer Communications*, 54:1–31, 2014.
- [7] Bruno Costa, Paulo F. Pires, and Flávia Coimbra Delicato. Towards the adoption of OMG standards in the development of SOA-based IoT systems. *J. Syst. Softw.*, 169:110720, 2020.

- 
- [8] Federico Ciccozzi, Ivica Crnkovic, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Romina Spalazzese. Model-Driven Engineering for Mission-Critical IoT Systems. *IEEE Software*, 34(1):46–53, 2017.
- [9] Brice Morin, Nicolas Harrant, and Franck Fleurey. Model-Based Software Engineering to Tame the IoT Jungle. *IEEE Software*, 34(1):30–36, 2017.
- [10] Alireza Souri and Monire Norouzi. A state-of-the-art survey on formal verification of the Internet of Things applications. *Journal of Service Science Research*, 11(1):47–67, 2019.
- [11] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science*, 96(1):73–155, 1992. Publisher: Elsevier.
- [12] Joel dos Santos, Christiano Braga, and Débora C Muchaluat-Saade. A rewriting logic semantics for NCL. *Science of Computer Programming*, 107:64–92, 2015.
- [13] Francisco Durán, Camilo Rocha, and Gwen Salaün. Stochastic analysis of BPMN with time in rewriting logic. *Science of Computer Programming*, 168:1–17, 2018.
- [14] Elhillali Kerkouche, Khaled Khalfaoui, and Allaoua Chaoui. A rewriting logic-based semantics and analysis of UML activity diagrams: a graph transformation approach. *International Journal of Computer Aided Engineering and Technology*, 12(2):237–262, 2020. Publisher: Inderscience Publishers (IEL).
- [15] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about Maude—a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.
- [16] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. *Electronic Notes in Theoretical Computer Science*, 71:162–187, 2004.
- [17] Abdelouahab Fortas, Elhillali Kerkouche, and Allaoua Chaoui. Formal verification of IoT applications using rewriting logic: An MDE-based approach. *Science of Computer Programming*, 222:102859, 2022. ISSN 0167-6423.
- [18] Carl A Gunter. *Semantics of programming languages: structures and techniques*. MIT press, 1992.

- [19] Accele, [Online]. Available: <https://www.eclipse.org/acceleo/> [Accessed June-2022].
- [20] Abdelouahab Fortas, Elhillali Kerkouche, and Allaoua Chaoui. Application of MDE in the development of IoT systems: A simulation-based approach. In *2022 First International Conference on Computer Communications and Intelligent Systems (I3CIS)*, pages 93–98. IEEE, 2022.
- [21] Proteus, [Online]. Available: <https://www.labcenter.com/> [Accessed June-2022].
- [22] Arduino, [Online]. Available: <https://www.arduino.cc> [Accessed June-2022].
- [23] Rob Van Kranenburg. *The Internet of Things: A critique of ambient technology and the all-seeing network of RFID*. Institute of Network Cultures, 2008.
- [24] Louis Coetzee and Johan Eksteen. The Internet of Things-promise for the future? an introduction. In *2011 IST-Africa Conference Proceedings*, pages 1–9. IEEE, 2011.
- [25] Ovidiu Vermesan, Peter Friess, Patrick Guillemin, Sergio Gusmeroli, Harald Sundmaeker, Alessandro Bassi, Ignacio Soler Jubert, Margaretha Mazura, Mark Harrison, Markus Eisenhauer, et al. Internet of Things strategic research roadmap. In *Internet of things-global technological and societal trends from smart environments and spaces to green ICT*, pages 9–52. River Publishers, 2022.
- [26] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer networks*, 54(15):2787–2805, 2010.
- [27] Pallavi Sethi and Smruti R Sarangi. Internet of Things: architectures, protocols, and applications. *Journal of Electrical and Computer Engineering*, 2017, 2017.
- [28] Perry Xiao. *Designing Embedded Systems and the Internet of Things (IoT) with the ARM mbed*. John Wiley & Sons, 2018.
- [29] Hemant Ghayvat, Subhas Mukhopadhyay, Xiang Gui, and Nagender Suryadevara. WSN-and IOT-based smart homes and their extension to smart buildings. *Sensors*, 15(5):10350–10379, 2015.
- [30] Gaye Abdourahime and Dieynaba Mall. Analysis of authentication mechanisms and identity management in IoT: the resource constraints and security needs. In *2021 International*

- Conference on Information Systems and Advanced Technologies (ICISAT)*, pages 1–6. IEEE, 2021.
- [31] Luca Catarinucci, Danilo De Donno, Luca Mainetti, Luca Palano, Luigi Patrono, Maria Laura Stefanizzi, and Luciano Tarricone. An IoT-aware architecture for smart health-care systems. *IEEE internet of things journal*, 2(6):515–526, 2015.
- [32] Miao Wu, Ting-Jie Lu, Fei-Yang Ling, Jing Sun, and Hui-Ying Du. Research on the architecture of Internet of Things. In *2010 3rd international conference on advanced computer theory and engineering (ICACTE)*, volume 5, pages V5–484. IEEE, 2010.
- [33] Omar Said and Mehedi Masud. Towards Internet of Things: Survey and future vision. *International Journal of Computer Networks*, 5(1):1–17, 2013.
- [34] Ibrahim Mashal, Osama Alsaryrah, Tein-Yaw Chung, Cheng-Zen Yang, Wen-Hsing Kuo, and Dharma P Agrawal. Choices for interaction with things on Internet and underlying issues. *Ad Hoc Networks*, 28:68–90, 2015.
- [35] Rafiullah Khan, Sarmad Ullah Khan, Rifaqat Zaheer, and Shahid Khan. Future internet: the Internet of Things architecture, possible applications and key challenges. In *2012 10th international conference on frontiers of information technology*, pages 257–260. IEEE, 2012.
- [36] Deze Zeng, Song Guo, and Zixue Cheng. The Web of Things: A survey. *J. Commun.*, 6(6):424–438, 2011.
- [37] Soma Bandyopadhyay, Munmun Sengupta, Souvik Maiti, and Subhajit Dutta. Role of middleware for Internet of Things: A study. *International Journal of Computer Science and Engineering Survey*, 2(3):94–105, 2011.
- [38] Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhán Clarke. Middleware for Internet of Things: a survey. *IEEE Internet of things journal*, 3(1):70–95, 2015.
- [39] Noboru Koshizuka and Ken Sakamura. Ubiquitous ID: standards for ubiquitous computing and the Internet of Things. *IEEE Pervasive Computing*, 9(4):98–101, 2010.

- [40] Maria Rita Palattella, Nicola Accettura, Xavier Vilajosana, Thomas Watteyne, Luigi Alfredo Grieco, Gennaro Boggia, and Mischa Dohler. Standardized protocol stack for the Internet of (important) Things. *IEEE communications surveys & tutorials*, 15(3):1389–1406, 2012.
- [41] Sachin Kumar, Prayag Tiwari, and Mikhail Zymbler. Internet of Things is a revolutionary approach for future technology enhancement: a review. *Journal of Big data*, 6(1):1–21, 2019.
- [42] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. Interoperability in Internet of Things infrastructure: classification, challenges, and future work. In *International Conference on Internet of Things as a Service*, pages 11–18. Springer, 2017.
- [43] Carlos Pereira and Ana Aguiar. Towards efficient mobile M2M communications: Survey and open challenges. *Sensors*, 14(10):19582–19608, 2014.
- [44] Zheng Yan, Peng Zhang, and Athanasios V Vasilakos. A survey on trust management for Internet of Things. *Journal of network and computer applications*, 42:120–134, 2014.
- [45] MA Rajan, P Balamuralidhar, KP Chethan, and M Swarnahpriyaah. A self-reconfigurable sensor network management system for Internet of Things paradigm. In *2011 International Conference on Devices and Communications (ICDeCom)*, pages 1–5. IEEE, 2011.
- [46] Debasis Bandyopadhyay and Jaydip Sen. Internet of Things: Applications and challenges in technology and standardization. *Wireless personal communications*, 58(1):49–69, 2011.
- [47] Komal Batool and Muaz A Niazi. Modeling the Internet of Things: a hybrid modeling approach using complex networks and agent-based models. *Complex Adaptive Systems Modeling*, 5(1):1–19, 2017.
- [48] Gabriele D’Angelo, Stefano Ferretti, and Vittorio Ghini. Modeling the Internet of Things: a simulation perspective. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 18–27. IEEE, 2017.
- [49] Gabor Kecskemeti, Giuliano Casale, Devki Nandan Jha, Justin Lyon, and Rajiv Ranjan. Modelling and simulation challenges in Internet of Things. *IEEE cloud computing*, 4(1):62–69, 2017.

- [50] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE'07)*, pages 37–54. IEEE, 2007.
- [51] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis lectures on software engineering*, 3(1):1–207, 2017.
- [52] Richard Soley et al. Model driven architecture. *OMG white paper*, 308(308):5, 2000.
- [53] Alan W Brown. Model driven architecture: Principles and practice. *Software and systems modeling*, 3(4):314–327, 2004.
- [54] Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Varanda João Maria Pereira, Matej Črepinšek, Cruz Daniela Da, and Rangel Pedro Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 7(2): 247–264, 2010.
- [55] Ankica Bariic, Vasco Amaral, and Miguel Goulao. Usability evaluation of domain-specific languages. In *2012 Eighth International Conference on the Quality of Information and Communications Technology*, pages 342–347. IEEE, 2012.
- [56] Benoît Combemale. Ingénierie dirigée par les modèles (IDM)–état de l’art. 2008.
- [57] OMG. Object Constraint Language (OCL), [Online]. Available: <http://www.omg.org/spec/OCL/> [Accessed June-2022].
- [58] Eclipse Sirius, [Online]. Available: <https://www.eclipse.org/sirius/> [Accessed June-2022].
- [59] Xtext, [Online]. Available: <https://eclipse.org/Xtext/> [Accessed June-2022].
- [60] Nelly Bencomo, Robert B France, Betty HC Cheng, and Uwe Aßmann. *Models@ run. time: foundations, applications, and roadmaps*, volume 8378. Springer, 2014.
- [61] Heather J Goldsby and Betty HC Cheng. Automatically generating behavioral models of adaptive systems to address uncertainty. In *International Conference on Model Driven Engineering Languages and Systems*, pages 568–583. Springer, 2008.
- [62] Ali Arsanjani. Service-oriented modeling and architecture. *IBM developer works*, 1:15, 2004.

- [63] Antonio Cicchetti, Federico Ciccozzi, Silvia Mazzini, Stefano Puri, Marco Panunzio, Alessandro Zovi, and Tullio Vardanega. CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 362–365, 2012.
- [64] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):39–91, 2006.
- [65] Kleanthis Thramboulidis and Foivos Christoulakis. UML4IoT—A UML-based approach to exploit IoT in cyber-physical manufacturing systems. *Computers in Industry*, 82:259 – 272, 2016.
- [66] Open Mobile Alliance (OMA), “Lightweight Machine to Machine Technical Specification: Core”, Approved Version: 1.1.1 - 2019 06 17, [Online]. Available: [http://www.openmobilealliance.org/release/LightweightM2M/V1\\_1\\_1-20190617-A/OMA-TS-LightweightM2M\\_Core-V1\\_1\\_1-20190617-A.pdf](http://www.openmobilealliance.org/release/LightweightM2M/V1_1_1-20190617-A/OMA-TS-LightweightM2M_Core-V1_1_1-20190617-A.pdf) [Accessed June-2022].
- [67] Federico Ciccozzi and Romina Spalazzese. MDE4IoT: supporting the internet of things with model-driven engineering. In *International Symposium on Intelligent and Distributed Computing*, pages 67–76. Springer, 2016.
- [68] Action Language For Foundational UML - ALF, [Online]. Available: <https://www.omg.org/spec/ALF/> [Accessed June-2022].
- [69] Ferry Pramudianto, Carlos Alberto Kamienski, Eduardo Souto, Fabrizio Borelli, Lucas L Gomes, Djamel Sadok, and Matthias Jarke. IoT link: An Internet of Things prototyping toolkit. In *2014 IEEE 11th Intl Conf on Ubiquitous Intelligence and Computing and 2014 IEEE 11th Intl Conf on Autonomic and Trusted Computing and 2014 IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops*, pages 1–9. IEEE, 2014.
- [70] C.M. de Farias, I.C. Brito, L. Pirmez, F.C. Delicato, P.F. Pires, T.C. Rodrigues, I.L. dos Santos, L.F.R.C. Carmo, and T. Batista. COMFIT: A development environment for the

- Internet of Things. *Future Generation Computer Systems*, 75:128–144, 2017. doi: 10.1016/j.future.2016.06.031.
- [71] J.C. Kirchof, B. Rumpe, D. Schmalzing, and A. Wortmann. MontiThings: Model-Driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software*, 183, 2022. doi: 10.1016/j.jss.2021.111087.
- [72] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Montiarc-architectural modeling of interactive distributed and cyber-physical systems. *arXiv preprint arXiv:1409.6578*, 2014.
- [73] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer Science & Business Media, 2012.
- [74] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, and Bernhard Rumpe. MontiBelle-Toolbox for a model-based development and verification of distributed critical systems for compliance with functional safety. In *AIAA Scitech 2020 Forum*, page 0671, 2020.
- [75] Dimitris Soukaras, Pankesh Patel, Hui Song, and Sanjay Chaudhary. IoTSuite: a tool-suite for prototyping Internet of Things applications. In *The 4th International Workshop on Computing and Networking for Internet of Things (ComNet-IoT), co-located with 16th International Conference on Distributed Computing and Networking (ICDCN)*, page 6, 2015.
- [76] P. Patel and D. Cassou. Enabling high-level application development for the Internet of Things. *Journal of Systems and Software*, 103:62–84, 2015. ISSN 01641212 (ISSN). doi: 10.1016/j.jss.2015.01.027. Publisher: Elsevier Inc.
- [77] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, and Alfonso Pierantonio. Towards a modeling and analysis environment for industrial IoT systems. *arXiv preprint arXiv:2105.14136*, 2021.
- [78] Mohammad Sharaf, Mai Abusair, Rami Eleiwi, Yara Shana’a, Ithar Saleh, and Henry Mucini. Modeling and Code Generation Framework for IoT. In Pau Fonseca i Casas, Maria-Ribera Sancho, and Edel Sherratt, editors, *System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0*, Lecture Notes in Computer Science, pages 99–115, Cham, 2019. Springer International Publishing. ISBN 978-3-030-30690-8.



- [79] I. Berrouyne, M. Adda, J.-M. Mottu, and M. Tisi. A Model-Driven Methodology to Accelerate Software Engineering in the Internet of Things. *IEEE Internet of Things Journal*, 9(20):19757–19772, 2022. doi: 10.1109/JIOT.2022.3170500.
- [80] L. Burgueño, J. Boubeta-Puig, and A. Vallecillo. Formalizing Complex Event Processing Systems in Maude. *IEEE Access*, 6:23222–23241, 2018.
- [81] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):1–62, 2012.
- [82] P. C. Ölveczky and J. Meseguer. Semantics and Pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007. Publisher: Springer.
- [83] Ajay Krishna and Gwen Salaün. Business process models for analysis of industrial IoT applications. In *11th International Conference on the Internet of Things*, pages 102–109, 2021.
- [84] Sofia Abbas, El Hillali Kerkouche, Khaled Khalfaoui, and Allaoua Chaoui. Combined use of PBMN and rewriting logic for specification and analysis of IoT applications. In *International Symposium on Modelling and Implementation of Complex Systems*, pages 62–75. Springer, 2023.
- [85] Francisco Durán, Ajay Krishna, Michel Le Pallec, Radu Mateescu, and Gwen Salaün. Models and analysis for user-driven reconfiguration of rule-based IoT applications. *Internet of Things*, 19:100515, 2022.
- [86] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Frédéric Lang, Christine McKinty, Vincent Powazny, Wendelin Serwe, and Gideon Smeding. Reference manual of the LNT to LOTOS translator, 2018.
- [87] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.
- [88] Mozilla, WebThings, [Online]. Available: <https://iot.mozilla.org/> [Accessed June-2022].

- [89] Bruno Costa, Paulo F. Pires, Flávia Coimbra Delicato, Wei Li, and Albert Y. Zomaya. Design and Analysis of IoT Applications: A Model-Driven Approach. In *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, DASC/PiCom/DataCom/CyberSciTech 2016, Auckland, New Zealand, August 8-12, 2016*, pages 392–399. IEEE Computer Society, 2016.
- [90] Martin Bauer, M Boussard, N Bui, F Carrez, C Jardak, J De Loof, C Magerkurth, S Meissner, A Nettsträter, A Olivereau, et al. Deliverable D1. 5—Final architectural reference model for the IoT v3. 0. *Internet of things architecture (IOT-A)*, 2013.
- [91] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.
- [92] S. Xu, W. Miao, T. Kunz, T. Wei, and M. Chen. Quantitative Analysis of Variation-Aware Internet of Things Designs Using Statistical Model Checking. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 274–285, August 2016.
- [93] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.
- [94] Flávio Oquendo. Formally Describing Self-organizing Architectures for Systems-of-Systems on the Internet-of-Things. In Carlos E. Cuesta, David Garlan, and Jennifer Pérez, editors, *Software Architecture - 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24-28, 2018, Proceedings*, volume 11048 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2018.
- [95] Valdemar Vicente Graciano Neto. Validating emergent behaviours in systems-of-systems through model transformations. In *SRC@ MoDELS*, 2016.

- [96] W. Tang, H. Feng, K. Hisazumi, and A. Fukuda. A Verification Method for Security and Safety of IoT Applications through DSM Language and Lustre. In *ACM International Conference Proceeding Series*, pages 166–170, 2020. doi: 10.1145/3388176.3388211.
- [97] Eclipse Modelling Framework (EMF), [Online]. Available: <https://www.eclipse.org/modeling/emf/> [Accessed June-2022].
- [98] ThingML open-source project, [Online]. Available: <https://github.com/TelluIoT/ThingML> [Accessed June-2022].
- [99] Blink specification, [Online]. Available: <https://github.com/TelluIoT/ThingML/blob/master/org.thingml.samples/src/main/thingml/samples/blink.thingml> [Accessed June-2022].
- [100] Paulo Eduardo Papotti, Antonio Francisco do Prado, Wanderley Lopes de Souza, Carlos Eduardo Cirilo, and Luís Ferreira Pires. A quantitative analysis of model-driven code generation through software experimentation. In *International Conference on Advanced Information Systems Engineering*, pages 321–337. Springer, 2013.
- [101] Tellu , [Online]. Available: <https://tellu.no/> [Accessed June-2022].
- [102] Brice Morin, Franck Fleurey, Knut Eilif Husa, and Olivier Barais. A Generative Middleware for Heterogeneous and Distributed Services. In *19th International ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE 2016, Venice, Italy, April 5-8, 2016*, pages 107–116. IEEE Computer Society, 2016.
- [103] Moez Krichen. *Contributions to model-based testing of dynamic and distributed real-time systems*. PhD thesis, École Nationale d’Ingénieurs de Sfax (Tunisie), 2018.
- [104] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1-2):35–132, 2000.
- [105] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and C Talcott. Maude manual (version 3.1). *SRI International University of Illinois at Urbana-Champaign*, 2020. URL <http://maude.lcc.uma.es/maude31-manual-html/maude-manual.html>.

- [106] José Meseguer. Rewriting logic and Maude: a wide-spectrum semantic framework for object-based distributed systems. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 89–117. Springer, 2000.
- [107] Alberto Verdejo and Narciso Martí-Oliet. Executable structural operational semantics in Maude. *The Journal of Logic and Algebraic Programming*, 67(1-2):226–293, 2006. Publisher: Elsevier.
- [108] Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. Programming and symbolic computation in Maude. *Journal of Logical and Algebraic Methods in Programming*, 110:100497, 2020.
- [109] José Meseguer. Membership algebra as a logical framework for equational specification. In *International Workshop on Algebraic Development Techniques*, pages 18–61. Springer, 1997.
- [110] Gordon D Plotkin. A structural approach to operational semantics. 1981. Publisher: Computer Science Department, Aarhus University Denmark.
- [111] Traian Florin Şerbănuţă, Grigore Roşu, and José Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207(2):305–340, 2009. Publisher: Elsevier.
- [112] Matthew Hennesy. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Wiley & Sons, 1990.
- [113] Gilles Kahn. Natural semantics. In *Annual symposium on theoretical aspects of computer science*, pages 22–39. Springer, 1987.
- [114] . HEADS-project, [Online]. Available: <https://github.com/HEADS-project> [Accessed June-2022].
- [115] . HEADS research project, [Online]. Available: <https://cordis.europa.eu/project/id/611337> [Accessed June-2022].
- [116] PingPong, [Online]. Available: [https://github.com/HEADS-project/training/tree/master/1.ThingML\\_Basics/2.PingPong/](https://github.com/HEADS-project/training/tree/master/1.ThingML_Basics/2.PingPong/) [Accessed June-2022].

- 
- [117] Eclipse modeling project (EMP), [Online]. Available: <http://www.eclipse.org/modeling/> [Accessed June-2022].
- [118] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. Sirius: A rapid development of DSM graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, pages 233–238. IEEE, 2014.
- [119] Lorenzo Addazi and Federico Ciccozzi. Blended graphical and textual modelling for UML profiles: A proof-of-concept implementation and experiment. *Journal of Systems and Software*, 175:110912, 2021.
- [120] José Meseguer and Grigore Roşu. The rewriting logic semantics project: A progress report. *Information and Computation*, 231:38–69, 2013.

---

# APPENDIX A

---

## A.1 Abstract syntax for the ThingML action language

### 1. Syntactic categories

$A$	$\in$	$Action$
$e$	$\in$	$Exp$
$be$	$\in$	$Bexp$
$v$	$\in$	$Val$
$bv$	$\in$	$BVal$
$vl$	$\in$	$ValList$
$x$	$\in$	$Var$
$bx$	$\in$	$Bvar$
$xl$	$\in$	$VarList$
$op$	$\in$	$Op$
$rop$	$\in$	$ROp$
$bop$	$\in$	$Bop$
$v$	$\in$	$Val$
$bv$	$\in$	$BVal$
$s$	$\in$	$Status$
$p$	$\in$	$PortId$
$m$	$\in$	$MsgId$
$msg$	$\in$	$Msg$

### 2. Definitions

$e$	$::=$	$v \mid x \mid e' \ op \ e'' \mid - \ e$
$op$	$::=$	$+ \mid - \mid * \mid / \mid rem$
$rop$	$::=$	$= \mid != \mid < \mid > \mid \leq \mid \geq$
$be$	$::=$	$bv \mid bx \mid Not \ be' \mid be \ bop \ be' \mid e \ rop \ e' \mid be = be' \mid be != be'$
$bop$	$::=$	$And \mid Or$
$bv$	$::=$	$true \mid false$
$vl$	$::=$	$v \mid bv \mid vl' , vl''$
$xl$	$::=$	$x \mid bx \mid xl' , xl''$
$msg$	$::=$	$m() \mid m(vl) \mid m(xl)$
$A$	$::=$	$x := e \mid A' ; A'' \mid If \ be \ Then \ A \ Else \ A' \mid If \ be \ Then \ A \mid While \ be \ Do \ A \mid x +$ $+ \mid x - - \mid p ! \ msg \mid goto \ ( \ s )$

## A.2 Evaluation semantics for expressions

<b>Val_Rule :</b>	$\frac{}{\langle v, st \rangle \Rightarrow_A v}$
<b>Var_Rule :</b>	$\frac{}{\langle x, st \rangle \Rightarrow_A st(x)}$
<b>VaList_Rule :</b>	$\frac{}{\langle vl, st \rangle \Rightarrow_A vl}$
<b>VarList_Rule :</b>	$\frac{}{\langle xl, st \rangle \Rightarrow_A st(xl)}$
<b>Op_Rule :</b>	$\frac{\langle e, st \rangle \Rightarrow_A v \quad \langle e', st \rangle \Rightarrow_A v'}{\langle e \text{ op } e', st \rangle \Rightarrow_A Ap(\text{op}, v, v')}$
<b>UMinus_Rule :</b>	$\frac{\langle e, st \rangle \Rightarrow_A v}{\langle - e, st \rangle \Rightarrow_A Ap(-, v)}$
-----	
<b>Bval_Rule :</b>	$\frac{}{\langle bv, st \rangle \Rightarrow_B bv}$
<b>Bvar_Rule :</b>	$\frac{}{\langle bx, st \rangle \Rightarrow_B st(bx)}$
<b>BOp_Rule :</b>	$\frac{\langle be, st \rangle \Rightarrow_B bv \quad \langle be', st \rangle \Rightarrow_B bv'}{\langle be \text{ bop } be', st \rangle \Rightarrow_B Ap(\text{bop}, bv, bv')}$
<b>Not_Rule :</b>	$\frac{\langle be, st \rangle \Rightarrow_B bv}{\langle \text{Not } be, st \rangle \Rightarrow_B Ap(\text{Not}, bv)}$
-----	
<b>ROp_Rule :</b>	$\frac{\langle e, st \rangle \Rightarrow_A v \quad \langle e', st \rangle \Rightarrow_A v'}{\langle e \text{ rop } e', st \rangle \Rightarrow_C Ap(\text{rop}, v, v')}$
<b>BEqu_Rule :</b>	$\frac{\langle be, st \rangle \Rightarrow_B bv \quad \langle be', st \rangle \Rightarrow_B bv'}{\langle be = be', st \rangle \Rightarrow_C Ap(=, bv, bv')}$
<b>BNEqu_Rule :</b>	$\frac{\langle be, st \rangle \Rightarrow_B bv \quad \langle be', st \rangle \Rightarrow_B bv'}{\langle be \neq be', st \rangle \Rightarrow_C Ap(\neq, bv, bv')}$

### A.3 Evaluation semantics for ThingML actions

<b>Eff_Rule :</b>	$\frac{\langle e, st \rangle \Rightarrow_A v}{\langle x := e, st \rangle \Rightarrow_D st [v / x]}$
<b>Dec_Rule :</b>	$\frac{\langle x := x - 1, st \rangle \Rightarrow_A st'}{\langle x --, st \rangle \Rightarrow_D st'}$
<b>Inc_Rule :</b>	$\frac{\langle x := x + 1, st \rangle \Rightarrow_A st'}{\langle x ++, st \rangle \Rightarrow_D st'}$
<b>Action_Rule :</b>	$\frac{\langle A, st \rangle \Rightarrow_D st' \quad \langle A', st' \rangle \Rightarrow_D st''}{\langle A ; A', st \rangle \Rightarrow_D st''}$
<b>Goto_Rule :</b>	$\overline{\langle goto (s) , st \rangle \Rightarrow_D st[s]}$
<b>Send_Rule1 :</b>	$\overline{\langle p ! msg (vl) , st \rangle \Rightarrow_D st[msg(vl)   - p]}$
<b>Send_Rule2 :</b>	$\frac{\langle xl , st \rangle \Rightarrow_A vl}{\overline{\langle p ! msg (xl) , st \rangle \Rightarrow_D st[msg(vl)   - p]}}$
<b>Send_Rule3 :</b>	$\overline{\langle p ! msg () , st \rangle \Rightarrow_D st[msg()   - p]}$
<b>If_Then_Rule1 :</b>	$\frac{\langle be, st \rangle \Rightarrow_B true \quad \langle A, st \rangle \Rightarrow_D st'}{\overline{\langle If be Then A Else A', st \rangle \Rightarrow_D st'}}$
<b>If_Then_Rule2 :</b>	$\frac{\langle be, st \rangle \Rightarrow_B false \quad \langle A', st \rangle \Rightarrow_A st'}{\overline{\langle If be Then A Else A', st \rangle \Rightarrow_A st'}}$
<b>If_Rule1 :</b>	$\frac{\langle be, st \rangle \Rightarrow_B true \quad \langle A, st \rangle \Rightarrow_D st'}{\overline{\langle If be Then A, st \rangle \Rightarrow_D st'}}$
<b>If_Rule2 :</b>	$\frac{\langle be, st \rangle \Rightarrow_B false}{\overline{\langle If be Then A, st \rangle \Rightarrow_D st}}$
<b>While_Rule1 :</b>	$\frac{\langle be, st \rangle \Rightarrow_B true \quad \langle A ; While be Do A, st \rangle \Rightarrow_D st'}{\overline{\langle While be Do A, st \rangle \Rightarrow_D st'}}$
<b>While_Rule2 :</b>	$\frac{\langle be, st \rangle \Rightarrow_B false}{\overline{\langle While be Do A, st \rangle \Rightarrow_D st}}$



---

# APPENDIX B

---

## B.1 The THINGML-CONVERSION module

The ThingML data types are mapped to Maude predefined sorts. For this, we define the module THINGML-CONVERSION, which imports the predefined module CONVERSION and renames the appropriate operators.

```
1  fmod THINGML-CONVERSION is
2    pr CONVERSION
3      * (--- Renamings in FLOAT
4        sort Float to Float ,
5        sort String to STRING ,
6        sort Char to CHAR , sort FindResult to FindResult ,
7        op abs : Float -> Float to absF ,
8        op notFound : -> FindResult to NotFound ,
9        op char : Nat ~> Char to CHAR ,
10       op _ : Float -> Float to -F_ ,
11       op +_ : Float Float -> Float to +F_ ,
12       op -_ : Float Float -> Float to -F_ ,
13       op /_ : Float Float -> Float to /F_ ,
14       op *_ : Float Float -> Float to *F_ ,
15
16       op min : Float Float -> Float to minF ,
17       op max : Float Float -> Float to maxF ,
18       op _rem_ : Float Float -> Float to _remF_ ,
19       op ^_ : Float Float -> Float to ^F_ ,
20       op <_ : Float Float -> Bool to <F_ ,
21       op <=_ : Float Float -> Bool to <=F_ ,
22       op >_ : Float Float -> Bool to >F_ ,
23       op >=_ : Float Float -> Bool to >=F_ ,
24       op floor : Float -> Float to floorF ,
25       op ceiling : Float -> Float to ceilingF ,
26
27       --- Renamings in STRING
28       op <_ : String String -> Bool to _ltt_ ,
29       op <=_ : String String -> Bool to _leq_ ,
30       op >_ : String String -> Bool to _gtt_ ,
31       op >=_ : String String -> Bool to _geq_ ,
32       op ++_ : String String -> String to ++_ ,
33       op float : String ~> Float to string2float ) .
34  endfm
```

## B.2 The THINGML-EXP-SYNTAX module

In the functional module THINGML-EXP-SYNTAX, we have defined the syntax of arithmetic and logical expressions de the ThingML language. We have declared new arithmetic operators corresponding to Maude's predefined operators (with the same properties).

```

1  fmod THINGML-EXP-SYNTAX is
2    including THINGML-CONVERSION .
3
4    sorts Exp BExp Var BVar Op BOp ROp ExpList ValList VarList Value Variable .
5    subsorts Bool Int Nat FLoat SString < Value .
6    subsorts Var BVar < Variable .
7    subsort Var < Exp .
8    subsorts Int Nat FLoat SString < Exp .
9    subsort BVar < BExp .
10   subsort Bool < BExp .
11
12   --- List of expressions
13   subsort Exp BExp < ExpList .
14   op emptyExpList : -> ExpList .
15   op _,_ : ExpList ExpList -> ExpList [assoc prec 30] .
16
17   --- List of values
18   subsort Value < ValList .
19   op emptyValList : -> ValList .
20   op _,_ : ValList ValList -> ValList [assoc prec 30] .
21   subsort ValList < ExpList .
22
23   --- List of variables
24   subsort Variable < VarList .
25   op emptyVarList : -> VarList .
26   op _,_ : VarList VarList -> VarList [assoc prec 30] .
27   subsort VarList < ExpList .
28
29   --- These operators are used by the function Ap
30   ops .+ .* .- ./ .rem : -> Op .
31   ops .= .!= .< .> .<= .>= : -> ROp .
32   ops And Or : -> BOp .
33
34   --- "And", "Or", and "not" operations corresponding to the "and" and "or" operations of BOOL
35   module
36   --- They keep the same properties
37   op _And_ : BExp BExp -> BExp [ctor assoc comm prec 55] .
38   op _Or_ : BExp BExp -> BExp [ctor assoc comm prec 59] .
39   op Not_ : BExp -> BExp [ctor prec 53] .
40
41   --- New arithmetic operators corresponding to Maude's predefined operators (with the same
42   properties)
43   op _-_ : Exp -> Exp [ctor prec 53] .
44   op _+_ : Exp Exp -> Exp [ctor gather (E e) prec 33] .
45   op _-_- : Exp Exp -> Exp [ctor gather (E e) prec 33] .
46   op _*_ : Exp Exp -> Exp [ctor gather (E e) prec 31] .
47   op _./_ : Exp Exp -> Exp [ctor gather (E e) prec 31] .
48   op _rem_ : Exp Exp -> Exp [ctor gather (E e) prec 31] .
49
50   --- New relational operators corresponding to Maude's predefined operators (with the same
51   properties)
52   op _=_ : Exp Exp -> BExp [ctor prec 51 poly (1 2)] .
53   op _!=_ : Exp Exp -> BExp [ctor prec 51 poly (1 2)] .
54   op _<_ : Exp Exp -> BExp [ctor prec 37] .
55   op _>_ : Exp Exp -> BExp [ctor prec 37] .
56   op _<=_ : Exp Exp -> BExp [ctor prec 37] .
57   op _>=_ : Exp Exp -> BExp [ctor prec 37] .
58
59   endfm

```

### B.3 The THINGML-SYNTAX module

The THINGML-SYNTAX module defines the syntax of the ThingML constructs. It defines the syntax of Things, messages, ports, states, actions, events and configurations (instances and connectors).

```

1  mod THINGML-SYNTAX is
2  including CONFIGURATION .
3  including THINGML-EXP-SYNTAX .
4
5  --- Thing
6  sorts ThingId Statement Store Action .
7  subsort ThingId < Cid . --- Cid : Class identifiers
8  subsorts Bool Int Nat FFloat SString < Statement .
9  subsort Store < Statement .
10 --- Declaration of the environment attribute
11 op environment:_ : Statement -> Attribute [ctor gather (&)] .
12 op <,_> : Action Store -> Statement .
13 op <,_> : Exp Store -> Statement .
14 op <,_> : BExp Store -> Statement [ditto] .
15 --- Message
16 sorts MsgId MsgSet .
17 op `(`) : MsgId -> Msg [ctor] .
18 op `(`) : MsgId ExpList -> Msg [ctor] .
19 subsort Msg < MsgSet .
20 op noMsg : -> MsgSet [ctor] .
21 op ;_ : MsgSet MsgSet -> MsgSet [ctor comm assoc id: noMsg] .
22 op parmsg : MsgId -> VarList .
23
24 --- Ports
25 sorts Port PortId PortName .
26 subsort Port < Cid . --- Cid : Class identifiers
27
28 op ._ : InstanceId PortId -> PortName [ctor] .
29 subsort PortName < Oid . --- Oid : Object identifiers
30 --- Declaration of port classes
31 ops ProvidedPort RequiredPort InternalPort : -> Port .
32 --- class ProvidedPort | buffer: MsgSet .
33 --- class RequiredPort | buffer: MsgSet .
34 --- class InternalPort | buffer: MsgSet .
35 --- Declaration of the buffer attribute
36 op buffer:_ : MsgSet -> Attribute [ctor gather (&)] .
37
38 --- State
39 sorts StateId CompositeStateId AtomicStateId .
40 subsorts CompositeStateId AtomicStateId < StateId .
41 sort Status .
42 subsorts AtomicStateId < Status .
43 op `(`) : CompositeStateId Status -> Status [ctor] .
44 op noState : -> Status [ctor] .
45 op ||_ : Status Status -> Status [ctor assoc id: noState] .
46 ---sorts SessionId RegionId .
47 ---subsort SessionId RegionId < CompositeStateId .
48
49 --- Action language
50 sort BlockAction .
51 op noAction : -> Action .
52 op _:=_ : Var Exp -> Action [prec 39] .
53 op ;_ : Action Action -> Action [assoc prec 40] .
54 op If_Then_Else_ : BExp Action Action -> Action [prec 50] .
55 op If_Then_ : BExp Action -> Action [prec 50] .
56 op While_Do_ : BExp Action -> Action [prec 60] .
57 op !_ : PortId Msg -> Action [prec 60] .
58 op _++ : Var -> Action [prec 39] .
59 op _-- : Var -> Action [prec 39] .
60 op goto`(`) : Status -> Action [prec 39] .
61 op print`(`) : SString -> Action [prec 39] .
62 op do_end : Action -> BlockAction [prec 39] .
63
64 --- Event
65 sort Event .
66 op ?_ : PortId MsgId -> Event .
67 op noEvent : -> Event .
68 --- Configuration
69 sorts InstanceId .
70 subsort InstanceId < Oid . --- Oid : Object identifier
71 sort Connector .
72 subsort Connector < Object .
73 op connector | client:_--> server:_ : Object Object -> Connector [ctor] .
74 endm

```

## B.4 The THINGML-STORE module

The THINGML-STORE imports the THINGML-SYNTAX module and implements the Store concept in Maude. It considers all the information of the thing, which allows for storing and managing the thing's status, messages, properties, parameters, and events. The THINGML-STORE module includes operators and equations that ensure different functions such as reading, writing, and modifying variables (arithmetic, Boolean, and string), status, messages, and events.

```

1  mod THINGML-STORE is
2    including THINGML-SYNTAX .
3
4    op mt : -> Store . --- empty store
5    --- SS = Singleton Store
6    --- `(_=_)` : use to associate a value to their variable
7    op `(_=_)`      : Variable Value -> Store [prec 20] . --- SS of a variable (For properties and
   parameters)
8    op `(_)`       : Event          -> Store [prec 20] . --- SS of a event
9    op `(_Via_)`   : Msg PortId     -> Store [prec 20] . --- SS of a message
10   op `(status:_)` : Status          -> Store [prec 20] . --- SS of the status
11   op _;_ : Store Store -> Store [assoc id: mt prec 30] . --- Union of SSs
12   --- The next operators are used to add or update the SEs of the:
13   op _[_/_`] : Store Value Variable -> Store [prec 35] . --- variable
14   op _[_/_`] : Store VaList VarList -> Store [prec 35] . --- list of variable
15   op _[_|_] : Store Msg PortId     -> Store [prec 35] . --- message
16   op _[_] : Store Status            -> Store [prec 35] . --- status
17   --- `(_)` : returns a value (or a list) of the variable (or a list)
18   op _`(_)` : Store Variable -> Value .
19   op _`(_)` : Store VarList  -> VaList .
20   --- The next part of the module represents the implementation of the operations defined above.
21   op remove : Store Variable -> Store .
22   var m : Msg . var v : Value . vars s s' : Status .
23   var nl : VaList . var st : Store . var p : PortId .
24   var xl : VarList . var ev : Event . vars x x' : Variable .
25   eq st [v / x] = remove(st, x) ; ( x = v ) .
26   eq remove(mt, x) = mt .
27   ceq remove((x = v) ; st, x') = st if x == x' .
28   ceq remove((x = v) ; st, x') = (x = v) ; remove(st,x') if x /= x' .
29   --- eq remove (( x = v ) ; st, x) = st .
30   --- eq remove((x = v) ; st, x') = (x = v) ; remove(st,x') [owise] .
31   eq st [v, nl / x, xl] = (st[v / x]) [nl / xl] .
32   eq st [emptyVarList / emptyVaList] = st .
33
34   eq st [ m | - p ] = st ; ( m Via p ) .
35   eq ((m Via p) ; st)(x') = st(x') .
36   eq remove((m Via p) ; st, x') = (m Via p) ; remove(st,x') .
37   eq (status: s') ; st [s] = ( status: s' || s ) ; st .
38   eq ((status: s) ; st)(x') = st(x') .
39   eq ((ev) ; st)(x') = st(x') .
40   eq remove((status: s) ; st, x') = (status: s) ; remove(st,x') .
41
42   ceq ( ( x = v ) ; st)(x') = v if x == x' .
43   ceq ( ( x = v ) ; st)(x') = st(x') if x /= x' .
44   --- eq ( ( x = v ) ; st)(x) = v .
45   --- eq ( ( x = v ) ; st)(x') = st(x') [owise] .
46
47   eq st(emptyVarList) = emptyVaList .
48   eq ( (status: s) ; st)(xl) = st(xl) .
49   eq ( (ev) ; st)(xl) = st(xl) .
50   eq remove((ev) ; st, x') = (ev) ; remove(st,x') .
51   eq ( (m Via p) ; st)(xl) = st(xl) .
52   eq st(x , xl) = ( (st(x)) , (st(xl))) .
53   endm

```

## B.5 The THINGML-AP module

The functional module AP defines an operation Ap that enables the application of a binary operator to two already evaluated arguments. It allows switching between the defined operations and the corresponding Maude's predefined operations. The latter enables the execution of the arithmetic and logical operations concretely.

```

1  fmod THINGML-AP is
2    including THINGML-EXP-SYNTAX .
3
4    op Ap : Op Value Value -> Value .
5    op Ap : ROp Value Value -> Bool .
6    op Ap : BOp Bool Bool -> Bool .
7    vars bv bv' : Bool .
8
9    --- ---- Int & Nat
10   ---
11   eq Ap(+,v:Int,v':Int)      = v:Int + v':Int      .
12   eq Ap(*,v:Int,v':Int)      = v:Int * v':Int      .
13   eq Ap(-,v:Int,v':Int)      = v:Int - v':Int      .
14   eq Ap(.rem,v:Int,v':Int)   = v:Int rem v':Int    .
15   eq Ap(/,v:Rat,v':Rat)      = v:Rat / v':Rat     .
16   eq Ap(<,v:Int,v':Int)      = v:Int < v':Int     .
17   eq Ap(>,v:Int,v':Int)      = v:Int > v':Int     .
18   eq Ap(<=,v:Int,v':Int)     = v:Int <= v':Int    .
19   eq Ap(>=,v:Int,v':Int)     = v:Int >= v':Int    .
20   ---
21   --- ---- FFloat
22   ---
23   eq Ap(+,v:FFloat,v':FFloat) = v:FFloat +F v':FFloat .
24   eq Ap(*,v:FFloat,v':FFloat) = v:FFloat *F v':FFloat .
25   eq Ap(-,v:FFloat,v':FFloat) = v:FFloat -F v':FFloat .
26   eq Ap(.rem,v:FFloat,v':FFloat) = v:FFloat remF v':FFloat .
27   eq Ap(/,v:FFloat,v':FFloat) = v:FFloat /F v':FFloat .
28   eq Ap(<,v:FFloat,v':FFloat) = v:FFloat <F v':FFloat .
29   eq Ap(>,v:FFloat,v':FFloat) = v:FFloat >F v':FFloat .
30   eq Ap(<=,v:FFloat,v':FFloat) = v:FFloat <=F v':FFloat .
31   eq Ap(>=,v:FFloat,v':FFloat) = v:FFloat >=F v':FFloat .
32   ---
33   --- ---- String
34   ---
35   eq Ap(+,v:String,v':String) = v:String ++ v':String .
36   eq Ap(<,v:String,v':String) = v:String ltt v':String .
37   eq Ap(>,v:String,v':String) = v:String gtt v':String .
38   eq Ap(<=,v:String,v':String) = v:String leq v':String .
39   eq Ap(>=,v:String,v':String) = v:String geq v':String .
40   ---
41   --- ---- All data types
42   ---
43   eq Ap(=,v:Value,v':Value)    = v:Value == v':Value .
44   eq Ap(!=,v:Value,v':Value)   = v:Value /= v':Value .
45   ---
46   --- ---- Bool
47   ---
48   eq Ap(And,bv,bv')            = bv and bv' .
49   eq Ap(Or,bv,bv')            = bv or bv' .
50   ---
51   --- ---- Conversion RAt2FFloat
52   ---
53   var o : Op . var ro : ROp .
54   eq Ap(o,v:Rat,v':FFloat)     = Ap(o,float(v:Rat),v':FFloat) .
55   eq Ap(o,v:FFloat,v':Rat)     = Ap(o,v:FFloat,float(v':Rat)) .
56   eq Ap(ro,v:Rat,v':FFloat)    = Ap(ro,float(v:Rat),v':FFloat) .
57   eq Ap(ro,v:FFloat,v':Rat)    = Ap(ro,v:FFloat,float(v':Rat)) .
58   endfm

```

## B.6 The THINGML-EXP-EVALUATION module

The THINGML-EXP-EVALUATION module implements the evaluation semantics of the arithmetic and logical expression language.

```

1  mod THINGML-EXP-EVALUATION is
2  including THINGML-STORE .
3  including THINGML-AP .
4
5  var st      : Store . var x : Variable . var bx     : BVar .
6  vars e e'   : Exp . vars be be' : BExp . vars v v'  : Value .
7  vars bv bv' : Bool . var xl : VarList .
8  --- Value evaluation rule (for all value types = Bool Int Nat Float String)
9  rl [Value-R] : < v, st > => v .
10 --- Variable evaluation rule (for all variable types = Bool Int Nat Float String)
11 rl [Variable-R] : < x, st > => st(x) .
12 --- Evaluation rule of variable list
13 rl [VarList-R] : < xl, st > => st(xl) .
14
15 --- Evaluation rules for arithmetic operations ( + , - , * , / and rem )
16 crl [Add-R] : < e .+ e', st > => Ap(+,v,v') if < e, st > => v /\ < e', st > => v' .
17 crl [Min-R] : < e .- e', st > => Ap(-,v,v') if < e, st > => v /\ < e', st > => v' .
18 crl [Mul-R] : < e .* e', st > => Ap(*,v,v') if < e, st > => v /\ < e', st > => v' .
19 crl [Div-R] : < e ./ e', st > => Ap(/,v,v') if < e, st > => v /\ < e', st > => v' .
20 crl [Rem-R] : < e .rem e', st > => Ap(.rem,v,v') if < e, st > => v /\ < e', st > => v' .
21
22 --- Evaluation rule for the - (unary minus) operator
23 crl [UMinus-R1] : < .- e, st > => - v if < e, st > => v /\ ( v :: Int ) .
24 crl [UMinus-R2] : < .- e, st > => -F v if < e, st > => v /\ ( v :: FFloat ) .
25
26 --- Evaluation rules for relational operations ( ==, !=, <, >, <=, >=)
27 crl [Equ-R] : < e .== e', st > => Ap(==,v,v') if < e, st > => v /\ < e', st > => v' .
28 crl [NEq-R] : < e .!= e', st > => Ap(!=,v,v') if < e, st > => v /\ < e', st > => v' .
29 crl [Low-R] : < e .< e', st > => Ap(<,v,v') if < e, st > => v /\ < e', st > => v' .
30 crl [Gre-R] : < e .> e', st > => Ap(>,v,v') if < e, st > => v /\ < e', st > => v' .
31 crl [LEq-R] : < e .<= e', st > => Ap(<=,v,v') if < e, st > => v /\ < e', st > => v' .
32 crl [GEq-R] : < e .>= e', st > => Ap(>=,v,v') if < e, st > => v /\ < e', st > => v' .
33
34 --- Evaluation rules for boolean operations ( and, or, not, ==, !=)
35 crl [And-R] : < be And be', st > => Ap(And, bv, bv') if < be, st > => bv /\ < be', st > => bv' .
36 crl [Or-R] : < be Or be', st > => Ap(Or, bv, bv') if < be, st > => bv /\ < be', st > => bv' .
37 crl [EquB-R] : < be .== be', st > => Ap(==, bv, bv') if < be, st > => bv /\ < be', st > => bv' .
38 crl [NEquB-R] : < be .!= be', st > => Ap(!=, bv, bv') if < be, st > => bv /\ < be', st > => bv' .
39 crl [Not-R] : < Not be, st > => not bv if < be, st > => bv .
40 endm

```

## B.7 The THINGML-ACTION-SEMANTICS module

The THINGML-ACTION-SEMANTICS module imports the THINGML-EXP-EVALUATION module and implements the evaluation semantics of the action language.

```

1  mod THINGML-ACTION-SEMANTICS is
2  including THINGML-EXP-EVALUATION .
3
4  var x : Var . var v : Value . var e : Exp . var be : BExp .
5  var xl : VarList . var vl : ValList . var m : MsgId . var p : PortId .
6  var T : ThingId . var I : InstanceId . vars A A' : Action . var s : Status .
7  vars st st' st'' : Store .
8

```

```

9      --- Evaluation semantics for actions
10     crl [Eff-R] : < x := e, st > => st[v / x] if < e, st > => v .
11     crl [Dec-R] : < x --, st > => st' if < x := x .- 1, st > => st' .
12     crl [Inc-R ] : < x ++, st > => st' if < x := x .+ 1, st > => st' .
13     crl [Act-R] : < A ; A', st > => st'' if < A, st > => st' /\ < A', st' > => st'' .
14     crl [Send-R1] : < p ! m(xl), st > => st [ m(vl) |- p ] if < xl, st > => vl .
15
16     rl [Send-R2] : < p ! m(vl), st > => st [ m(vl) |- p ] .
17     rl [Send-R3] : < p ! m(), st > => st [ m() |- p ] .
18     rl [GoTo-R] : < goto(s), st > => st[s] .
19     rl [PrIntR] : < print(v), st > => st .
20     rl [noAction] : < I : T | environment: st > => < I : T | environment: < noAction, st > > .
21
22     crl [If-Then-R1] : < If be Then A Else A', st > => st' if < be, st > => true /\ < A, st > => st' .
23     crl [If-Then-R2] : < If be Then A Else A', st > => st' if < be, st > => false /\ < A', st > => st' .
24     crl [If-R1] : < If be Then A, st > => st' if < be, st > => true /\ < A, st > => st' .
25     crl [If-R2] : < If be Then A, st > => st if < be, st > => false .
26     crl [While-R1] : < While be Do A, st > => st if < be, st > => false .
27     crl [While-R2] : < While be Do A, st > => st' if < be, st > => true
28                                     /\ < A ; (While be Do A), st > => st' .
29     endm

```

## B.8 The THINGML-MSG-SEMANTICS module

The THINGML-MSG-SEMANTICS module implements the routing of messages between the sending instance and the receiving instance.

```

1  mod THINGML-MSG-SEMANTICS is
2    including THINGML-ACTION-SEMANTICS .
3
4    var T : ThingId . var I : InstanceId . var A : Action .
5    var Po : PortId . var P : PortName . vars st st' : Store .
6    var MS : MsgSet . var O : Object . var M : Msg .
7    var msgId : MsgId . var vl : VaList . var ATTS : AttributeSet .
8
9    --- To move the messages from instance environment to the connector (client buffer)
10   rl [Env2CliBuff] : < I : T | environment: < A, st ; (M Via Po) ; st' > >
11                   connector | client: < I . Po : RequiredPort | buffer: MS > --> server: O
12                   => < I : T | environment: < A, st ; st' > >
13                   connector | client: < I . Po : RequiredPort | buffer: (MS ; M) > --> server: O .
14   --- To move the messages from instance environment to the connector (server buffer)
15   rl [Env2SerBuff] : < I : T | environment: < A, st ; (M Via Po) ; st' > >
16                   connector | client: O --> server: < I . Po : ProvidedPort | buffer: MS >
17                   => < I : T | environment: < A, st ; st' > >
18                   connector | client: O --> server: < I . Po : ProvidedPort | buffer: (MS ; M) > .
19   --- To move the messages from connector ( server buffer) to instance environment (message with
20   parameters)
21   rl [BuffSer2EnvR1] : < I : T | environment: < A, st > >
22                   connector | client: < I . Po : RequiredPort | ATTS >
23                   --> server: < P : ProvidedPort | buffer: ((msgId (vl)) ; MS) >
24                   => < I : T | environment: < A, (st [ vl / (parmsg(msgId)) ]) ; (Po ? msgId) > >
25                   connector | client: < I . Po : RequiredPort | ATTS >
26                   --> server: < P : ProvidedPort | buffer: MS > .
27   --- To move the messages from connector ( client buffer) to instance environment (message with
28   parameters)
29   rl [BuffCli2EnvR1] : < I : T | environment: < A, st > >
30                   connector | client: < P : RequiredPort | buffer: ((msgId (vl)) ; MS) >
31                   --> server: < I . Po : ProvidedPort | ATTS >
32                   => < I : T | environment: < A, (st [ vl / (parmsg(msgId)) ]) ; (Po ? msgId) > >
33                   connector | client: < P : RequiredPort | buffer: MS >
34                   --> server: < I . Po : ProvidedPort | ATTS > .
35   --- To move the messages from connector ( server buffer) to instance environment (message without
36   parameters)
37   rl [BuffSer2EnvR2] : < I : T | environment: < A, st > >
38                   connector | client: < I . Po : RequiredPort | ATTS >
39                   --> server: < P : ProvidedPort | buffer: ((msgId ()) ; MS) >

```

```
37         => < I : T | environment: < A , st ; ( Po ? msgId ) > >
38         connector | client: < I . Po : RequiredPort | ATTS >
39         -->         server: < P           : ProvidedPort | buffer: MS > .
40 --- To move the messages from connector ( client buffer) to instance environment (message without
41 parameters)
41 rl [BuffCli2EnvR2] : < I : T | environment: < A , st > >
42         connector | client: < P           : RequiredPort | buffer: ((msgId ()) ; MS) >
43         -->         server: < I . Po : ProvidedPort | ATTS >
44         => < I : T | environment: < A , st ; ( Po ? msgId ) > >
45         connector | client: < P           : RequiredPort | buffer: MS >
46         -->         server: < I . Po : ProvidedPort | ATTS > .
47 endm
```



## ملخص :

أنظمة إنترنت الأشياء (IoT) عبارة عن مجموعات معقدة من المكونات التي تتعاون لتحقيق أهداف مشتركة. تتركز هذه المكونات على تقنيات مختلفة غير متجانسة وتتواصل مع بعضها البعض باستخدام بروتوكولات إتصال مختلفة. هذا التباين يجعل تصميم وتطوير تطبيقات إنترنت الأشياء مشكلة صعبة. تم اقتراح مناهج متنوعة تعتمد على الهندسة الموجهة بالنموذج (MDE) للتغلب على هذه المشكلة الرئيسية وذلك باستخدام لغات نمذجة مناسبة. ThingML هو إمتداد للغة UML واعد لنمذجة تطبيقات إنترنت الأشياء يهدف إلى مواجهة تحديات عدم التجانس. ومع ذلك ، لا يحتوي ThingML على دلالات صارمة ، مما يجعله غير مناسب للتحقق الدقيق وتحليل تصميمات النظام. كما أنه يفتقر كذلك إلى الأدوات اللازمة لاختبار الكود الذي تم إنشاؤه قبل استعماله في أجهزة إنترنت الأشياء. في هذه الأطروحة ، نقترح نهجاً دقيقاً قائماً على MDE لتحديد دلالات دقيقة للغة ThingML باستخدام منطق إعادة الكتابة ولغتها Maude . بالإضافة لذلك، قمنا بتطوير محرر نصي - رسومي للغة ThingML ونقدم نهجاً قائماً على المحاكاة لاختبار كود المصدر الذي تم إنشاؤه بواسطة خاصية إنشاء الكود لـ ThingML . الأساليب المقترحة تم توضيحها بواسطة دراسات حالة.

**كلمات مفتاحية :** إنترنت الأشياء، التحقق الدقيق، منطق إعادة الكتابة، لغة مود، الهندسة النموذجية.

**Abstract:** Internet of Things (IoT) systems are complex assemblies of components that collaborate to achieve common goals. These components are based on different heterogeneous technologies and communicate with each other using various communication protocols. This heterogeneity makes the design and development of IoT applications a challenging issue. Diverse approaches based on Model-Driven Engineering (MDE) have been proposed to overcome this major issue using suitable modeling languages. ThingML is a promising UML profile for modeling IoT applications that aims to address the challenges of heterogeneity. However, ThingML does not have rigorous semantics, which makes it unsuitable for formal verification and analysis of system designs. It also lacks tools to test the generated code before deploying it in IoT devices. In this thesis, we propose an MDE-based formal approach to define a formal semantics for ThingML language using Rewriting Logic and its language Maude. In addition, we develop a hybrid textual-graphical editor for the ThingML language and we present a simulation-based approach to test the source code generated by the ThingML code generation framework. The proposed approaches are illustrated through case studies.

**Key Words :** *Internet of Things, Formal verification, Rewriting logic, Maude language, Model Driven Engineering.*

**Résumé:** Les systèmes de l'Internet des objets (IoT) sont des assemblages complexes de composants qui collaborent pour atteindre des objectifs communs. Ces composants sont basés sur différentes technologies hétérogènes et communiquent entre eux à l'aide de divers protocoles de communication. Cette hétérogénéité fait de la conception et du développement d'applications IoT un véritable défi. Diverses approches basées sur l'Ingénierie Dirigée par les Modèles (IDM) ont été proposées pour surmonter ce problème majeur en utilisant des langages de modélisation appropriés. ThingML est un profil UML prometteur pour la modélisation des applications IoT qui vise à relever les défis de l'hétérogénéité. Cependant, ThingML ne possède pas de sémantique rigoureuse, ce qui le rend inadapté à la vérification et à l'analyse formelles des conceptions de systèmes. Il manque également des outils pour tester le code généré avant de le déployer dans les dispositifs IoT. Dans cette thèse, nous proposons une approche formelle basée sur l'IDM pour définir une sémantique formelle pour le langage ThingML en utilisant la logique de réécriture et son langage Maude. En outre, nous développons un éditeur hybride textuel graphique pour le langage ThingML et nous présentons une approche basée sur la simulation pour tester le code source généré par le cadre de génération de code ThingML. Les approches proposées sont illustrées à travers des études de cas.

**Mots clés :** *Internet des Objets, Vérification formelle, Logique de réécriture, Langage Maude, Ingénierie Dirigée par les Modèles.*