



T.R.

EGE UNIVERSITY

Graduate School of Applied and Natural
Science



PLATFORM INDEPENDENT MODELING OF WIRELESS SENSOR NETWORK PROGRAMS

Master Thesis

Burak KARADUMAN

International Computer Institute

Information Technologies

İzmir

2020

T.R.

EGE UNIVERSITY

Graduate School of Applied and Natural Science

**PLATFORM INDEPENDENT MODELING OF
WIRELESS SENSOR NETWORK PROGRAMS**

Burak KARADUMAN

Supervisor: Assoc. Prof. Dr. Orhan DAĞDEVİREN

International Computer Institute

Information Technologies

İzmir

2020

Burak KARADUMAN tarafından yüksek lisans tezi olarak sunulan "Platform Independent Modeling of Wireless Sensor Network Programs" başlıklı bu çalışma EÜ Lisansüstü Eğitim ve Öğretim Yönetmeliği ile EÜ Fen Bilimleri Enstitüsü Eğitim ve Öğretim Yönergesi'nin ilgili hükümleri uyarınca tarafımızdan değerlendirilerek savunmaya değer bulunmuş ve 04.02.2020 tarihinde yapılan tez savunma sınavında aday oybirliği/~~oyçokluğu~~ ile başarılı bulunmuştur.

Jüri Üyeleri:

İmza

Jüri Başkanı :Doç.Dr. Orhan DAĞDEVİREN

Raportör Üye :Doç.Dr. Geylani KARDAŞ

Üye :Dr.Öğr.Üyesi Uras TOS

.....

.....

.....

EGE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ

ETİK KURALLARA UYGUNLUK BEYANI

EÜ Lisansüstü Eğitim ve Öğretim Yönetmeliğinin ilgili hükümleri uyarınca Yüksek Lisans Tezi olarak sunduğum “Platform Independent Modeling of Wireless Sensor Network Programs” başlıklı bu tezin kendi çalışmam olduğunu, sunduğum tüm sonuç, doküman, bilgi ve belgeleri bizzat ve bu tez çalışması kapsamında elde ettiğimi, bu tez çalışmasıyla elde edilmeyen bütün bilgi ve yorumlara atıf yaptığımı ve bunları kaynaklar listesinde usulüne uygun olarak verdiğimi, tez çalışması ve yazımı sırasında patent ve telif haklarını ihlal edici bir davranışımın olmadığını, bu tezin herhangi bir bölümünü bu üniversite veya diğer bir üniversitede başka bir tez çalışması içinde sunmadığımı, bu tezin planlanmasından yazımına kadar bütün safhalarda bilimsel etik kurallarına uygun olarak davrandığımı ve aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul edeceğimi beyan ederim.

04. / 02. / 2020

Burak KARADUMAN



ÖZET**KABLOSUZ DUYARGA AĞLARIN PROGRAMLARI İÇİN PLATFORMDAN BAĞIMSIZ MODELLEME ORTAMI**

KARADUMAN, Burak

Yüksek Lisans Tezi, Uluslararası Bilgisayar

Danışman: Assoc. Doç. Dr. Orhan DAĞDEVİREN

Şubat 2020, 166 pages

Bu tez çalışmasında, Kablosuz Duyarga Ağları (KSA)'ndaki yazılım karmaşıklığını gidermek amacıyla Model-Güdümlü Geliştirme yaklaşımı çalışılmıştır. Dahası, KSA seviyesi bir ağ geçiti aracılığı ile internete açılmış ve sistemin bir kısmında Nesnelerin İnterneti uyumlu cihazlar kullanılarak, hem Kablosuz Duyarga Ağları hemde Nesnelerin İnterneti paradigmaları için modelleme ortamı sunulmuştur.

İlk aşamada, bir Model-Güdümlü Geliştirme (MGG) yaklaşımı sağlamak için, Contiki ve RIOT işletim sistemleri kullanılarak bir sistem tasarlanmış ve geliştirilmiştir. Sistemin geliştirilmesi ve tasarlanmasında, karmaşık ve zaman harcayan kısımlar analiz edilmiştir.

İkinci aşamada, iki işletim sisteminin ortak özellikleri belirlenmiştir. Bu ortak özelliklere bağlı olarak, hem RIOT hemde Contiki işletim sistemleri için bir Platforma-Özgü Model (PÖM) oluşturmak için üst-model ve bir modelleme ortamı tasarlanmıştır. PÖM kullanılarak, tasarımcı Contiki ve RIOT ortamları için kod üretebilir ve Petri-net ile tasarlanan istemin ayrık olay analizlerini yapabilir.

Son aşama, bu iki işletim sistemi için Platformdan-Bağımsız Model (PBM) oluşturmaktır. PBM, PÖM seviyesi için hızlı tasarım sağlamak için tasarlanmıştır. Ortak özellikler ve nitelikler PIM seviyesinde birleştirilmiştir. Bu sayede, model elemanları ve nitelikleri, PBM'den PÖM'e dönüşüm yaparken tasarımcıya daha fazla soyutlama sağlamak için dönüşüm kurallarında otomatik olarak üretilmek üzere tanımlanmıştır.

Anahtar Kelimeler: Model-Güdümlü Geliştirme, Kablosuz Duyarga Ağları, Nesnelerin İnterneti, Contiki İşletim Sistemi, RIOT İşletim Sistemi, Petri-net.

ABSTRACT**PLATFORM INDEPENDENT MODELING OF WIRELESS SENSOR NETWORK PROGRAMS**

KARADUMAN, Burak

MSc in International Computer

Supervisor: Assoc. Prof. Dr. Orhan DAĞDEVİREN

February 2020, 166 pages

In this thesis study, a Model-Driven Development (MDD) approach is studied to deal with software complexity in Wireless Sensor Network (WSN). Moreover, using a gateway the WSN level is opened to the Internet and in a part of the system Internet of Things (IoT) devices are used to present modeling environments for both WSN and IoT platforms.

In the first phase to provide a Model-Driven Development approach, a system is designed and developed in WSN using Contiki and RIOT operating systems. During the design and development of this system, complex, and time-consuming parts are analyzed.

The second phase of the common features of these two operating systems are determined. Based on these commonalities, a meta-model and a modeling environment are designed to construct a Platform-Specific Model (PSM) for both RIOT and Contiki operating systems. Using PSM, the designer can generate codes for Contiki and RIOT target platforms and can make discrete event analyses of the designed system in Petri-net.

The last phase is to create a Platform-Independent Model (PIM) for these two operating systems. The PIM is designed according to provide fast design for the PSM level. The common features and attributes are merged in the PIM level. In this way, model elements and attributes are defined to be automatically generated to provide more abstraction for the designer when transforming PIM to PSM model.

Keywords: Model-Driven Development, Wireless Sensor Networks, Internet of Things, Contiki Operating System, RIOT Operating System, Petri-net.

PROLOGUE

In order to facilitate and accelerate the development of software for Wireless Sensor Networks and the Internet of Things based systems, Domain-Specific Modeling Languages have been developed. However, many of these studies are based on outdated operating systems and technologies. Moreover, within the proposed languages, very few studies have been proposed the code generation and analyses for the target system. Since the continuation of these studies had not made, they have become out of date. Parallely, advancing technologies have created more complexity. As a result, to overcome the increasing complexity and to replace the old studies with the new ones two Domain-Specific Modeling Languages are developed

It has become a necessity for the progress of academic researches in this domain to unite in a standard. For this purpose, a roadmap for modeling of operating systems is defined. Moreover, to provide sustainability of the study in the future, the upper-level Platform-Independent Meta-Model is proposed. In this way, the Platform-Specific Meta-Models can be gathered under this PIM and the Platform-Specific Meta-Models can correspond to the proposed Platform-Independent Meta-Model.

According to these requirements, the necessity and limits of this thesis are determined. As a result of the literature review, research, development and laboratory studies for this study, a Platform-Independent Modeling tool is developed to generate Platform-Specific Models. Next, a code is generated and system analyses are made for the Internet of Things and Wireless Sensor Network platforms using the generated Platform-Specific Models.

TABLE OF CONTENTS

	<u>Page</u>
ÖZET	vii
ABSTRACT	xi
ABBREVIATIONS & SYMBOLS	ix
PROLOGUE.....	xi
1.INTRODUCTION	1
2.BACKGROUND	6
2.1.Wireless Sensor Networks & Internet of Things	6
2.2.Model-Driven Engineering and Domain-Specific Modeling Languages.....	7
2.3.Tools and Technologies.....	8
3.RELATED WORKS	10
4.OPERATING SYSTEM LEVEL & CASE STUDIES.....	13
4.1.Contiki Operating System	13
4.1.1.Tmote-Sky Mote	13
4.2.Case Study: Smart Fire Detection System.....	14
4.2.1.Modeling Approach for Source Node	14
4.2.2.Modeling Approach for Sink Node.....	18
4.2.3.Simulation of Smart Fire Detection Case Study	20
4.3.Case Study: Smart Lighting System	23
4.3.1.Actuator Node Software	23
4.3.2.Simulation of Smart Lighting System	24
4.4.Modeling Approach of Protosocket feature of Contiki for TCP Communication.....	25
4.5.RIOT Operating System	26
4.5.1.ESP32-WROOM-32 Microcontroller	26
4.5.2.Modeling Approach for Server Node.....	27
4.5.3.Modeling Approach for Client Node	28
4.5.4.Case Study: Smart Irrigation System	29

TABLE OF CONTENTS (cont'd)

	<u>Page</u>
4.5.5.Case Study: Smart Lighting System.....	31
4.6.Modeling Approach for TCP Communication	31
5.PLATFORM-SPECIFIC MODELS & CODE GENERATION RULES.....	33
5.1.Code Generation Rules for ContikiOS	33
5.2.Code Generation Rules for RiotOS.....	36
5.3.Code Generation Rules for Protosocket feature of ContikiOS	40
5.4.Code Generation Rules for Saul feature of RiotOS	41
5.5.Models of the Case Studies	42
5.5.1.ContikiOS: Smart Fire Detection System Model.....	42
5.5.2.ContikiOS: Smart Lighting System Model	43
5.5.3.RiotOS: Smart Irrigation System Model	44
5.5.4.RiotOS: Smart Lighting System Model	45
6.PLATFORM-INDEPENDENT MODEL	47
6.1.Indirect Viewpoint: Smart Fire Detection System	51
6.2.Indirect Viewpoint: Smart Lighting System.....	52
6.3.Direct Viewpoint: Smart Irrigation System.....	53
6.4.Direct Viewpoint: Smart Lighting System	54
7.META-MODELS & TRANSFORMATION RULES	56
7.1.Meta-model of Contiki Operating System	56
7.2.Meta-model of Riot Operating System	58
7.3.Platform-Independent Meta-model of Contiki & Riot Operating Systems.....	61
7.4.Transformation Rules for PIM-to-PSM	62
8.SEMANTIC CONSTRAINTS	67
8.1.Semantic Constraints of the DSML4Contiki	67
8.2.Semantic Constraints of the DSML4Riot.....	68
8.3.Semantic Constraints of the Topology Design Viewpoint	69
9.THE OVERVIEW OF THE PROPOSED DSMLS.....	70

TABLE OF CONTENTS (cont'd)

	<u>Page</u>
10.TRANSFORMING A TOPOLOGY MODEL TO A PETRI-NET MODEL.....	72
10.1.Topology Design in proposed DSML	73
10.2.k-boundedness feature of Petri-net.....	74
10.3.Analyzing IoT systems with Petri-net models.....	74
10.4.Petri-Net Case Study: Smart Fire Detection System for a Library.....	76
10.4.1.Generating Petri-net Model for the Fire Detection System.....	76
10.5.LoLA Model-to-Code Transformation.....	77
10.6.PIPE Model-to-Code Transformation	79
10.7.Analyzing the Fire Detection System with Petri-net.....	80
11.EVALUATION.....	82
11.1.Evaluation of DSML4Contiki for Model-to-Text Transformation.....	82
11.1.1.Evaluation of Smart Fire Detection System	82
11.1.2.Evaluation of Smart Lighting System	83
11.2.Evaluation of DSML4Riot for Model-to-Text Transformation.....	85
11.2.1.Evaluation of the Smart Irrigation Case Study	85
11.2.2.Evaluation of the Smart Lighting Case Study.....	86
11.3.Evaluation of Platform-Independent Model to Platform-Specific Model Transformation.....	87
11.3.1.DSML4Contiki: Evaluation of Smart Fire Detection System Model.....	87
11.3.2.DSML4Contiki: Evaluation Smart Lighting System Model	88
11.3.3.DSML4Riot: Evaluation of Smart Irrigation System Model.....	89
11.3.4.DSML4Riot: Evaluation of Smart Lighting System Model.....	89
11.4.Evaluation Results of Code Generation	90
11.4.1.Evaluation Results of Code Generation of the DSML4Contiki...	90
11.4.2.Evaluation Results of Code Generation of the DSML4Riot	92

TABLE OF CONTENTS (cont'd)

	<u>Page</u>
11.4.3.Average of Evaluation Results of DSMLs' Code Generation Success.....	94
11.5.Code Generation Success Rates with PIM-to-PSM Success Rates	96
11.6.Challenges & Issues	98
12.CONCLUSION & FUTURE WORK	99
REFERENCES	103
APPENDIXES	



LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
Figure 1.1 Block diagram representation of the study.....	3
Figure 1.2 Block Diagram of Model-to-Text Rules.....	4
Figure 4.1 The sink and source nodes.....	13
Figure 4.2 Process that has an ETimer.....	16
Figure 4.3 TCPIP Event Element and Send Reply Attribute.....	20
Figure 4.4 Simulation of the case study in CooJa.....	21
Figure 4.5 Message paths are shown in CooJa.....	22
Figure 4.6 Node 4 sends message with relay node.....	22
Figure 4.7 Client Node sends messages to Actuator and Server Nodes.....	24
Figure 4.8 A Server Node that has a thread and a thread that has a Timer.....	28
Figure 4.9 Smart Irrigation System and ESP32s.....	30
Figure 4.10 Smart Lighting System and ESP32s.....	31
Figure 5.1 A process that TCP and UDP flags are activated.....	34
Figure 5.2 A Simple Server-Client Modeling in DSML4Riot.....	37
Figure 5.3 Platform-Specific Model of Smart Fire Detection System.....	43
Figure 5.4 Platform-Specific Model of Smart Lighting System.....	44
Figure 5.5 Platform-Specific Model of smart irrigation system.....	45
Figure 5.6 Platform-Specific Model of smart lighting system.....	46
Figure 6.1 Design viewpoint that contains Direct and Indirect viewpoints.....	50
Figure 6.2 Platform-Independent Model of smart fire detection case study.....	51
Figure 6.3 Platform-Independent Model of smart lighting case study.....	52
Figure 6.4 The Name attribute of the EventBasedTask is set as Light.....	53
Figure 6.5 Platform-Independent Model of the Smart Irrigation System.....	54
Figure 6.6 Usage of the RealTimeTask and RealTimeTimer elements.....	54
Figure 6.7 Platform-Independent Model of the Smart Irrigation System.....	55
Figure 6.8 The designer can sen attributes of the RealTimeTask and RealTimeTimer elements.....	55
Figure 7.1 ContikiOS' meta-model.....	56
Figure 7.2 Riot operating system's meta-model.....	59
Figure 7.3 Platform-Independent Meta-model.....	61
Figure 7.4 Model transformation process.....	62
Figure 7.5 PIM to PSM transformation process.....	63
Figure 7.6 PIM to PSM mapping table.....	65
Figure 8.1 A warning message for a process that has a sensor element.....	68
Figure 8.2 A warning and an error messages for Thread and Socket elements.....	69
Figure 8.3 A warning message for distance violation.....	69
Figure 9.1 Process Model of proposed DSMLs.....	70
Figure 10.1 Transformation table of DSML elements to Petri-net elements.....	72

LIST OF FIGURES (cont'd)

<u>Figure</u>	<u>Page</u>
Figure 10.2 A designed topology of a network in the Topology Viewpoint.....	73
Figure 10.3 Petri-Net model of a designed topology.....	77
Figure 11.1 Bar chart of the smart fire detection system.....	90
Figure 11.2 Bar chart of the smart lighting system.....	91
Figure 11.3 Bar chart of the smart irrigation system.....	92
Figure 11.4 Bar chart of the smart lighting system.....	93
Figure 11.5 Bar chart of average success rate of the DSML4Contiki.....	94
Figure 11.6 Bar chart of average success rate of the DSML4Riot.....	95
Figure 11.7 Code generation success and model transformation success for DSML4Contiki.....	96
Figure 11.8 Code generation success and model transformation success for DSML4Riot....	97

LIST OF TABLES

<u>Table</u>	<u>Page</u>
Table 6.1 Platform-Independent Model icons and their definitions.	50
Table 7.1 Contiki operating system's meta-model elements and their definitions.....	58
Table 7.2 Riot operating system's meta-model elements and their definitions.	60
Table 11.1 Evaluation table of the smart fire detection system model.	88
Table 11.2 Evaluation table of the smart lighting system model.	88
Table 11.3 Evaluation table of the smart irrigation system model.	89
Table 11.4 Evaluation table of the smart lighting system model.	89



LIST OF LISTINGS

<u>Listing</u>	<u>Page</u>
Listing 4.1 A code snippet from the source node software.	14
Listing 4.2 Process Thread code of ContikiOS.	15
Listing 4.3 Process Thread code with a continuous loop.	15
Listing 4.4 Usage of events and timers in Process Thread.	16
Listing 4.5 Full code of the Sink node software.	17
Listing 4.6 Connection and port definition of UDP.	18
Listing 4.7 RPL code to create a multi-hop network.	19
Listing 4.8 A code excerpt of connection creation, message sending, and event handling	19
Listing 4.9 Actuator node's software.	24
Listing 4.10 TCP server side's code using Protosocket.	25
Listing 4.11 A code excerpt from the Server side's UDP.	27
Listing 4.12 A code snippet of a UDP client that uses IPv4.	29
Listing 4.13 The code that is used to model TCP communication.	32
Listing 5.1 Rule to create code files for Sink and Source nodes.	33
Listing 5.2 Code generation rules to generate timer codes.	35
Listing 5.3 Code generation rules to send reply message.	35
Listing 5.4 Code rules to generate libraries.	36
Listing 5.5 Code generation rules to generate necessary parameters for IPv4 and IPv6.	37
Listing 5.6 Code rule to generate threads.	37
Listing 5.7 Code generation rule for thread stack, thread function, and thread parameters. ...	38
Listing 5.8 Code generation rules for CoAP.	38
Listing 5.9 Code generation rule to establish a TCP communication.	39
Listing 5.10 Code generation rules for Protosocket feature.	40
Listing 5.11 Code generation rules to use a temperature sensor via SAUL.	41
Listing 7.1 Meta-model inclusion to use the meta-elements.	63
Listing 7.2 The helper rules that is used for enumerating elements' labels.	64
Listing 7.3 A transformation rule to transform a PIM's Design element to PSM's IoTSystem element.	64
Listing 7.4 Transformation rules to transform PIM elements to PSM elements.	66
Listing 8.1 Semantic constraints of the DSML4Contiki.	67
Listing 8.2 Semantic constraints of the DSML4Riot.	68
Listing 8.3 Semantic Constraints of the Topology Design Viewpoint.	69
Listing 10.1 Calculation of the BOM in Acceleo.	74
Listing 10.2 Java code of calculateDistance function.	75
Listing 10.3 Acceleo Query to trigger calculateDistance function.	75
Listing 10.4 A variable in Acceleo code to store return values of the calculateDistance function.	76
Listing 10.5 Rules to generate LoLA code.	78
Listing 10.6 Rules to generate PIPE code.	79

LIST OF LISTINGS (cont'd)

<u>Listing</u>	<u>Page</u>
Listing 11.1 Delta codes of the server side.	82
Listing 11.2 Delta codes of the client side.	83
Listing 11.3 Server side's delta codes.	84
Listing 11.4 Client side's delta codes.	84
Listing 11.5 Actuator nodes' delta codes.	85
Listing 11.6 Delta codes of the server side.	85
Listing 11.7 Delta codes of the client side.	86
Listing 11.8 Delta codes of the server side.	86
Listing 11.9 Delta codes of the client side.	87

ABBREVIATIONS & SYMBOLS

Abbreviations

ADC	Analog to Digital Converter
AQL	Acceleo Query Language
ATL	Atlas Transformation Language
BOM	Bill of Materials
CoAP	The Constrained Application Protocol
CPU	Central Process Unit
DAC	Digital to Analog Converter
DSML	Domain-Specific Modeling Language
EMF	Eclipse Modeling Framework
FTG+PM	The Formalism Transformation Graph and Process Model
GPU	Graphical Processing Unit
HTTP	Hyper Text Transfer Protocol
IEEE	The Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
KSA	Kablosuz Duyarga Ağları

Abbreviations (Cont'd)

LED	Light Emitting Diode
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MGG	Model-Güdümlü Geliştirme
PBM	Platformdan-Bağımsız Model
PIM	Platform-Independent Model
PIMM	Platform-Independent Meta-Model
PÖM	Platforma-Özgü Model
PSM	Platform-Specific Model
PSMM	Platform-Specific Meta-Model
RPL	Routing Protocol for Low Power and Lossy Networks
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
UML	Unified Modeling Language
USB	Universal Serial Bus

<u>Symbols</u>	<u>Explanation</u>
k	maximum number of tokens to be bound
v or V	Potential electrical difference between two points
R_x	Receiver of an antenna
T_x	Transmitter of an antenna
Avg	Average of variables
T	Period in seconds
mA	milli-amper
mAh	milli-amper hour
ohm	Electrical resistance of a material

1. INTRODUCTION

Internet of Things (IoT) systems are swiftly taking their place in different technologies such as household appliances, Industry 4.0 applications, smart buildings, and Digital Twin systems. Generally, in these systems, embedded devices create a network and work in coordination to present an intelligent environment. The IoT systems consist of different components such as actuators, sensors, computation modules, and log managers for data management. Moreover, IoT systems can work parallelly with Wireless Sensor Networks systems (WSN) to make their network topology more extensible. WSNs increase the coverage of the existing IoT system in the physical environment, without requesting for a direct Internet connection. However, creating a system including WSN and IoT require more design components such as source nodes, sink nodes, client nodes, server nodes, and gateways. The planned system becomes complex with different components requiring to be programmed and verified to work collaboratively and functional. Because of this complexity, design, analyses and, verification phases of these IoT and WSN systems becomes time-consuming, expensive, and burdensome. This complexity can be addressed with Model-Driven Engineering (MDE) techniques (Kardas et al., 2010) to increase the level of abstraction and automatically synthesize the system artifacts. Furthermore, using a model based development methodology, design models can be used for the pre-deployment analyses and verification of the system.

Many developers construct operating systems for Internet of Things (IoT) and Wireless Sensor Networks (WSNs) to provide abstraction from hardware and to present additional features such as routing protocols, threading, and timers. Even if operating systems provide an abstraction to the developer, it is not enough to design an IoT and/or WSN system. Since there are various operating systems and firmware to develop such systems, this variety increases the design complexity, development cost, and learning curve. Further, analyzing and troubleshooting these systems is a time-consuming and error-prone process. As an approach to cope with these problems is to increase the abstraction level using the Model-Driven Development (MDD) approach. In this way, an increase in productivity and reduction in development costs and time can be achieved. To address this problem, it is aimed to provide a higher level of abstraction additional to operating systems for analyzing and developing IoT systems using Model-Driven Engineering (MDE) techniques. For this reason, two Domain-Specific Modeling Languages (DSMLs) based on Wireless Sensor Network (WSN) and Internet of Things (IoT) systems are presented. Moreover, the designers can analyze the target system using Petri-net in the development phase to reduce errors that can occur unexpectedly after the deployment. These are achieved using model transformations that transform the model elements in Platform-Specific Model (PSM) level to both the target platform codes as well as Petri-net elements. By applying the k -boundedness feature to Petri-

net model, various analyses such as power consumption, bottlenecks, and first crashing node are realized for the designed system.

In this thesis study two operating systems, Riot and Contiki are modeled and four case studies are implemented to evaluate the proposed DSMLs. The smart fire detection system is developed on the ContikiOS side and the smart irrigation system is developed on the RiotOS side as case studies. The smart lighting system is implemented as a common case study and this case study is implemented both using DSML4Riot and DSML4Contiki. In this way, code generation capabilities of DSMLs are also evaluated using a common case study.

Contiki is a lightweight, event-driven, and flexible operating system for tiny networked sensors (Dunkels et al., 2004). Contiki operating system has event-driven kernel but the system supports preemptive multi-threading. It has a network stack to provide Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Hyper Text Transfer Protocol (HTTP) communications between nodes as well as including IPv6 (Internet Protocol version 6) and IPv4 (Internet Protocol version 4). Moreover, it has a routing protocol feature to establish a multi-hop network. It's multi-threading is achieved by protothreads. Protothreads are an extremely lightweight, stackless type of threads. RiotOS focuses on program low-power, memory-constrained and wireless Internet of Things devices (Baccelli et al, 2013). It has full support for real-time, multi-thread and pure C based applications as well as considering modularity. Unlike an event-based Contiki operating system, RIOT is a real-time operating system.

The Domain-Specific Modeling Language that is implemented for Riot operating system is named DSML4Riot and the DSML that is implemented for the Contiki operating system is named DSML4Contiki. The following Figure 1.1 describes levels of Domain-Specific Modeling Language. On top of Figure 1.1, the Platform-Independent Model is shown. This model transforms into two Platform-Specific Models that are a model for the Contiki operating system and a model for Riot operating system. Moreover, in additional viewpoint at the Platform-Specific Level, a topology model for the Wireless Sensor Network and Internet of Things systems can be designed. This topology model is transformed into a Petri-net model using the model-to-text technique. Since the output of the transformation is a configuration file, it can be viewed as a model in a Petri-net tool. Platform-Specific Model is transformed to target codes for Riot operating system and Contiki operating system.

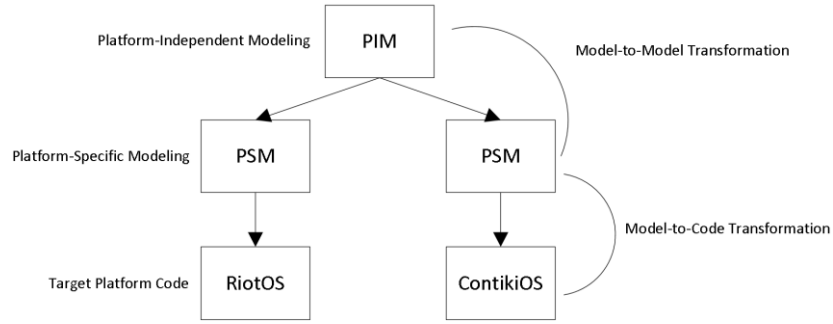


Figure 1.1 Block diagram representation of the study.

In software engineering, model transformations are used widely. Model transformation is about transforming a high-level and generalized model into a more specific and low-level representation of a system. The development process is about transforming a high-level model into an executable program or another model using a set of transformation rules. To achieve this, model-to-model and model-to-text techniques are used.

PIM level represents the Platform-Independent Model. Platform-Independent Model is a model of any system that is independent of any specific platform used to implement this system. Using the Platform Independent Model, Platform-Specific Models related to these two operating systems can be gained. Next, target codes for Riot operating system and Contiki operating system can be generated using these Platform-Specific Models. Hence, providing a PIM level can achieve an efficient development for IoT and WSN based systems. In particular, providing a PIM level is an important process because, if there is no common interface between the various meta-models (Wimmer and Langer, 2013).

The transformation from the Platform-Independent Model to Platform-Specific Model is made using a model-to-model transformation technique. This technique transforms a Platform-Independent Model to one or more Platform-Specific Model according to a set of transformation rules. To achieve this transformation Atlas Transformation Language (ATL) is used. In the Platform-Independent Model, the elements, elements' relations, and modeling process must be independent of specific terms, namings, and configurations. Therefore, the designer should design a Platform-Independent Model, then the transformation rules should be set necessary naming, configurations, and terms during the transformation process. For example, the designer decides to use multiple tasks in the Platform-Independent Model, when these tasks are transforming to threads in Platform-Specific Model their attributes must be set automatically. Because in Riot operating system priority level of each thread must be equal and set to zero to give run-time for all threads. Moreover, in the Contiki operating system autostart process and the continuous attribute must be set to true to start these threads when the device

is powered on. These kinds of configurations should be done in the transformation process. In this way, the designer can model a system independently from the platform as well as achieving fast prototyping.

Model-to-text transformation is a transformation that the result of the transformation is a source code or a configuration text. To transform the Platform-Specific Model to target platform codes, model-to-text transformation is used. The same technique is used to transform a sensor topology model to a Petri-net model. To achieve this transformation, the Acceleo source code generator tool is used. To generate target source codes for Riot operating system and Contiki operating system, libraries, functions, stacks, and examples belong to these operating systems are studied. Code generation rules are written in modular. The following figure 1.2 describes this modular hierarchy for code generation rules of the Contiki operating system and Riot operating system.

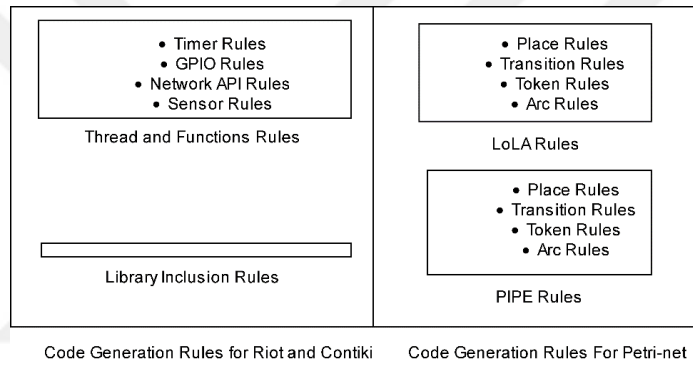


Figure 1.2 Block Diagram of Model-to-Text Rules.

Figure 1.2 represents block diagram of code generation rules. On the left side of figure 2, code generation rules for Riot operating system and Contiki operating system are represented. In operating system side, at the first level library inclusion, thread, and function rules are written. Since library inclusion rules do not contain another rule, it is shown with a blank rectangle. The thread and function rules contain the rules for timer, GPIO, Network API, and Sensor. On the right side of figure 1.2, code generation rules for Petri-net are shown. The code generation rules for LoLA (LoLa, 2009) and PIPE (Liu et al., 2011) are composed of the rules for place, transition, token, and arcs.

This thesis study is organized as follows: Section 2 gives the necessary information on the background of the research domain. Section 3 discusses the related works. Section 4 gives operating system level information, mentions the modeling approach of the operating systems, and presents the case studies. Section 5 introduces the Platform-Specific Models of the case studies and code generation rules. The Platform-Independent Model that is independent level from the platform

is presented in Section 6. The Platform-Specific Level's meta-models that are also the meta-models of the Riot and Contiki operating systems are addressed in Section 7. The semantic constraints of the DSML4Riot, DSML4Contiki, and topology design viewpoint are given in Section 8. The overview process model of the proposed DSMLs is given and is explained in Section 9. Section 10 explains the transformation of a topology model to a Petri-net model and k-boundedness feature of the Petri-net based on a case study. Evaluation of the proposed DSMLs and PIM to PSM transformation is made in Section 11 as well as mentioning challenges and issues that are encountered during the development of this thesis study. The thesis study is concluded and future works are mentioned in Section 12.

In Appendix A, figures that belong the first phases and previous versions of the case studies are given. In Appendix B, the codes that are used for the implementation of the case studies using DSML4Contiki and DSML4Riot are given as well as including code generation rules.

2. BACKGROUND

In this section Wireless Sensor Networks, Internet of Things, and Model-Driven Development and their relation are explained. Moreover, tools and the modeling paradigm that are used in this thesis study are mentioned.

2.1. Wireless Sensor Networks & Internet of Things

Embedded systems are computing systems designed to perform a single task-based on micro-controllers to measure environmental values and to convert them into a human-readable format using electrical signals. These systems can perform the tasks in an event-based or real-time way. Additionally, the embedded systems may have an operating system to manage the system resources. Additional antenna chip and communication protocol are required to create a network between embedded systems to collect data in a wide area and transmit it wirelessly. The devices that have wireless communication capability use neighbor devices to establish a Wireless Sensor Network. Thus, WSNs can be used for data acquisition in wide areas for low-power applications. Using a low-power microcontroller (e.g TmoteSky mote) and an operating system, this can be achieved. An operating system can sleep the device and provide resource management. When a process is yielded, the resources can be released or until an event occurs, it can switch the device to low power mode.

Generally, WSNs use multiple neighbor nodes to send a packet from a source node to a sink node. The nodes that are collecting the data are called the source node, and the node that receives all data and transmits them to a gateway called the sink node. If a gateway collects all the data from a sink node and sends them to the Internet, then a WSN system can become a part of an Internet of Thing (IoT) system. Thus, WSNs and other Wi-Fi integrated microcontrollers can be connected to the Internet. Moreover, the WSN empowers the communication of IoT devices in a wide area (using IEEE 802.15.4 protocol) where direct Internet connection (using IEEE 802.11 protocol) is not available. Moreover, an Internet of Things device can be used in the system, thus these devices can be controlled from the Internet and send data to the Internet, directly.

A system that has WSN and IoT subsystems are composed of many components is a complex system. The software to implement these components can be written in different programming languages, operating systems, and platforms. This causes even more complexity in the development process of these systems. To tackle this complexity, a higher level of abstraction is needed, such as a Domain-Specific Modeling Language (DSML). A DSML provides a model checking for verification, code generation to speed up the development process, and an analysis model. As a result, it can generate a highly completed system including processes

such as network protocols, threads, and sensors that work together in the designed system.

2.2. Model-Driven Engineering and Domain-Specific Modeling Languages

A recent approach to handle the increasing complexity of software concentrated systems is Model-Driven Engineering (MDE) (Schmidt, 2006). Increasing variety and complexity in software systems make the development process of these systems difficult using traditional methods. This problem caused to require new software development methods and techniques. One of the methods to overcome this problem is to apply Model-Based Development to a system. This method aims to move the starting point of software development processes to model from coding. Accordingly, domain meta-models, system models, model-to-model transformations are required to generate codes for the system.

Model-Driven Architecture (MDA) (Frankel, 2003), that provides the concretization of MDD, also increases MDD productivity (Kos et al., 2011) and reduces development costs (Vallecillo, 2008). MDD provides a higher level of abstraction to enable software developers to focus on the problem before to focus on to try reducing the complexity. (Schmidt, 2006).

The basic concepts of the MDD approach according to Object Management Group (OMG) are meta-models, models, and transformations. A system is represented by a model and a model is represented by its meta-model. According to the meta-model, a model is instantiated. The instantiated model can either be transformed into another model or codes can be generated according to this instance model.

MDD uses various methods to achieve goals. One of these methods is the Domain-Specific Modeling Languages (DSML). A DSML has concreated images that represents a piece of code that will be generated. A designer creates a model using and combining these concrete images to achieve a goal. DSML generates codes according to the designer's model. In other words, a model that is represented by a graphical syntax provides fast development, fast prototyping, and intelligibility (Schmidt, 2006; Gray et al., 2007). The development of a DSML usually starts with a definition of its language model (Strembeck and Zdun, 2009). In other terms, elements, abstractions, and relations belong to a domain must be defined in order to represent a language model. A language model conforms to a meta-model. A meta-model includes cardinality constraints, attributes, and attribute types of a language model. A language model is supported by a modeling tool for the designer.

In the literature, there are various studies on IoT and DSMLs. Some of them address meta-models as well as issues such as node connectivity, code generation (Harrand et al., 2016; Negash et al., 2017), security (Beaulaton et al., 2018), service architecture (Adda et al., 2014), and run-time adaptability (Hussein et al., 2017). Additionally, to traditional approaches, recently new approaches are studied such as facet-oriented modeling (de Lara et al., 2018), model federation (Golra et al., 2016), and template-based meta-modeling (de Lara and Guerra, 2010). In the study of (Bruneliere et al., 2019), a comprehensive survey of related modeling approaches is addressed.

2.3. Tools and Technologies

In this thesis study, Acceleo, Sirius, Eclipse Modeling Framework (EMF), and Atlas Transformation Language (ATL) are used to construct a DSML. Acceleo is a template-based software to create custom code generators. It has simple syntax, efficient code generation, and helpful features. It can easily create a code generator from the source code of an existing system (Acceleo, 2019). Sirius is a software that allows to easily create graphical modeling interface for the designer. Sirius provides a generic workbench for model-based engineering that could be easily developed to meet the domain's requirements. Based on a viewpoint approach, it makes possible to deal with complex architectures on target domains. It provides multiple representations of a view, constraint checking of a model, compatibility with Java language to interact with other systems (Sirius, 2019). The EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model (EMF, 2019). ATL (ATL Transformation Language) is a model transformation language. In the field of MDE, ATL provides ways to produce a set of target models from a set of source models. An ATL transformation program is composed of rules that define how to source model elements are matched and navigated to create and initialize the elements of the target models (ATL, 2019).

The Petri-net is a simple language for modeling discrete distributed systems. Because of its simplicity, it can be adapted to a wide range of domains to verify a system. Therefore, it is frequently used to sustain a DSML with a well-supported language such as Petri-net (Peterson, 1981). To provide a graphical representation for a Petri-net model PIPE high-level modeling tool is used (Liu et al., 2011; Bonet et al., 2007). Moreover, LoLA low-level verification tool is used (LoLA, 2009). Even if LoLA is not suitable for modeling, its interface is designed easily integrate into a modeling framework. Input and output operations of LoLA is organized as files. This makes easier to generate a LoLA model using code generation rules.

The Formalism Transformation Graph and Process Model (FTG+PM) (Lucio et al., 2012) represents the MDE process by explicitly modeling it. The FTG+PM consists of the Formalism Transformation Graph (FTG) with its complement, the Process Model (PM), and design steps in the MDE lifecycle such as requirement gathering, domain-specific design, use cases, verification, simulation, analysis, deployment, code generation, and execution. The FTG describes explicitly and precisely of formalisms and their relationships as transformations between these formalisms. The PM specifies an MDE process using these formalisms and transformations as well as using the instances of those formalisms. The PM is specified in the Unified Modeling Language (UML) Activity Diagram 2.0 language (Object Management Group, 2017).



3. RELATED WORKS

In this section, the studies related to this thesis study are mentioned. They are about modeling of Wireless Sensor Networks and Internet of Things (Vangheluwe et al., 2016). Firstly, a brief history of Model-based Design is mentioned. Secondly, studies in the literature are discussed by comparing this thesis study. Lastly, the idea about the future of DSMLs is addressed.

After the 1980's software crisis, new paradigms are defined to overcome such as Object-Oriented Programming, Expert Systems, and Model-Driven Engineering. Especially, Object-Oriented Programming had a big impact on Software Engineering by its definition of "Everything is an object". However, this paradigm is shifted to the Model-Driven paradigm by a new definition of "Everything is a Model" (Egesoy, 2017). The idea behind the Model-Driven paradigm is using diagrams to represent a detailed specification of a system then generate code automatically for new systems or transform a model to another model using transformation rules (Bézivin, 2005).

In the literature, some MDE studies are conducted to allow faster the design, development, and implementation steps of WSN and IoT systems. Two surveys (Essadi et al., 2016; Malavolta et al., 2014) realized a systematic mapping study of this domain. According to these surveys, various MDE based languages were developed between the years 2007 and 2015. The main motivation for these studies is code-generation to provide fast prototyping. In none of these studies, ContikiOS and RiotOS are not addressed.

To facilitate the design and development of WSN and IoT systems, several DS(M)Ls are proposed. In the study of (Dantas et al., 2013), a DSL named LwiSSy is presented to model Wireless Sensor and Actuator Network systems. In their study, the separation between application domain experts and network experts is provided. Moreover, the DSL considers structural, behavioral, and optimization as three-dimension design steps using multiple viewpoints. However, the study is just limited to a high abstraction level and is not include PSM level and an operating system for WSN and IoT. Further, the study does not have a well-accepted tool like Petri-net and does not offer a road map for verification and validation of the design process.

The study of (Rodrigues et al., 2017) proposes a Model-Driven Architecture (MDA) to provide Platform-Independent modeling (PIM), Platform-Specific modeling (PSM), and transformation rules to develop a WSN application. The development process is stated. However, there is not any tool or method studied for post-design. In the proposed DSMLs of this thesis study, a model is generated to analyze power consumption, bottlenecks, and first crashing node of designed

network topology using Petri-nets. A modeling framework in study of (Doddapaneni et al., 2012) proposed to separately model the software components and their interactions, the low-level and hardware specification of the nodes, and the physical environment where the nodes are deployed. This multi-view architectural approach requires relating the models together before mapping them to code. The study offers a code generation for the simulation of the designed network however, the simulation can be made on supported devices. This causes a device constraint for the design. Moreover, what if there are devices that use other wireless protocols (e.g IEEE 802.11) and what if there have to be unsupported devices in design to be simulated. To resolve this problem, traditional methods can be used. Additionally, a traditional method Petri-nets are used to provide basic analyses for WSN and IoT based systems.

In a complex and multi-component system, a general solution must be provided to analyze the system for pre-design and post-design phases by covering all hardware components. A process model is proposed in (Tei et al., 2014) to enable stepwise refinement to address data and network-related concerns. Their goal of modeling is similar to the modeling purposes of the study in (Doddapaneni et al., 2012). The study focuses on the separation of responsibilities between domain experts and network experts to design WSNs. Both studies are focused on nesC language and TinyOS. Both studies present code generation for nodes and simulation. However, they are not considered what happens if there are two operating systems or one does not have a simulation tool and there are unsupported devices that cannot be used in a simulator. As a result, these questions are the general idea behind why Petri-net is used for analyses by covering all hardware in the system and separated from operating systems. Surely, the Petri-net cannot achieve as much as a simulation tool can. However, using well-known methods of Petri-net (e.g k-boundedness) critical analyses can be made. In the study of (Kim et al., 2018), a discrete event system specification is made for IoT systems using multi-core CPUs (Central Programming Unit) and GPUs (Graphics Processing Unit). They simulated IoT nodes to sense the fire spreading. To achieve the simulation, they used high tech CPUs and GPUs. However, this method requires high cost and it only covers IoT devices. In this thesis study, a transformation that covers all system components from WSN and IoT to Log Manager is considered without the requirement of costly CPU or GPU.

The study (Lekidis et al., 2019), introduces design flow for web service applications in WSN. Moreover, it also offers statistical model checking of a stochastic model. They use Contiki as an operating system and Contiki's REST feature as an application. They provide their stochastic model checking based on their proposed DSL. However, there are limitations to that study. First, they are bounded to hard-wares that are supported by ContikiOS. In the Petri-net model, the hardware level is separated from operating systems. The study is only focused on

the REST API (Application Programming Interface) of Contiki. In this thesis study, it is highly focused on creating server/client applications based on TCP, UDP, and WebServer in Contiki viewpoint and server/client applications based on TCP, UDP, and CoAP (The Constrained Application Protocol) in Riot viewpoint. Moreover, this thesis study's proposed DSML is not constrained only Contiki and Riot operating system, it also includes code generation for ESP8266, RaspberryPI, and a Log Manager. However, since they are achieved in teamwork, they are not included in this thesis study. A relevant reader can look into the study of (Asici et al., 2019) to have information about it.

A similar study in the Riot side (Kuhlwein et al, 2019), addresses a DSML for ultra-thin IoT devices. They provide a DSML named IoT-PML and code generation is made for a sensor device driver. They use Riot operating system to implement a case study. However, their study is in its early stages of development. Their DSML has very limited graphical editor support and the model references are not shown in the editor. Their DSML requires manual operations to achieve code generation. Furthermore, they do not evaluate the code generation success and their case study does not contain a network-based process. It is important to establish a network-oriented case study in IoT-based systems and DSMLs. In this thesis study, two case studies are given in the Riot side considering a network protocol as well as threading and input/output operations. The meta-model, model and code generation rules are also given.

Even using a single feature of Petri-net (e.g k-boundedness) many analyses results can be achieved. An experienced designer can apply other features of Petri-nets to have different analyses. The study of (Lekidis et al., 2019) also does not offer any power analysis. In this thesis study, the k-boundedness feature of Petri-net is used for bottleneck analysis, power consumption analysis, and node crash (first node die) analysis. Moreover, their DSML has a constraint to at most a few nodes that can be added and it does not offer scale-ability.

Finally, there are several studies that are addressing the manual Petri-net modeling of the IoT systems for system analysis, e.g. (Nakahori and Yamaguchi, 2017). However, these approaches do not benefit from MDE techniques for the automatic development of Petri-net models. This thesis study contributes to the aforementioned studies by providing a generative MDE approach for analyzing WSN and IoT models using Petri-net. In the future, Model-Driven software engineering is going to take an important part in Industry 4.0 applications (Thramboulidis et al., 2017), Digital Twin systems (Bagherzadeh et al., 2017), and Artificial Intelligence (Mülders et al., 2011).

4. OPERATING SYSTEM LEVEL & CASE STUDIES

In the following sub-sections, Contiki and Riot operating systems are mentioned. The case studies are introduced. Code snippets are given and the modeling approach is mentioned to explain how these two operating systems are modeled. To implement the case studies, most of the target codes are generated using the DSML4Contiki and the DSML4Riot. The case studies' implementation steps are listed and case studies are represented by figures. The features which are not covered by the case studies and which have importance are explained in sub-sections 4.4 and 4.6. The code generation rules of these features and more are mentioned in the next section.

4.1. Contiki Operating System

Contiki is an open-source and event-based operating system for the Internet of Things. Contiki is designed for low-cost and low-power microcontrollers. It supports IPv6 and IPv4, including 6LoWPAN (Mulligan, 2007), RPL (Routing Protocol for Low Power and Lossy Networks) (Winter, 2011), CoAP (Shelby et al., 2014). Contiki is implemented in standard C. It has a Cooja simulator. Applications of Contiki can be emulated in Cooja before flashed into hardware. A new group extends the Contiki operating system with the name of Contiki Next Generation, however, Contiki-NG is too new to gain maturity and has a lot of issues (Tomasic et al., 2018). Consequently, the case studies and DSML development are made based on ContikiOS.

4.1.1. Tmote-Sky Mote

TmoteSky is a low power wireless sensor device. It offers a high data rate sensor network for applications requiring ultra-low power (TmoteSky Datasheet, 2019). It also provides high reliability and ease of development. It has multiple integrated peripherals including a 12-bit ADC, 12-bit DAC, Timer, and UART (Universal Asynchronous Receiver Transmitter) bus protocols. It has an antenna chip that is IEEE 802.115.4 compliant and sensors for measurements such as humidity, temperature, and light. Figure 4.1 describes the TmoteSky motes that are used to implement the smart fire detection and the smart lighting system.



Figure 4.1 The sink and source nodes.

4.2. Case Study: Smart Fire Detection System

The smart fire detection system was developed to evaluate the DSML. The case study was implemented as a multi-hop network. During the implementation of this case study, it was aimed to use features of the Contiki operating system as much as it can. To have more modeling elements in DSML, a case study that includes many features is required. As a result, to have a comprehensive DSML, the system should be analyzed in detail and the codes that represent system elements should be determined. In the following sub-sections, Contiki application codes and modeling approaches based on these codes are given.

4.2.1. Modeling Approach for Source Node

The mote used to implement the case studies is TmoteSky. It operates in the frequency of 2.4 GHz and has an antenna that uses the IEEE 802.15.4 protocol. Sampled sensor values are transmitted to neighbor nodes and they reach the sink node endpoint. To activate multi-hop forwarding in ContikiOS, the RPL library is used. The sampled temperature values are sent using UDP protocol with IPv6. To establish a UDP communication, port numbered 3000 as outgoing and port numbered 3001 as incoming are used. Event timers are added in each process to create event-based processes. Moreover, since a multi-hop network is implemented, when source nodes are sampled a temperature value, it can be sent directly to the sink node or it can be sent via a neighbor node in case of sink node is not in range. In Listing 4.1, a code snippet from source node software is given.

```
PROCESS_THREAD(measure_temp, ev, data) {
    static struct etimer et;
    PROCESS_BEGIN();
    etimer_set(&et, SEND_INTERVAL);
    while(1) {
        PROCESS_WAIT_EVENT();
        if(ev == PROCESS_EVENT_TIMER) {
            value[0]=get_temp();
            etimer_restart(&et);
            PROCESS_END();
        }

        PROCESS_THREAD(broadcast_process, ev, data){
            static struct etimer et, uip_ipaddr_t ipaddr;
            PROCESS_BEGIN();
            client_conn = udp_new(&ipaddr, UIP_HTONS(3000), NULL);
            udp_bind(client_conn, UIP_HTONS(3001));
            while(1) {
                PROCESS_YIELD();
                if(etimer_expired(&et))
                else if(ev == tcpip_event) }
                PROCESS_END();
            }
            static void send_packet(){
                char buf[100];
                sprintf(buf, "Temp %d Humid %d Light %d", value[0], value[1], value[2]);
                uip_udp_packet_sendto(client_conn, buf, strlen(buf), &server_ipaddr, UIP_HTONS(3000));
            }
        }
    }
}
```

Listing 4.1 A code snippet from the source node software.

Since Contiki is an event-based operating system, it requires event creation to process a task. An event can be generated based on the time using the etimer feature of Contiki. In other words, the Contiki operating system has processes and these processes have timers that create an event. To achieve this, an etimer named *et* is defined. If the etimer's value in RAM must be preserved for each CPU cycle, then it must be defined as static. *SEND_INTERVAL* defines the time interval as seconds. A while loop is needed to prevent *PROCESS_END*'s instruction. If it is instructed, then the process ends. Moreover, a process should be yielded voluntarily until any event occurs, in this way process does not run unnecessarily. When an event is generated, the process continues to run and collects temperature data to send it to the sink node.

The UDP packet is preparing to be sent in *send_packet* function. This function creates a buffer, prints the sensor values to this buffer, and calls necessary UDP function. The measurement data is sent to the sink node using *uip_udp_packet_sendto* function from the defined outgoing port. In these studies, (Karaduman et al., 2019a, 2019b), this case study is extended and explained in detail.

So far, this code snippet showed a process block, an event timer, a buffer, a continuous loop, UDP functions, and sensor functions. To model these elements, each code block that represents operating system features was decomposed and was modeled modularly. The following Listing 4.2 shows the basic generated code when the designer adds a process thread element in the model.

```
PROCESS_THREAD (measure_temp, ev, data) {
PROCESS_BEGIN();
PROCESS_END ();
}
```

Listing 4.2 Process Thread code of ContikiOS.

The designer may decide to run a process just one CPU cycle. Therefore, this process should have a feature that determines it is either continuous or non-continuous. This can be achieved with a single Boolean attribute in the meta-model. According to this Boolean attribute, the DSML generates either a while loop to create a continuous process or do not create a while loop (see Listing 4.3).

```
PROCESS_THREAD (measure_temp, ev, data) {
PROCESS_BEGIN();
while(1) {
} // end of while loop scope.
PROCESS_END ();}
```

Listing 4.3 Process Thread code with a continuous loop.

It is important to use timers in operating systems. Tasks can operate according to a period using timers. For example, it is known that temperature value does not change instantly, so an experienced designer considers saving battery power and sets a sampling period for that process. For these kinds of purposes, a process should have a timer and the following code should be generated. Moreover, since it is wise to yield this process when there is not any event then *PROCESS_WAIT_EVENT* macro should be added (see Listing 4.4).

```
PROCESS_THREAD (measure_temp, ev, data)
{
    static struct etimer timer_name;
    PROCESS_BEGIN();
    etimer_set(&timer_name, time_interval);

    while(1) {
        PROCESS_WAIT_EVENT();
        if(ev == PROCESS_EVENT_TIMER)
        {

            ... // user code here.

        } // end of while loop scope.

        PROCESS_END ();
    }
}
```

Listing 4.4 Usage of events and timers in Process Thread.

According to the above code, the designer has a continuous process that has an event timer. *if (ev == PROCESS_EVENT_TIMER)* statement may include the user's delta codes that runs when the time interval ends. This is the basic code that is generated when the designer decides to have a timer event-based continuous process. A timer-based process provides the designer to select a time interval for data acquisition. In time-critic domains, the user may select a shorter time interval. However, it is suggested that in time-critic domains a real-time operating system with domain-specific hardware should be used. For this purpose, Riot is selected as a real-time operating system and ESP32 (Espressif Systems, 2019) development board is selected as hardware to implement the case studies. In section 4.5.1, it is mentioned. Figure 4.2 represents a continuous process model that has an *Etimer* element in DSML4Contiki.



Figure 4.2 Process that has an ETimer.

The process is autostarted when the device is powered on. It is connected to *Etimer* element by *hasEtimer* relation. *Etimer* element named *Etimer Example*

counts down to 0 starting from 30 and when it is expired, it generates a timer event in this process.

```

PROCESS(measure_temp, "Measure Temp process");
PROCESS(broadcast_process, "Collect all data");

AUTOSTART_PROCESSES(&measure_temp, &broadcast_process);
#define MAX_BUF_LEN 45
#define SEND_INTERVAL 3 * CLOCK_SECOND

static struct uip_udp_conn *client_conn;
static char buf[100];
static int value[2];

static void timeout(void){
printf("Client sending to: ");
PRINT6ADDR(&client_conn->ripaddr);
sprintf(buf, "Temp %d ", value[1]);
printf(" (ALL DATA: %s)\n", buf);
}

static void
data_handler(void){
char *str;
if(uip_newdata()){
str = uip_appdata;
str[uip_datalen()] = '\0';
printf("DATA recv '%s'\n", str);
PRINT6ADDR(&Client_Connection->ripaddr);} //end of data_handler

PROCESS_THREAD(measure_temp, ev, data){
static struct etimer et;
PROCESS_BEGIN();
etimer_set(&et, SEND_INTERVAL);

while(1) {
PROCESS_WAIT_EVENT();
if(ev == PROCESS_EVENT_TIMER) {
value[1]=get_temp();
etimer_restart(&et);}
PROCESS_END();
}

PROCESS_THREAD(broadcast_process, ev, data){
static struct etimer et, uip_ipaddr_t ipaddr;
PROCESS_BEGIN();

PRINTF("Start UDP Protocol\n");
client_conn = udp_new(&ipaddr, UIP_HTONS(3000), NULL);
udp_bind(client_conn, UIP_HTONS(3001));
PRINT6ADDR(&client_conn->ripaddr);
etimer_set(&et, SEND_INTERVAL+1);

while(1) {

PROCESS_YIELD();
if(etimer_expired(&et)) {
timeout();
etimer_restart(&et);
}
else if(ev == tcpip_event) {
data_handler();}

PROCESS_WAIT_EVENT();
if(ev==PROCESS_EVENT_TIMER)
{
etimer_restart(&et);
}
PROCESS_END();
}
}

```

Listing 4.5 Full code of the Sink node software.

At the above Listing 4.5 the source node software is given for temperature data acquisition. *PROCESS* function defines the processes that will run. To start these processes automatically, when the device is turned on, *AUTOSTART_PROCESS* function is used. Memory addresses of processes are given as a parameter to this function. The function named *get_temp* is used to sample temperature data. Its definition can be found in *sht11* library of the Contiki operating system.

```
client_conn = udp_new(&ipaddr, UIP_HTONS(3000), NULL);
udp_bind(client_conn, UIP_HTONS(3001));
PRINT6ADDR(&client_conn->ripaddr);
```

Listing 4.6 Connection and port definition of UDP.

At the above Listing 4.6, *client_conn* struct is bound to ports 3000 and 3001. Port number 3000 is defined as the outgoing port and port number 3001 is defined as the incoming port. A special print function is called to prompt the IP (Internet Protocol) address of the node. When the timer period ends, *timeout* function is called. When this function is called, sampled temperature values are sent to the sink node. As a result, these components and features are decomposed and modeled either an attribute or an EClass in the meta-model.

4.2.2. Modeling Approach for Sink Node

In these studies, a single-hop (Karaduman et al., 2019a) and a multi-hop (Karaduman et al., 2019b) based systems are designed and implemented. To achieve a multi-hop network a set of codes must be added in the sink node application code. Creating a single-hop network in the Contiki operating system does not require codes in Listing 4.7. In single-hop mode, any data packet can be transmitted when the sink node in coverage of a source node. If the sink node is not in the coverage of a source node, the data packet cannot be delivered. A multi-hop network can be established to overcome this situation. However, Routing Protocol for Low-Power and Lossy Network (RPL) library must be used to create a multi-hop network. This library and its functions are used in the sink node's application code. The sink nodes create a network tree and advertise itself as the root node. A source node decides its closest neighbor node to hand over its data packet. In this way, source nodes can be repeater nodes and can forward data packets to the sink node. In the below Listing 4.7 the code required to create a multi-hop network is shown.

```
if(root_if != NULL) {
    rpl_dag_t *dag = rpl_set_root (
        RPL_DEFAULT_INSTANCE,(uip_ip6addr_t *) &ipaddr );
    printf("RPL is created\n");
} else {
```

```
printf("RPL cannot be created\n");}
rpl_repair_root(RPL_DEFAULT_INSTANCE);
```

Listing 4.7 RPL code to create a multi-hop network.

In the DSML4Contiki, the designer can design a single-hop network or a multi-hop network. If the designer decides to use a multi-hop network the above code is generated in the sink node's code. The root is set according to the IP address of the sink node using *rpl_set_root* function. Moreover, the sink node must repair the network in case there is a change in the topology. To achieve this, *rpl_repair_root* function is used.

```
static struct uip_udp_conn *Server_Connection, uip_ipaddr_t ipaddr;
static void TCPIP_Event_handler(void){
char buf [50];
if(uip_newdata()) {

((char *)uip_appdata)[uip_datalen()] = 0;
PRINTF("Server received: '%s' from ", (char *)uip_appdata);
PRINT6ADDR(&UIP_IP_BUF->srcipaddr);
PRINTF("DATA sending reply\n");
uip_ipaddr_copy(&Server_Connection->ripaddr, &UIP_IP_BUF->srcipaddr);
uip_udp_packet_send(Server_Connection, "hi", sizeof("hi"));
uip_create_unspecified(&Server_Connection->ripaddr);
memset(&Server_Connection->ripaddr, 0, sizeof(Server_Connection->ripaddr)); }

PROCESS_THREAD(MY_Server_Process, ev, data) {
PROCESS_BEGIN();

Server_Connection = udp_new(NULL, UIP_HTONS(3000), NULL);
if(Server_Connection == NULL) {
PRINTF("No UDP connection available, exiting the process!\n");}
udp_bind(Server_Connection, UIP_HTONS(3001));
PRINT6ADDR(&Server_Connection->ripaddr);
PRINTF(" local/remote port %u/%u\n", UIP_HTONS(Server_Connection->lport),
UIP_HTONS(Server_Connection->rport));
while(1){

PROCESS_YIELD();
if(ev==tcpip_event)
{
TCPIP_Event_handler();
}}
PROCESS_END();
}
```

Listing 4.8 A code excerpt of connection creation, message sending, and event handling

To send a reply to a received message, the *TCPIP Event* element must be added to the model in the Sink node side. Since Contiki is an event-based operating system, if any UDP or TCP packet is received, an event named *tcipip_event* is generated. Therefore, sending the reply message is bound to this event (see Listing 4.8). Figure 4.3 summarizes what is described so far. It represents a process thread that has a *tcipip_event* element and *send reply* attribute is on.

To print the IP address, *PRINT6ADDR* function is used. This is a special function that is implemented in Contiki operating system to print the IPv6 address. The *uip_ipaddr_copy* function copies the incoming packet's source address to the

destination address to send the reply message, then it sends the message using the send function. Next, the *uiplib_create_unspecified* function is called to remove the remote IP address when the reply packet is sent. To be sure that memory area is filled by zeros, memset function is called (see Listing 4.8).

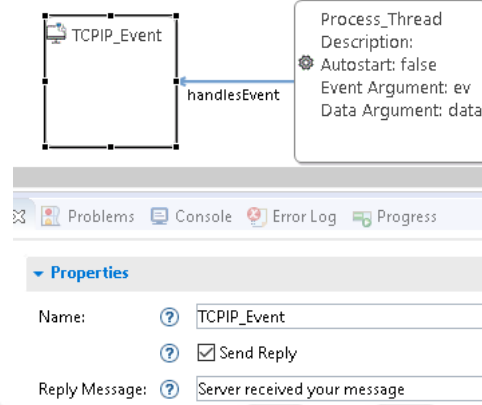


Figure 4.3 TCPIP Event Element and Send Reply Attribute.

Using these mentioned rules, the required codes are generated for the target system. These codes are generated according to code generation rules and designed models. Generated codes might be required the user's delta code to be completed such as the controlling conditions, an algorithm or a variable to have a fully functional system. In the following sub-section, the simulation of the smart fire detection system is made in CooJa. The nodes are programmed to measure light, humidity, and temperature value. Moreover, some calculations are made for an indoor positioning system for simulation purposes (Karimpour et al., 2019). To determine which nodes send a message their IPv6 addresses included in their message packet.

4.2.3. Simulation of Smart Fire Detection Case Study

A simulator for wireless sensor networks is a valuable tool to develop a system. Software development for sensor networks can be simplified by using a simulator tool. CooJa is a simulator for the Contiki operating system (Österlind et al., 2006). It has a lot of features and capabilities to run applications in it without physically programming a wireless sensor device. A sensor node software can be run either as a compiled program and run in the simulator, or in a physical sensor node to be emulated by the simulator. During the implementation of the case study, the sink and the source node software are simulated in CooJa, then flashed into TmoteSky motes.

When the simulation results are satisfiable, the code is tested in Tmotesky motes. In this way, a lot of re-flashing operations are prevented. The CooJa

simulator also provides the scalability of the nodes. There can be many nodes in the simulation environment. Even if there are not enough sensor nodes or there are not any nodes, the CooJa can simulate many sensor nodes and various programs of the sensor nodes. In this way, a lot of time-consuming parameters are eliminated such as re-flashing of a lot of nodes, tracking each node's functionality, tracking sensor topology.

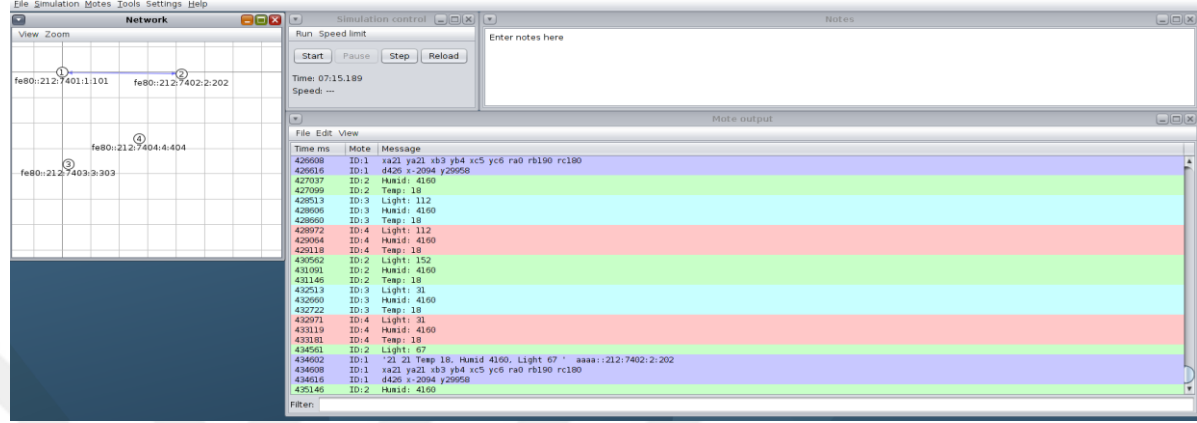


Figure 4.4 Simulation of the case study in CooJa.

Figure 4.4 shows the sink node (number 1) creates an RPL tree and source nodes find their closest neighbor node to deliver their packets to the sink node. The bi-directional blue arrow shows that node 1 and node 2 are communicating. The sink node gets the IP address of *fe80::212:7401:1:101*. This IP address is given by the CooJA simulator. In the physical world, Tmote Sky motes get IP addresses from the addressing chip on it. Since the closest node is the node that is numbered 2, it is the first node that receives the sink node's advertising message. It can be predicted that the next node that is going to receive the advertising message is number 4 then number 3.

This topology represents a single-hop network since all source nodes can directly communicate with the sink node. If one of the nodes can not reach directly to the sink node, it has to select a neighbor node. It is selected according to distance. Because, if more than one node was selected as neighbor nodes, then the message had to be duplicated for each neighbor. This situation would create complexity for the system and would cause more battery consumption for the nodes. As a result, selecting the closest node as a neighbor node is also beneficial for propagation delays. The importance of propagation delays is mentioned in Section 10.

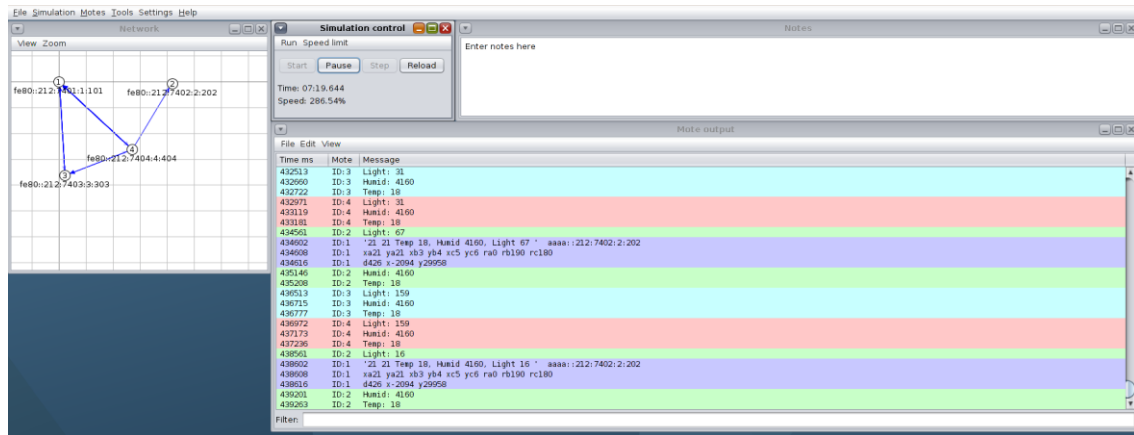


Figure 4.5 Message paths are shown in CooJa.

As figure 4.5 describes, a multi-hop network is created and client nodes send their sensor values to the sink node. The blue arrows show the direction of data packets. The sink node receives multiple data packets from the source nodes. The node 4 is in coverage of the sink node. It can directly send its data packets. However, its position is changed and the topology becomes a multi-hop network.

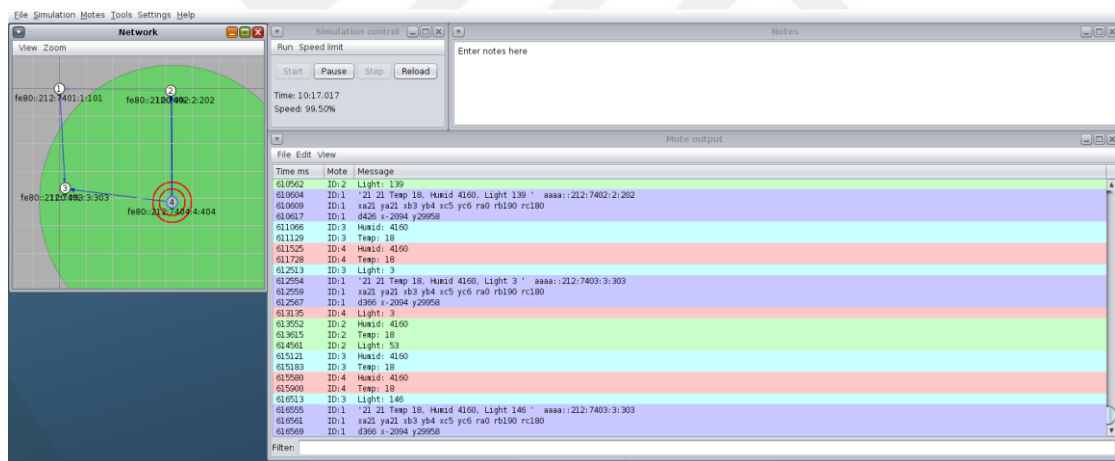


Figure 4.6 Node 4 sends message with relay node.

Figure 4.6 shows that node number 4 could not establish a direct connection with the sink node 1. As the blue arrows show, number 4 has two neighbor nodes which are 3 and 2. The closest neighbor node is numbered 3. Therefore, data packet must be delivered to node numbered 3. The colored window shows the output messages of the nodes. The received measurement results and IP addresses of the nodes are prompt to this window by the sink node that is represented by ID 1. The numbers that are shown following with temperature and IP address values are coordinates of the nodes and RSSI values. This is implemented for test purposes. As it can be seen that, when the node 4 is moved away from the coverage of the sink node, it started to send its messages via its neighbor node. The green circle

shows the coverage of node 4. Since the sink node 1 is not inside of the green circle of the source node 4, the source node 4 cannot send its message. This is also an example of the topology change. Because at first, the source node 4 was in the coverage of the sink node 1, then it is moved a few meters away. As mentioned, the sink node must repair the network topology in case of changes. The sink node can repair the topology using the RPL library's function *rpl_repair_root*. This can be beneficial for the source nodes that have to be in movement.

4.3. Case Study: Smart Lighting System

To evaluate the DSML4Contiki more comprehensive, the second case study was implemented. The case study has a source, a sink, and an actuator node. The sink node has the same code that is used to implement the smart fire detection case study. It receives the source node's messages and delivers them to the centralized computer. The actuator node receives a message from the source node to either turn on or off the light. The source node measures the light intensity of the environment and delivers them to the sink node. Moreover, it sends the on/off message to the actuator node according to measured light intensity. The source node nearly has the same code with a smart fire detection case study. Since they are only the actuator node's code different than the smart fire detection case study, only the actuator node's code is mentioned in the following sub-section.

4.3.1. Actuator Node Software

The following code snippet shows the actuator node's software. It listens to port 8888 for a UDP packet that is sent by the source node. In the protothread, *PROCESS_WAIT_EVENT* macro is added not to run process until any event occurs. In this case, an event can be generated as a *tcpip_event*. When this event is instructed in the process, the receiver function as instructed next. The *simple_udp_register* function has a special usage, it registers the UDP port and attaches to a function. In this case, a function named *receiver* is attached (see Listing 4.9). When the process runs, this function is called and the received message is prompt. Moreover, when the on/off message is received the system makes logical comparison and decides to turn on or turn off the green LED (Light Emitting Diode). In the following sub-sections, simulation results of these software and the case studies are given.

```
PROCESS(unicast_receiver,"actuator process");
AUTOSTART_PROCESSES(&unicast_receiver);

static void receiver (struct simple_udp_connection *c, const uip_ipaddr_t *sender_addr, uint16_t sender_port, const
uip_ipaddr_t *receiver_addr, uint16_t receiver_port, const uint8_t *data, uint16_t datalen)
{
    char *text=0;
    text=(char*)data;
    printf("Data received from ");
    uip_debug_ipaddr_print(sender_addr);
    printf(" on port %d from port %d with length %d: %s\n",receiver_port, sender_port, datalen, data);
    if(strcmp(text,"on")==0)
```

```

{
leds_on(LED_GREEN);
printf("on");
}
else if(strcmp(text,"off")==0)
{
leds_off(LED_GREEN);
printf("off");
}
data=0;
}
static struct simple_udp_connection unicast_connection;
PROCESS_THREAD(unicast_receiver, ev, data)
{
PROCESS_BEGIN();
simple_udp_register(&unicast_connection, 8888, NULL, 8888, receiver);
while(1)
{
PROCESS_WAIT_EVENT();
}
PROCESS_END();
}

```

Listing 4.9 Actuator node's software.

4.3.2. Simulation of Smart Lighting System

As the second case study, the smart lighting system is designed, simulated, and implemented. Node 1 is the sink node, node 2 is the client node and node 3 is the actuator node. The server node receives the client's messages. The client node measures light intensity and sends an on/off message to the actuator node according to light intensity value. The actuator node turns on or turns off the light according to the received message. The green LED of the actuator node is used to represent light automation. Figure 4.7 describes the simulation process.

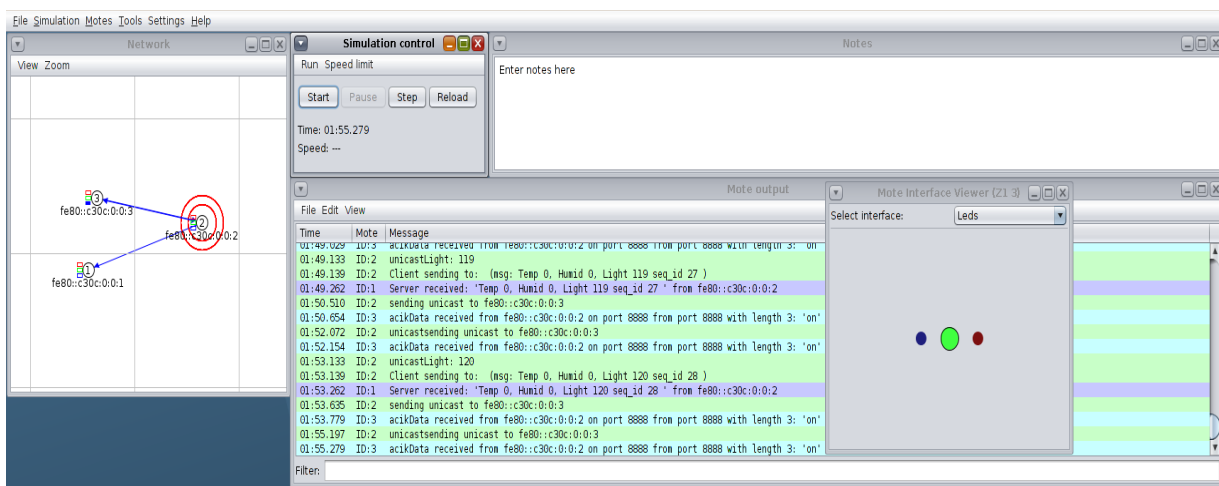


Figure 4.7 Client Node sends messages to Actuator and Server Nodes.

The client node sends light intensity data to the server node and an on/off message to the actuator node every hour. If the light intensity is above a certain set point it sends on message and if it is not, then it sends off message. On the right

side of figure 4.7, the state of the LEDs is shown. The client node sends a unicast message to the actuator node using port number 8888. Making a unicast is beneficial for a wide network. Because there might be a lot of client nodes to sample light intensity and a few of actuator nodes. Since an actuator node can control the whole area's lighting, unicasting is beneficial to activate a specific area's lighting. Moreover, the client node does not have to broadcast the message. Therefore, it can save battery power by just unicasting to the actuator. Power consumption and saving battery power are important criteria that have to be considered in the WSNs and IoT based systems. The system lifetime must be analyzed before deployment. This is mentioned in Section 10.

4.4. Modeling Approach of Protosocket feature of Contiki for TCP Communication

In this sub-section, modeling the Protosocket feature of the Contiki operating system is explained. In the Contiki operating system TCP communication is made using protosockets. The protosocket feature is modeled and their code generation rules are written. To model this feature, the following code snippet is used. Listing 4.10 shows a simple protosocket server to establish a TCP connection.

```
static struct psock ps;
static uint8_t buffer[100];

PT_THREAD(P_T_connection(struct psock *p))
{
    PSOCK_BEGIN(p);

    PSOCK_READTO(p, "\n");
    printf("Received: %s \n", buffer);

    // PSOCK_SEND_STR(p, " Reply Message Here \r\n");

    PSOCK_CLOSE(p);
    PSOCK_END(p);
}
...
PSOCK_INIT(&ps, buffer, sizeof(buffer));
```

Listing 4.10 TCP server side's code using Protosocket.

A protosocket always requires a protothread to run (ContikiOS GitHub, 2006). Firstly, a special protothread for protosocket is defined using *PT_THREAD* macro. Secondly, the beginning and ending of protosocket are defined. *PSOCK_READTO* macro reads the incoming message, and since this protosocket is initialized with a buffer, the incoming message is saved to this buffer. If the server should send a reply message, then *PSOCK_SEND_STR* macro is used with the necessary parameters. To close the TCP connection *PSOCK_CLOSE* macro is used. Lastly, the whole protosocket can be ended with *PSOCK_END* macro. It should be noted that, if the connection should not be closed, then *PSOCK_CLOSE* macro must

not be generated. Even from this code snippet, some code generation rules can be extracted. The buffer size should be inputted from the designer. If the server must send a reply message, then *PSOCK_SEND* must be generated and the designer must input the reply message. If the connection must not be closed, then the *PSOCK_CLOSE* macro should be generated. Moreover, to initialize a protosocket *PSOCK_INIT* macro must be used with this protosocket and a buffer. It should be known that using a TCP connection instead of the UDP connection has advantages and disadvantages. TCP connection establishes a reliable connection. However, because of retransmissions and ACKs, it consumes more battery power compared to a UDP connection. It should be used for the event-critic tasks, such as controlling an emergency door and triggering an alarm (Wagenknecht et al., 2009). Since it leads to more battery power consumption, it can result in problems such as the bottleneck, the first node die, and down of the network. In Section 10, these problems are discussed and explained in detail. Since the client-side protosocket code generation rules are nearly the same as the server-side, only the server side's approach is mentioned.

4.5. RIOT Operating System

RiotOS focuses on program low-power, memory-constrained and wireless Internet of Things devices (Baccelli et al, 2013). It has full support for real-time, multi-thread and pure C based applications as well as considering modularity. Unlike an event-based Contiki operating system, RIOT is a real-time operating system designed for source limited devices. It reduces development overhead and supports devices that are generally used in the Internet of Things based systems. It has developed considering multi-threading, real-time, and uniform API access capabilities, with fulfilling abstraction from the hardware. Riot has a full IPv6 network stack including the latest standards like RPL, TCP, and UDP (Bacceli et al, 2018). In this thesis study, model-based development is made mostly on RIOT's TCP, UDP, and CoAP protocols. Moreover, IPv4 can also be used with TCP and UDP protocols. This feature and related target codes are added in the model to use IPv4 with TCP and UDP. The RIOT operating system is selected, because, it has big substructure, hardware, and community support.

4.5.1. ESP32-WROOM-32 Microcontroller

ESP32-WROOM-32 is a 32-bit, low-cost and dual-core microcontroller. It has integrated Wi-Fi and Bluetooth modules that target a wide variety of applications ranging from low-power sensor networks to the most demanding tasks (Espressif Systems, 2019). It is preferred in various domains such as smart farming, smart home automation, and indoor-positioning systems (Karimpour et al., 2019).

For the implementation of case studies, ESP32 microcontrollers are used. It is mostly ported to the RIOT operating system and still in development. When the ESP-Mesh library is integrated into the RIOT operating system, creating a multi-hop network with ESP32s will be present. In below two case studies are mentioned and they are used to evaluate the DSML4Riot.

4.5.2. Modeling Apporach for Server Node

This sub-section modeling approach for the server side is explained. To achieve this the semantics of the Riot examples are studied and two case studies are implemented. Unlike the Contiki operating system's protothread feature, Riot operating system has a classical C-type main thread feature. In other words, a Riot application must have the main function, even to initialize other threads. Listing 4.11 represents a simple UDP server. In the main function, the initialization of necessary variables is made. At first, *SOCK_IPV6_EP_ANY* is used to bind the device to an IPv6 address. Next, a UDP socket variable is declared and the listen port is defined as 12345. So far, it can be seen that to bind an IPv6 address *SOCK_IPV6_EP_ANY* and to bind an IPv4 address *SOCK_IPV4_EP_ANY* must be used to a type of *sock_udp_ep_t* variable (RiotOS Docs., 2019). Therefore, in the model, the designer can select either IPv6 or IPv4. Therefore, to generate the necessary definition, a rule that generates these definitions must be written. Moreover, when the designer inputs a port number, this must be the initialization value of the *local.port* variable. Since the rest of the code are the functions that use these variables, so they can be static text not a code generation rule. In the Listing 4.11 a code excerpt from the server side is given to intensify the given definitions.

```
int main(void){
    sock_udp_ep_t local = SOCK_IPV6_EP_ANY;
    sock_udp_t sock;
    local.port = 12345;
    if (sock_udp_create(&sock, &local, NULL, 0) < 0)
    {
        puts("Error creating UDP sock");
        return 1;
    }
    while (1) {
        sock_udp_ep_t remote;
        ssize_t ret;
        if ((ret = sock_udp_recv(&sock, buf, sizeof(buf), NULL, &remote)) >= 0)
        {
            if (sock_udp_send(&sock, buf, ret, &remote) < 0)
            {
                puts("Error on reply");
            }
        }
    }
    return 0;
}
```

Listing 4.11 A code excerpt from the Server side's UDP.

The UDP server's code as instructed in the main function. However, to provide modular code generation, it must be instructed in a thread function. In this

way, each task can be instructed in a thread. Otherwise, if UDP and TCP codes were in the same thread, because of the while loops that are used to listen to the incoming packets would block to the instruction of the other protocol. Due to the endless loops, code generation of these protocols must be made in separate threads. Following figure 4.8 represents a node that has a UDP protocol and a thread named *Experiment_Thread*. Moreover, this thread has a timer.

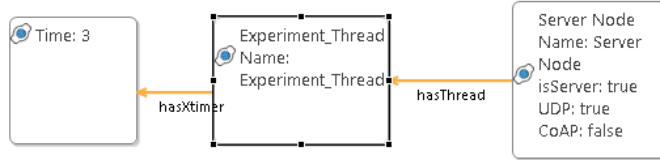


Figure 4.8 A Server Node that has a thread and a thread that has a Timer.

To use the UDP or TCP protocol, the designer does not have to define a thread in the model. The code generation rules are written to generate threads for UDP and TCP. In sub-section 5.2 it is explained. As figure 4.8 describes, server nodes' UDP attribute is true, so the above code in a thread is generated. Moreover, a thread named *Experiment_Thread* is generated as another thread with including *xtimer_sleep(3)* function. However, the above code also requires a *Socket* element to define the server node is going to use IPv6/IPv4 and other parameters such as message, listening, and outgoing port. In section 5.2, it is mentioned in detail.

4.5.3. Modeling Approach for Client Node

This sub-section modeling approach for the client side is explained. The client node has a similar implementation code with the server node. In addition to the client-side code, the IP address of the server node and the message to be sent must be inputted.

```
uint8_t buf[7];
int main(void){
sock_udp_ep_t local = SOCK_IPV4_EP_ANY;
sock_udp_t sock, local.port = 12346;
if (sock_udp_create(&sock, &local, NULL, 0) < 0) {
puts("Error creating UDP sock");
return 1; }
while (1) {
sock_udp_ep_t remote = { .family = AF_INET };
ssize_t res, remote.port = 12345;
ipv4_addr_from_str(ipv4_addr_t *)&remote.addr.ipv4,"255.255.255.0");
if (sock_udp_send(&sock, "Hello!", sizeof("Hello!"), &remote) < 0) {
puts("Error sending message");
sock_udp_close(&sock);
return 1;}
if ((res = sock_udp_rcv(&sock, buf, sizeof(buf), 1 * US_PER_SEC,NULL)) < 0)
{
if (res == -ETIMEDOUT) {
puts("Timed out");}
else {
```

```

puts("Error receiving message");
}
else {
printf("Received message: \n");
for (int i = 0; i < res; i++) {
printf("%c", buf[i]);
printf("\n");
}
xtimer_sleep(1);return 0;
}

```

Listing 4.12 A code snippet of a UDP client that uses IPv4.

Listing 4.12 shows a UDP client that uses IPv4. It has nearly the same functions as the server node's code mentioned in sub-section 4.5.2. However, it has more logical controls, functions, and parameters such as packet-timeout control, socket closing, and server's IP address. It sends UDP packets to the server's port number 12346. The socket family is generated as *AF_INET*, if the IPv4 is selected and it is generated as *AF_INET6*, if the IPv6 is selected. The function *ipv4_addr_from_str* converts an IP address type of string to the type of *ipv4_addr_t*. The same conversion is made for the IPv6 using *ipv6_addr_from_str* function. In the endless while loop, it sends *Hello!* message to the server node. If the message cannot be sent successfully then it closes the connection and erases the socket struct from memory. It is important to close the socket and exit from the thread. Because, if the message cannot be sent to the server node and the connection is not closed, then it will try to resend the message. As a result, the antenna would be used to resend the message of an unsuccessful connection and the battery power would be wasted.

The code generation rules and the semantics of the code are written to provide the best code artifacts for the designer. The generated codes are tested during the implementation of the case studies. It is aimed to require less input from the designer and to provide an executable, functional, and syntactically correct code. In section 5.2 modeling of the UDP and TCP protocols is explained.

4.5.4. Case Study: Smart Irrigation System

The first case study is the smart irrigation system. It works using UDP protocol with two nodes that are a server node and a client node. The client node uses a digital GPIO pin which has a moisture sensor connected. Moisture sensor's digital output pin is used and connected to the client node's GPIO pin. When moisture level drops to a certain level, it generates a digital HIGH voltage level then the client node sends a message to the server. If the received message is *Dry* word, then the server activates the GPIO pin to turn on the water motor for irrigation of the plant. The water motor runs for 3 seconds then the ESP32's GPIO pin is set as low to stop the motor. In the below system steps are listed.

- The moisture level of the soil is sampled by GPIO pin 15 of the ESP32.
- GPIO pin 2 toggles just to inform the person in charge of the system is running.
- If the moisture level is below a set point (this point is set by a variable resistor on the moisture sensor), in other words when the soil gets dry, moisture sensor sends a digital HIGH value to pin 15.
- Then, a string message *Dry* is sent to the server node.
- The server always listens to the necessary port. If the message named *Dry* is received, then it activates the GPIO 5 to run the water motor for 3 seconds to irrigate the soil.

At below figure 4.9 represents the case study. On the left side, the client node with a moisture sensor is shown. The blue board is an analog-to-digital converter. It sends the digital output signal to the *GPIO* 15. The sensitivity of the analog-to-digital converter can be set by a variable resistor on the blue board. On the right side, the server node with a water motor is represented. The water motor can be controlled by the digital outputs at 3.3 volts. To visualize better the case study, a plant is used and the moisture sensor is buried to the soil. The ESP32s are powered by USB (Universal Serial Bus) port of a computer.

This case study is beneficial for smart farming and arboriculture. It also reduces the water consumption. The water is used only when it is required. It is important to bury the moisture sensor and water hose to the closest point of the plant. In this way, the water can reach to the roots of the plant. The amount of the water can be set by modifying the irrigation time. However, during the implementation of this case study three seconds was enough for the plant.

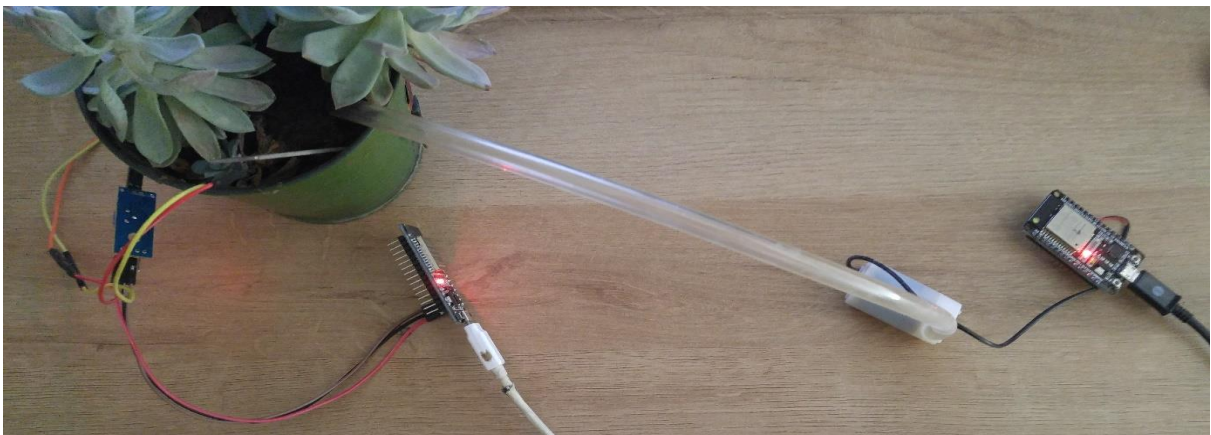


Figure 4.9 Smart Irrigation System and ESP32s.

4.5.5. Case Study: Smart Lighting System

As a second case study a light control system is designed and implemented. Most of the code is generated by DSML4RIOT and delta codes are added to finalize the software part. In the hardware part, client ESP32 is programmed to measure light intensity using a light-dependent resistor. A comparator circuit is built using a 4.7 kilo-ohm resistor and a 5 kilo-ohm light-dependent resistor to sample light intensity in a room. When light intensity drops below a certain level, the client sends a message to server node to open the room light. To sample the environment light intensity in the client node, the server node must close the room light not to belie the sampling results in the client side. Therefore, before sampling environmental light intensity, the client node must send a message to the server node to close the light. Next, the client node samples the environment light intensity and decides to open either the room light or to close it. Below figure 4.10 represents the smart lighting system case study.

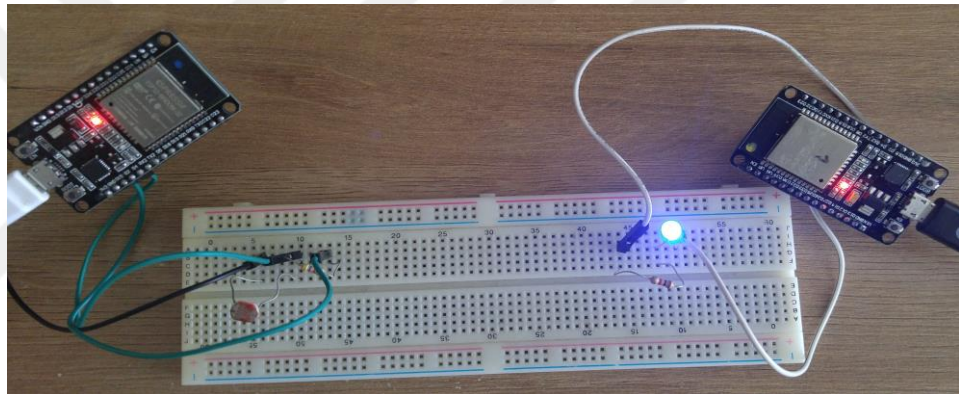


Figure 4.10 Smart Lighting System and ESP32s.

On the left side, the client node with a comparator circuit is shown. It has a 4.7 kilo-ohm and light-dependent resistors. On the right, the server node with a light-emitting diode is represented.

4.6. Modeling Approach for TCP Communication

In this sub-section modeling approach of TCP, sock API is mentioned. The TCP sock API has a nearly similar implementation with UDP sock API mentioned in the sub-sections 4.5.2 and 4.5.3. The similar API functions between these protocols also provide convenience for the modeling and writing code generation rules for the operating system. The API functions are *sock_tcp_listen*, *sock_tcp_accept*, *sock_tcp_read*, *sock_tcp_disconnect*, and *sock_tcp_stop_listen*. Firstly, a queue for the incoming sockets is created. This queue is defined to be a listen queue for the incoming socket objects. Moreover, a buffer is created to store received messages from the incoming sockets. Secondly, Port number 4646 is

wanted to listen for the incoming connections that have IPv6. In the sub-section, 4.5.2 configuration of IPv6 and IPv4 is mentioned. Using *local.port=4646* and *sock_tcp_ep_t local=SOCK_IPV6_EP_ANY*, the server is set to listen port 4646 for connections that has an IPv6 address. Next, a listen queue is created. This should be clear that the *sock_queue* variable is defined to store incoming connections' sockets and the *queue* variable is defined to store incoming connections. Lastly, the *sock_tcp_listen* function is called to initialize the listening process. The function's return value is checked whether any problem occurs, then it stops to run until the problem is solved (see Listing 4.13).

```
static sock_tcp_t sock_queue, uint8_t buff[128];
int main(void){
sock_tcp_ep_t local = SOCK_IPV6_EP_ANY, sock_tcp_queue_t queue, local.port = 4646;
if (sock_tcp_listen(&queue, &local, &sock_queue, 1, 0) < 0) {
puts("Error creating listening queue");
while (1) { sock_tcp_t *sock;
if (sock_tcp_accept(&queue, &sock, SOCK_NO_TIMEOUT) < 0){
puts("Error accepting new sock");} else{
int read_res = 0;
puts("Reading data");
while (read_res >= 0) {
read_res = sock_tcp_read(sock, &buff, sizeof(buff), SOCK_NO_TIMEOUT);
if (read_res < 0){
puts("Disconnected");
break;} else {
printf("Read: \n");
for (int i = 0; i < read_res; i++)
{printf("%c", buff[i]);}
if ((write_res = sock_tcp_write(sock, &buff, read_res)) < 0) {
puts("Errored on write, finished server loop"); break;}}}
sock_tcp_disconnect(sock);}
sock_tcp_stop_listen(&queue); return 0;}
```

Listing 4.13 The code that is used to model TCP communication.

In an endless loop, a socket variable is defined and the program waits for an incoming connection using *sock_tcp_accept* function. If the timeout parameter is required to set, then an integer value can be sent as a parameter instead of *SOCK_NO_TIMEOUT* macro. When a connection is successfully established with a client, the socket object that is sent by the client is tried to read using the *sock_tcp_read* function. The incoming message is stored in the buffer named *buf*. The timeout period also can be set as a parameter in this function. If any error occurs during the reading of the incoming message, then the loop is broken and the socket is disconnected. Otherwise, the message that is stored in the buffer is printed and it writes back to the socket that is established the connection. If any error occurs during the writing back to the socket, then the loop is broken. Lastly, if the endless loop is broken, the listening queue must be stopped to listen to the connections. Using *sock_tcp_stop_listen* function it is stopped, appropriately. So far, it is mentioned that to model the TCP sock API, the required parameters such as IP address version, port number, and IP address of the server can be inputted by the designer in the DSML4Riot. In the sub-section 5.4, code generation rules for TCP sock API is explained in detail.

5. PLATFORM-SPECIFIC MODELS & CODE GENERATION RULES

In this section code generation of DSML4Riot and DSML4Contiki are mentioned and a comparison between ContikiOS and RiotOS is made. Code generation rules' snippets are given. In the Contiki operating system, each thread is tried to run equally and application codes are written in process threads. Therefore, code generation rules are written considering this condition while developing DSML4Contiki. During the development of DSML4Riot same approach is used. The code generation rules are written to run each defined thread, equally. Network features such as TCP and UDP are defined in each special thread. Therefore, a modular code generation is also achieved. In RiotOS, a priority-based scheduling algorithm runs to schedule threads according to their priority level. In DSML4Riot, thread priorities can be set by the designer. Each thread including main is generated to run in the same priority as default. Each thread runs exactly by the scheduler if they all have the same priority. However, if the designer desires to set different priority levels and to run these threads especially, then it must be done by adding Riot's thread functions (delta codes) to handle thread sequence. To differentiate the application codes from the Acceleo codes, the Acceleo codes are bolded.

5.1. Code Generation Rules for ContikiOS

For each Contiki node element, a C file for application code is created. In implemented case studies, files for a server, a client, and an actuator (it is also a server node since it receives messages) are created. To specify this separation, a Boolean attribute is defined in the node element. This rule is shown in Listing 5.1. If the designer activates the *Sink* feature, code rules for a server node are generated.

<pre> [for(n:Node aIoTSystem.platform.node)] [if not(n.Sink)] [file (n.Name.concat('.c'), false, 'UTF-8')] ... [/file][/for] </pre>

Listing 5.1 Rule to create code files for Sink and Source nodes.

If it is not activated, then the client node's codes are generated. To generate codes of UDP and/or TCP the Boolean flags must be activated of a *process thread*. To create a reply message when a message is received, *tcpip_event* element in the model must be added and connected to the server node. Since Contiki is an event-based operating system, if any UDP packet or TCP packet is received, an event named *tcpip_event* is generated by the operating system. Therefore, sending the reply message is bound to this event. Using these rules, the required files and their architectural code are generated automatically for different components of the IoT system. These artifacts should be completed adding the user's delta code to have a

fully functional system. Figure 5.1 shows a process that TCP and UDP features are activated and Protosocket feature is not activated. Moreover, it has a *Sensor* element named Sensor.

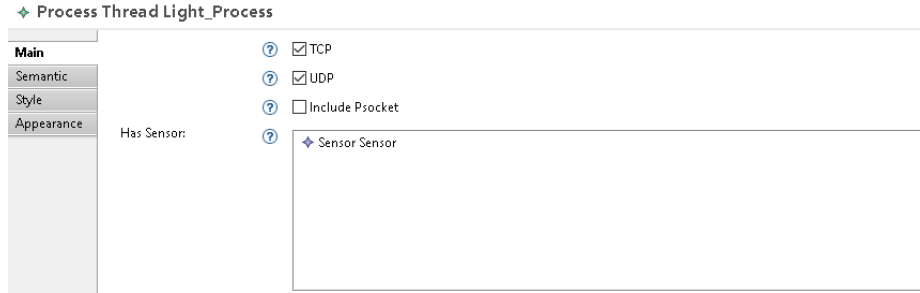


Figure 5.1 A process that TCP and UDP flags are activated.

If the UDP attribute is set to *true* by marking the boxes, then Acceleo generates codes of UDP in that process thread. The same is valid for TCP. Moreover, there is an attribute named *Multihop* which indicates that the network operates either in single-hop or in multi-hop topology. All timer features of the Contiki operating system are also modeled including *Ctimer*, *Rtimer*, *Stimer*, and *Etimer*. Their codes are generated when these elements are used. *Ctimer* calls a specific function specified in the *callfunction* attribute, and this function uses the *Data* attribute as a parameter. *Rtimer* runs for the specified duration for real-time applications. *Stimer* and *Etimer* elements are modeled to take two arguments which are name and duration attributes. Finally, *LocalPort* attribute in the *Client_Connetion* and *Server_Connection* elements are added to define communication ports between these elements. The following Listing 5.2 shows the code generation rules for Contiki's timers. The codes are generated using a simple logical control. If the designer enters any name for the timer, in other words, the name attribute is not empty then Acceleo generates timer's code with a given name.

```
[if (p.hasEtimer.Name.toString() <> 'invalid')]
    static struct etimer [p.hasEtimer.Name/];
[/if]

[if (p.hasStimer.Name.toString() <> 'invalid')]
    static struct stimer [p.hasStimer.Name/];
[/if]

[if (p.hasCtimer.Name.toString() <> 'invalid')]
    static struct ctimer [p.hasCtimer.Name/];
[/if]

[if (p.hasRtimer.Name.toString() <> 'invalid')]
    static struct rtimer [p.hasRtimer.Name/];
[/if]

[if (p.hasTimer.Name.toString() <> 'invalid')]
    static struct timer [p.hasTimer.Name/];
[/if]

[if (p.hasEtimer.Name.toString() <> 'invalid')]
    etimer_set(&[p.hasEtimer.Name/],[p.hasEtimer.Period/]*CLOCK_SECOND);
```

```

[/if]

[if (p.hasStimer.Name.toString()<>'invalid')]
    stimer_set(&[p.hasStimer.Name/],[p.hasStimer.Period/]*CLOCK_SECOND);
[/if]

[if (p.hasCtimer.Name.toString()<>'invalid')]
    ctimer_set(&[p.hasCtimer.Name/],[p.hasCtimer.Period/]*CLOCK_SECOND);
[/if]
[if (p.hasRtimer.Name.toString()<>'invalid')]
    rtimer_set(&[p.hasRtimer.Name/],[p.hasRtimer.Period/]*CLOCK_SECOND);
[/if]

[if (p.hasTimer.Name.toString()<>'invalid')]
    timer_set(&[p.hasTimer.Name/],[p.hasTimer.Interval/]*CLOCK_SECOND);
[/if]

```

Listing 5.2 Code generation rules to generate timer codes.

As mentioned, to send a reply to an incoming message, a *TCPIP_Event* element must be added to the model. Since Contiki is an event-based operating system, if any UDP or TCP packet is received, an event named *tcipip_event* is generated by the operating system. Therefore, sending the reply message is bound to this event. When the designer activates *send reply* flag the following code rule runs in Listing 5.3.

```

--Send reply according to selection of UDP/TCP connection
[let a:String=n.processThread.handlesEvent->
    select(event|event.eClass().name="TCPIP_Event")-> at(i).eGet('replyMessage')]
[if(n.UDP)] //NOTE:Add Server Connection to the model
    PRINTF("DATA sending reply\n");
    uip_ipaddr_copy(&[p.hasServerConnection.Name/]-> ripaddr, &UIP_IP_BUF->srcipaddr);
    uip_udp_packet_send([p.hasServerConnection.Name/],[a/], sizeof("[a/]"));
    uip_create_unspecified(&[p.hasServerConnection.Name/]->ripaddr);
[/if]

[if(n.TCP)] //TCPIP Reply
    sprintf(buf, " [a/] "(%d)",1);
    PRINTF(buf);
    PRINTF("\n\r");
    uip_send(buf, strlen(buf));
[/if]
[/let]

```

Listing 5.3 Code generation rules to send reply message.

During the implementation of the case studies to evaluate DSML4Contiki, in the sink node side, a few lines of delta codes are required. When the generated code is completed, the sink node fulfilled the requirements to achieve case studies. Since code generation rules are written to fulfill basic requirements such as receiving data packets, printing the IP address of the sender, and sending a reply to the client, the generated code for the sink node required a few lines of delta codes.

Using these rules, the required files and their architectural code are generated automatically for each component of the IoT system. Code generation rules are coded to generate an executable file. All syntax errors are aimed to be eliminated. Some guidance messages are added as comments. These comment messages are generated in these target codes files.

5.2. Code Generation Rules for RiotOS

In this sub-section code generation, rules for Riot operating system are explained. The code snippets from code generation rules are given. The explanation is started from the network stack of the Riot operating system. Code generation rules belong to TCP, UDP, and IP features of RiotOS are mentioned. These rules are used to implement two case studies. The rules that are not covered in the case studies are explained in additional sub-section. The following Listing 5.4 shows the code that will be generated when the TCP, UDP flags are activated, and version of the IP address is selected.

```
[if (n.socketR.isIPv4)]
#include "net/ipv4/addr.h" //for udp and tcp
[/if]

[if (n.socketR.isIPv6)]
#include "net/ipv6/addr.h" // for udp and tcp
[/if]

[if (n.TCP) or (n.UDP)]
#include "net/af.h" // for udp and tcp
[/if]

[ if (n.UDP) ]
# include "net/protnum.h" // for udp
# include "net/sock/udp.h" // for udp
[/if]

[if (n.TCP)]
# include "net/sock/tcp.h" // for tcp
[/if]
```

Listing 5.4 Code rules to generate libraries.

When the designer selects TCP and/or UDP communication protocols, specific threads and thread functions are generated. Therefore, additional threads are not required to be defined for UDP and TCP in the modeling environment. Moreover, these threads are generated at the same priority level. The priorities should be equal with the main thread, so each thread guarantees to be run. Threads for UDP and TCP protocols are defined to be started, when the device is powered on. A thread must be defined in the main function. Therefore, code generation rules are made according to this rule. Unlike ContikiOS's protothread feature, threads in Riot operating system must be defined in the main thread to be used. At the above listing 5.4, the code snippet represents code generation rules for UDP and TCP libraries. According to the designer's model, necessary libraries are included in the generated code. For example, if the designer designs a TCP based application, TCP library, and other related libraries are included in the generated file. As mentioned, the designer can use either IPv6 or IPv4. If the designer selects both IPv6 and IPv4, a constraint rule which is written to warn the designer pops-up to direct the designer to select only one of them. In below Listing 5.5, a snippet of code generation rules of socket element is given. If the model has a socket element, the following rules

are instructed to generate codes. Figure 5.2 describes simple a server-client model that has a socket element.

```

[if (socket.isIPv6)]
sock_udp_ep_t local = SOCK_IPV6_EP_ANY
[/if]

[if (socket.isIPv4)]
sock_udp_ep_t local = SOCK_IPV4_EP_ANY ;
[/if]

[if (socket.isIPv6)]
sock_udp_ep_t remote = { .family = AF_INET6 } ;
[/if]

[ if(socket.isIPv4)]
sock_udp_ep_t remote = { .family=AF_INET} ;
[/if]

```

Listing 5.5 Code generation rules to generate necessary parameters for IPv4 and IPv6.

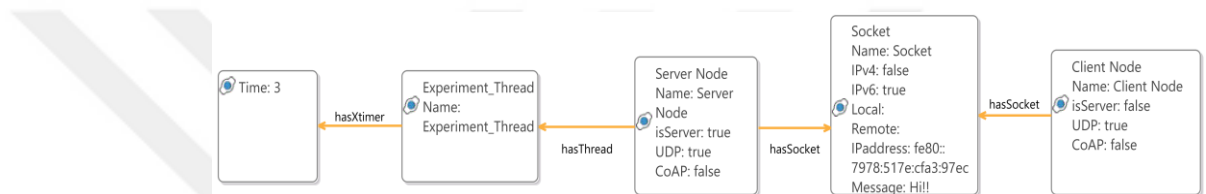


Figure 5.2 A Simple Server-Client Modeling in DSML4Riot.

A thread named *Experiment_Thread* is added to the model, so the following code generation rules are instructed and the name of the thread will be placed where *[thread.Name/]* is written. A for loop runs as many as thread elements are added to the model. A thread must be yielded when the task is finished. Therefore, at the last line of the thread *thread_yield* function is called to let the other threads run. Compared to the Contiki operating system, Riot operating system has less complex thread structure. Because, since it is a real-time operating system, it does not require an event and a timer controls in threads. In below, Listing 5.6 shows code generation rules for a thread.

```

[for (thread:Thread|n.threadR)]
char [thread.Name/]_stack['[THREAD_STACKSIZE_MAIN]'];
void *[thread.Name/](void *arg)
{ (void) arg;
  while(1)
  { .. //delta codes here
    thread_yield(); }
  return NULL; }[/for]

```

Listing 5.6 Code rule to generate threads.

The designer does not have to define additional threads in the model for UDP and TCP protocols. During the implementation of the code generation rules special threads for these protocols are defined, so special threads are generated to ease the designer's work. To run all threads equally, when the main thread is initialized

THREAD_CREATE_WOUT_YIELD parameter must be given to each thread. Below Listing 5.7 shows, if the designer activates TCP protocol then a thread, a stack for this thread, and a thread function are generated.

```
[if (n.TCP)]
char tcp_stack[THREAD_STACKSIZE_MAIN]';
[/if]

[for (thread:Thread|n.threadR)]

thread_create([thread.Name/]_stack, sizeof([thread.Name/]_stack),
THREAD_PRIORITY_MAIN - [thread.Priority/], THREAD_CREATE_WOUT_YIELD,
[thread.Name/], NULL, "[thread.Name/]");
[/for]
```

Listing 5.7 Code generation rule for thread stack, thread function, and thread parameters.

The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with source-limited devices and battery constrained networks in the Internet of Things. The protocol is designed for machine-to-machine (M2M) applications and it works using UDP protocol (Shelby et al., 2004). In DSML4Riot, this feature of the Riot operating system is modeled and code generation rules are written.

```
[if(n.CoAp)]
#include"net/nanocoap.h"
[for(coap:CoAPparams|n.coapparamsR)]
staticsize_t_[coap.functionName/]_handler(coap_pkt_t*pkt,
uint8_t*buf,size_tlen,void*context){
(void)context;
[for(gpio:GPIO|n.coapparamsR.gpioparams)]
externgpio_tg[gpio.Number/];
gpio_toggle(g[gpio.Number/]);
[/for]
returncoap_reply_simple(pkt,COAP_CODE_205,buf,len,
COAP_FORMAT_TEXT,(uint8_t*)RIOT_BOARD,strlen(RIOT_BOARD));}
[/for]
const coap_resource_tcoap_resources['[]']={
COAP_WELL_KNOWN_CORE_DEFAULT_HANDLER,
[for(coap:CoAPparams|n.coapparamsR)]
{"[/coap.functionName/]",[if(coap.isPost)]COAP_GET[else]
COAP_POST[/if],_[coap.functionName/]_handler,NULL},{"/[coap.functionName/]",[if(coap.isPost)]C
OAP_GET[else]
COAP_POST[/if],_[coap.functionName/]_handler,NULL}
[if(n.coapparamsR->size()>i)],[/if]
[/for]
};
```

Listing 5.8 Code generation rules for CoAP.

In the above Listing 5.8, a code snippet from CoAP code generation rules is shown. If the designer decides to use CoAP protocol, firstly, the necessary library is included to target code, then CoAP functions that are defined using *coapparams* are generated. If the designer wants to control a *GPIO* using *CoAP* protocol, then also *GPIO* configuration rules are generated. Moreover, a *CoAP* function (request URI) can be generated as either *POST* or *GET* request. The designer inputs a function name that is the address of the request and decides the request is a *POST*

or *GET*. Moreover, the designer can design a model that controls a *GPIO* using *CoAP*.

In the following Listing 5.9 code generation rules of the client side of TCP sock API are given. As mentioned in sub-section 4.6 the required parameters are inputted by the designer in the DSML4Riot using socket element. In the modeling environment of the DSML4Riot, a node can use UDP and TCP protocols. Using logical controls, the codes are generated and using the variables that are decided by the designer is placed. For example, the *[socket.RemotePortNumber/]* represents the socket element's remote port number attribute when the designer has inputted an integer, that integer is generated where the *[socket.RemotePortNumber/]* is written.

```
[if (n.TCP)]
uint8_t buf[128]/];
void *tcp_thread(void){ while(1) {
    int res;
[if (socket.isIPv6)]
    sock_tcp_ep_t remote = SOCK_IPV6_EP_ANY;
[/if]
[if (socket.isIPv4)]
    sock_tcp_ep_t remote = SOCK_IPV4_EP_ANY;
[/if]
    remote.port = [socket.RemotePortNumber/];
[if (socket.isIPv6)]
    ipv6_addr_from_str((ipv6_addr_t *)&remote.addr, "[socket.IPAddress/]");
[/if]
[if (socket.isIPv4)]
    ipv4_addr_from_str((ipv4_addr_t *)&remote.addr, "[socket.IPAddress/]");
[/if]
    static sock_tcp_t [socket.Name/];
    if (sock_tcp_connect(&[socket.Name/], &remote, 0, 0) < 0) {
        puts("Socket problem or Server is offline");
        return NULL;
    }
    puts("Sending \"[socket.Message/]!\");
    if ((res = sock_tcp_write(&[socket.Name/], "[socket.Message/]", sizeof("[socket.Message/]"))) < 0) {
        puts("Errored on write");
    }
    else {
        if ((res = sock_tcp_read(&[socket.Name/], &buf, sizeof(buf), SOCK_NO_TIMEOUT)) < 0) {
            puts("Disconnected");
        }
        printf("Read: \");
        for (int i = 0; i < res; i++)
        {
            printf("%c", buf[i]/]);
        }
        puts("\");
        sock_tcp_disconnect(&[socket.Name/]);
    }
[/if]
```

Listing 5.9 Code generation rule to establish a TCP communication

Moreover, the client code requires the IP address of the server, the message to be sent, and, socket variable's name. In the following sub-section code generation rules that are not covered by the case studies are explained.

5.3. Code Generation Rules for Protosocket feature of ContikiOS

As mentioned in sub-section 4.4, the code generation rules of protosockets are written to establish a TCP connection in the Contiki operating system. Since each protosocket requires a protothread, the outer for loop must generate the codes of the protothread. Firstly, the socket name and buffer name are generated. Secondly, the protothread of the protosocket is generated. After the *PSOCK_BEGIN* macro, the protosocket writes the sensor values to the buffer as well as a data defined by the designer. The sensor values are sent using *PSOCK_SEND_STR* function. It also sends a message that can be inputted by the designer. If any message is received, then it is read using *PSOCK_READTO* function and stored into a buffer. In the Listing 5.10, code generation rules for protosocket are given.

```
[for (pt : PT_Thread | n.ptThreadNode)]
static struct psock [pt.Name/];
static uint8_t buffer_[pt.Name/][100]/;
[/for]

[for (pt : PT_Thread | n.ptThreadNode)]
static int [pt.Name/]_connection(struct psock *[pt.hasPT_ArgumentType.Name/]){
PSOCK_BEGIN([pt.hasPT_ArgumentType.Name/]);

[for (psock : Psock | aIoTSystem.platform.psock->at(i))]

[if (psock.hasOutgoingData.Name.toString()<>'invalid')]
sprintf(buf,"Temp %d, Humid %d, Light %d msg %s \n
",value["0"/],value["1"/],value["2"/],"[psock.hasOutgoingData.Name/]");
[/if]

[/for]

static int seq_id;
printf("Client sending msg using PT: ");
PSOCK_SEND_STR([pt.hasPT_ArgumentType.Name/], buf);
{
PSOCK_READTO([pt.hasPT_ArgumentType.Name/], "\n");
printf("Message: %s", buffer_[pt.Name/]);
}
PSOCK_END([pt.hasPT_ArgumentType.Name/]);
}
[/for]
```

Listing 5.10 Code generation rules for Protosocket feature.

The message that is stored in the buffer is printed. The connection of the protosocket is ended using *PSOCK_END* function. The necessary parameters that are taken from the designer in the model, are replaced in the necessary functions. The user can use many protosockets in the model and these protosockets are generated in each protothreads. The name variables are taken from the designer in the model to separate each protothread. In this way, executable protothreads and protosockets codes can be generated. The designer may modify the generated code and may decide to implement an algorithm in the protothread. During the implementation of the code generation rules, it was aimed to present the designer with an executable and connection establishable output. In the following sub-

section 5.5 additional code generation rules for the Riot operating system are explained.

5.4. Code Generation Rules for Saul feature of RiotOS

Saul is an abstraction layer for sensor and actuator devices. Its purpose is to enable a common interface for several sensors and actuators using common functions and common data structures. A device has to be registered to the SAUL registry. Therefore, devices can be found and used using a few of SAUL functions. A device has to register its name and its type. In this way, any device can be found automatically. In other words, using type or name parameters a device can actively be used without dealing with configuration codes. For modeling purposes, SAUL is a well-suited abstraction layer. Because even the modeling of sensors or actuators of an operating system is a challenging task. Thanks to SAUL's offered common interface, code generation rules were written easily using two parameters and logical comparisons. Following listing 5.11 shows code generation rules for a temperature sensor.

```
[for (saul:SAUL| n.saulR)]
static saul_reg_t* saul_[saul.Name/];
[/for]

[for (saul:SAUL| n.saulR)]
    [if (saul.Temperature)]
    phydat_t phy_[saul.Name/]_temp[i/];
    [/if]
    [if (saul.Temperature)]
    saul_[saul.Name/] = saul_reg_find_type(SAUL_SENSE_TEMP);

    saul_reg_read(saul_[saul.Name/], &phy_[saul.Name/]_temp[i/]); // this contains sensor result
    [/if]
[/for]
```

Listing 5.11 Code generation rules to use a temperature sensor via SAUL

The designer can easily use a sensor device in the system by generating a few lines of code. However, this should be noted that, since Riot is an in-development operating system, pin connections of the hardware must be made according to the device driver library of the Riot operating system. These configurations can be found in the online documentation of the Riot operating system (RiotOS Docs., 2019).

To simplify the expression, only code rules for a temperature sensor is given. The designer may also decide to use a humidity sensor and a light sensor. If the designer has a SAUL element in the design and if the temperature feature is activated, then the above code snippet is instructed. A *phydat* type struct must be used to contain sampling data of the sensor device. Then, this sensor's type must be given to SAUL's function as a parameter. This can be achieved using *saul_reg_find_type* (SAUL_SENSE_TEMP). Then, to read the data from the sensor *saul_read_reg* function is used. As a result, thanks to the SAUL abstraction layer, it is easily modeled and it is served as a design element to the designer.

Compared to the Contiki operating system, Riot operating system has an abstraction layer for sensor and actuator devices. This feature can be beneficial for both the users of the operating system and for the modeling of the operating systems. In the following sub-sections models of the case studies are mentioned. When these models are run by the DSMLs, they trigger the code generation rules related to an element of these models. They instruct the code generation rules that are mentioned so far.

5.5. Models of the Case Studies

Before starting the implementation of the four case studies, their models are designed and target codes are generated for Contiki and Riot operating systems. The necessary delta codes are added to have a fully functional system. The delta codes are given and evaluation results of the DSMLs are mentioned in section 11. The designed models of these case studies are represented by figures in the following sub-sections.

It can be noticed that the models of the DSML4Riot's have fewer elements compared to models of the DSML4Contiki's. This was resulted because of the design choices during the implementation of the meta-models and features of the operating systems. It is mentioned in section 7 in detail.

5.5.1. ContikiOS: Smart Fire Detection System Model

The following figure 5.3 describes the Platform-Specific Model of the smart fire detection system that is introduced in sub-section 4.2. The model is designed in DSML4Contiki. It has two nodes that are a client and a server. Using the modeling approach that is mentioned in sub-sections 4.2.1 and 4.2.2, necessary modeling elements are implemented. Using these elements, the model of the case study is designed.

On the server-side, a process has *TCP_IP_Event* and *Server_Connection* elements. These elements are required to establish a network connection. The server node uses a server connection element and the client node uses a client connection element. In the client-side (source node), two processes are added. The first process measures temperature data and the second process sends this data to the server (sink) node every three seconds. *Measure_Temperature* process has a sensor element named *TemperatureSensor*. During the modeling of a system, it is important to give domain related names to the elements to describe the aimed task. For this purpose, the sensor element is named *TemperatureSensor*. The *Timeout_Event* element checks, if the event timer is expired, then it sends the temperature data to the server (sink) node. The server (sink) node prints the received messages.

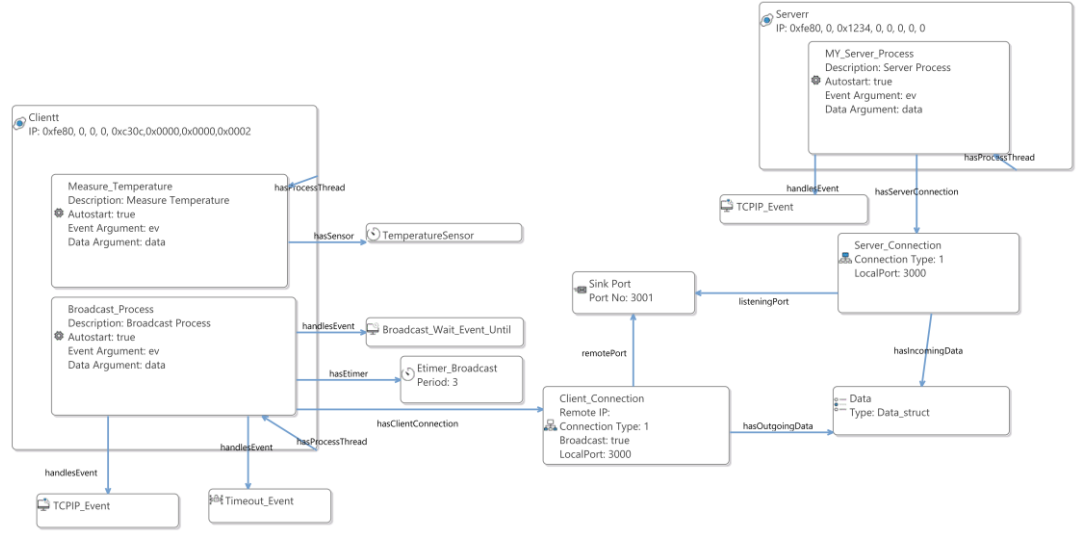


Figure 5.3 Platform-Specific Model of Smart Fire Detection System.

In the following sub-section 5.5.2 the Platform-Specific model of the Smart Lighting System is given and explained. It has nearly the same model elements with the model of the Smart Fire Detection System that is mentioned in this sub-section.

5.5.2. ContikiOS: Smart Lighting System Model

The following figure 5.4 describes the Platform-Specific Model of the smart lighting system that is introduced in sub-section 4.3. The model is designed in DSML4Contiki. It has three nodes that are an actuator, a client, and a server. Using the modeling approach that is mentioned in sub-sections 4.2.1 and 4.2.2, necessary modeling elements are implemented. Using these elements, the model of the case study is designed.

In the model, three nodes are an actuator node, a server node, and a client node. The actuator node has a process that operates as a server. It receives messages from the client node to either open or to close the lights. In sub-section 4.3.2, the actuator node's response to the received messages is shown. The client node sends the sampled light intensity values to the server (sink node). To achieve this, it uses a *Sensor* element. Since the sensor elements can be configured to sample any combination of the temperature, light, and humidity, for the case study's purpose, it is configured to sample light intensity values. Therefore, it sends the light intensity values to the sink node and it sends the on/off messages to the actuator node every 4 seconds. The *Etimer* element describes the message sending period. In the model the designer can add as much as *Node* elements. This model also represents the scale-ability of DSML4Contiki. The multiple *Nodes* also can be added in the DSML4Riot. The rest of the model elements has the same roles that are mentioned in the sub-section 5.5.1.

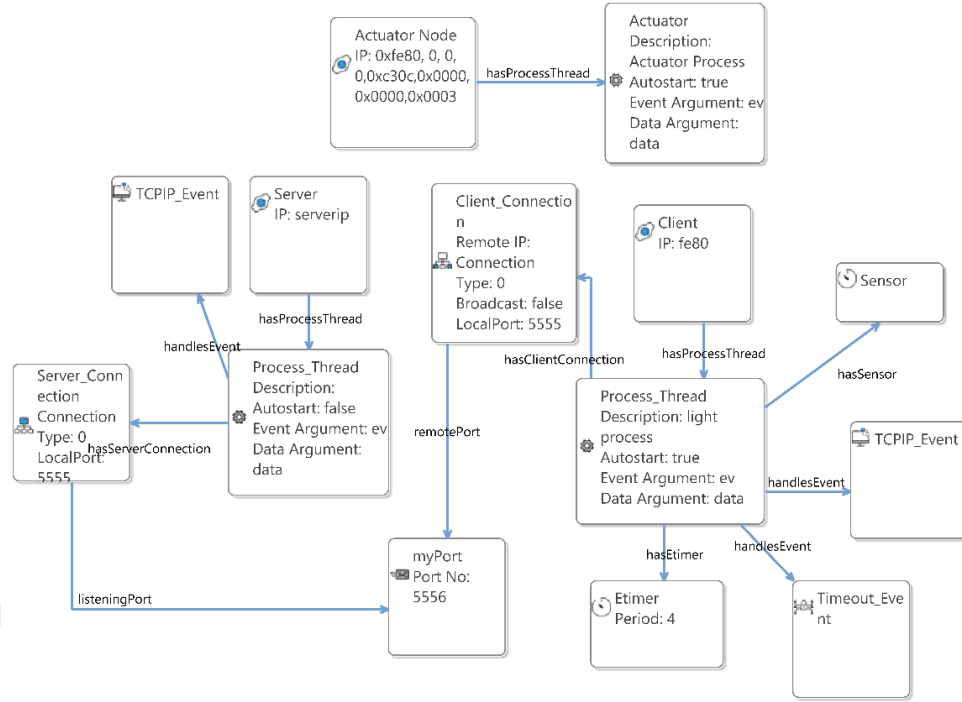


Figure 5.4 Platform-Specific Model of Smart Lighting System.

As a result, to create a system based on the Contiki operating system the above figures can be good references. In both models, sink and source nodes are used and UDP is selected to establish a network. Since Contiki is an event-based operating system, the event elements are added to the model. To sample a sensor data, a process thread element that has a sensor element is used in two models. In the sub-section 5.5.3, models of the two case studies that are used to generate codes for these case studies are given.

5.5.3. RiotOS: Smart Irrigation System Model

The following figure 5.5 describes the Platform-Specific Model of the smart irrigation system that is introduced in sub-section 4.5.4. The model is designed in DSML4Riot. It has two nodes that are a client and a server. Using the modeling approach that is mentioned in sub-sections 4.5.2 and 4.5.3, necessary modeling elements are implemented. Using these elements, the model of the case study is designed.

The server node's and client node's UDP flags are activated and set to true. The client node uses two pins. A *GPIO* element was added to the model to measure the moisture of the soil from the pin 15 and it was defined as input. A *GPIO* element that uses pin 2 was added to the model to indicate the system status.

The server node has one *GPIO* element. It uses to pin 5 to control the water motor according to the received message. The client node and the server node has a *Socket* element to define outgoing and listening ports, IP address, and the

message. Since the server node starts watering when the *Dry* keyword is received, the *Socket* element's message attribute was inputted as *Dry*.

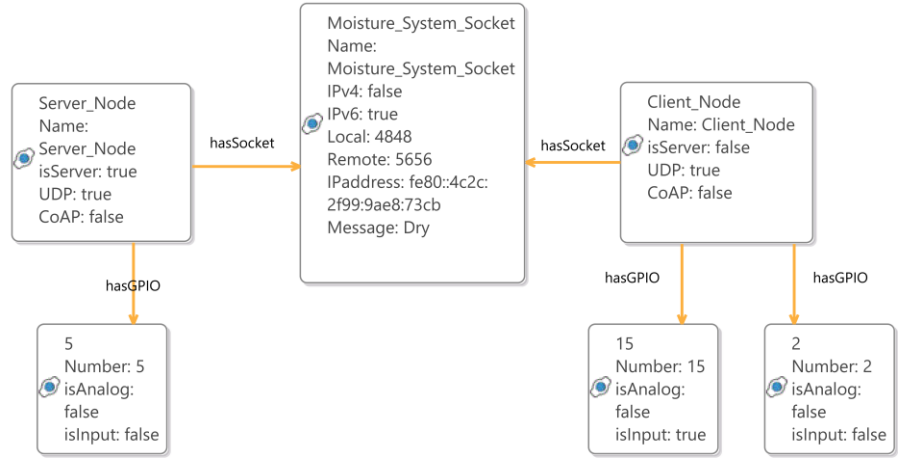


Figure 5.5 Platform-Specific Model of smart irrigation system.

In the following sub-section 5.5.4 the Platform-Specific Model of the smart lighting system is given and explained. It has nearly the same model elements with the model of the Smart Irrigation System that is mentioned in this sub-section.

5.5.4. RiotOS: Smart Lighting System Model

The following figure describes the Platform-Specific Model of the smart lighting system that is introduced in sub-section 4.2.6. The model is designed in DSML4Riot. It has two nodes that are a client and a server. Using the modeling approach that is mentioned in sub-sections 4.2.2 and 4.2.3, necessary modeling elements are implemented. Using these elements, the model of the case study is designed.

The server node's and client node's UDP protocol is enabled to establish communication. The client uses the pin 15 to sample the light intensity data. To achieve this, a *GPIO* element was added to the client node and it was defined as input. The model elements are binded with the relations named *hasSocket* and *hasGPIO*.

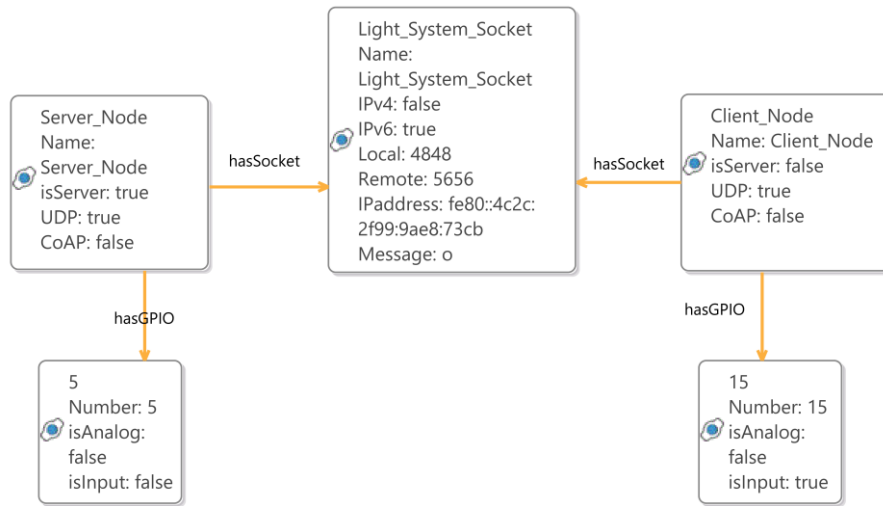


Figure 5.6 Platform-Specific Model of smart lighting system.

The client node has one *GPIO* element. It uses to pin 5 to control the lighting according to the received message. The client node and the server node has a *Socket* element to define outgoing and listening ports, IP address, and the message. Since the server node on or off the lighting according to the received message, the *Socket* element's message attribute was inputted as *o*. However, since the server node opens the lighting when the *o* character is received and closes the lighting when the *f* character is received, it does not enough generate two conditions using a *Socket* element. This functionality can be achieved by adding the delta codes to the target code that is generated by the model. The delta codes and the evaluation of these models are mentioned in section 11.

Both case studies can be modeled using at most 6 elements. The numbers of the elements could be more than 6 elements. The designer had to use thread elements at the server-side and the client-side to generate UDP communication codes to establish a server/client-based network. However, since during the implementation of the DSML4Riot, the special threads are generated when the designer selects the communication type. In this way, the requirement for additional elements is eliminated and the design is simplified.

It should be noted that eliminating some elements and providing fewer elements for the designer is not always a good option. Firstly, this also shifts the complexity during the implementation of the code generation rules. Secondly, the designer may not associate the design elements with the output of the expected target platform's code. As a result, optimal design elements should be presented and these design elements should be satisfactory to the designer's expectations from the generated code.

6. PLATFORM-INDEPENDENT MODEL

In Model-Driven Architecture, it is aimed to model Platform-Independent Level separated from the technical terms and technical operations of the Platform-Specific Level and relate the design elements of PIM to elements of PSM. The idea behind to define a Platform-Independent Model is to have a common structure that can be transformed into one or more Platform-Specific Models. Independence is a relative definition in terms of to define the level of the abstraction. It indicates the separation of one platform from another one. In other words, it represents two platforms level where one platform level is either more or less abstract compared to the other platform level.

The main advantage of this method is to derive various alternative representations of the Platform-Specific level using the same PIM. In this way, the derivation process of PSMs can be partially automated. Therefore, the code generation of the target system can be made using derived PSMs. The Platform-Independent Models should satisfy the requirements of the target system and should have configurable elements. Moreover, these elements should be configured during the transformation process. As a result, the derived PSMs from the PIM must have the best configurations to have nearly a fully functional system as well as requiring less interaction by the designer.

The benefits of the Model-Driven Architecture can be summarized as follows:

- The designer creates a domain model of the target system without thinking about how to implement it as well as requiring technical knowledge of the domain.
- Using a PIM, a lot of PSMs can be generated for the multiple target systems or a target system can be represented by alternative PSMs.
- It reduces the development time and implementation efforts. Moreover, models at all levels can be traced by the designer and stakeholders in the development process.
- The models can be transformed to another model representations (e.g. Petri-net). Therefore the model at any level can be extended in another platform, in this way many analyses results can be achieved.

The following figures represent the PIMs of *Design*, *Direct*, and *Indirect* viewpoints. The *Design* viewpoint includes the design elements of *Direct*, *Indirect*, *DataCollector*, and *Logger*. The meta-model of the Platform-Independent level is mentioned in sub-section 7.3. The EClasses that are

designed in the Platform-Independent Meta-model are instanced in the Platform-Independent Model to provide a Platform-Specific Model for the designer. The designer simply uses the PIM elements and viewpoints to derive PSMs. The viewpoints provide design simplicity for the designer. The designer uses the *Indirect* and *Direct* related elements in the viewpoints. Therefore, connection and design complexity are prevented. The element mappings of PIM to PSM are represented in figure 7.6.

The PIM elements are described by the meaningful icons. These icons are selected carefully both to guide the designer to simplify the design process and to provide a user-friendly environment. In following table 6.1 the icons and their definition are given.

Platform-Independent Model Icons and Their Definitions	
Icon	Definition
	DataCollector represents the RaspberryPI at the PSM level. When this element is included in the design it contains the Riot and Contiki viewpoints in the PSM level. It is described with a bridge-like icon to describe information is gathered at one point and transmitted to the other point.
	Logger represents the topology viewpoint at the PSM level. When this element is included in the design it also represents the log manager in the system.
	The Direct viewpoint is represented by a blue-white colored D icon. The designer can create many Direct viewpoints using this element. It has D letter to specify it is a Direct element.
	The Indirect viewpoint is represented by a black-white colored i icon. The designer can create many Indirect viewpoints using this element. It has i letter to specify it is an Indirect element.
	This icon represents the connector of server and client nodes both in Direct and Indirect viewpoints. It transforms to the server connection and client connection at the PSM level. This icon is selected because it looks like a plug or an outlet as if two elements must be connected to operate.
	This icon represents the DirectDevice element. At the Direct viewpoint side more colorful icons are selected. This icon has more frequent waves and colorful design. Since it represents an IoT node, this icon is suitable to use.

	In the Direct viewpoint, the designer can add multiple DirectDevice elements.
	This icon represents the IndirectDevice element. At the Indirect viewpoint side less colorful and simple icons are selected. This icon has less frequent waves. Since it represents a WSN node, this icon is suitable to use. In the Indirect viewpoint the designer can add multiple IndirectDevice elements.
	This icon represents a sensor element in the PSM level in an Indirect viewpoint. The designer adds this element to the design and selects which data is going to be sampled. It is represented by a signal and wave icon. It is selected because it imitates collecting and sensing sensor data.
	This icon represents a process thread element in the PSM level in the Indirect viewpoint and EventBasedTask at the PIM level. The designer adds this element to the design and selects the threads that are going to be created. It is represented by a black square and with line dots. This icon is selected because it imitates a box that can include anything and the dotted line looks like a discrete graph since it is an event-based task.
	This icon represents a timer element in the PSM level in the Indirect viewpoint and EventBasedTimer at the PIM level. The designer adds this element to the design and relates an EventBasedTask with an EventBasedTimer that are going to be created. It is represented by an hourglass. This icon is selected because it imitates the sand particles flow to the downside of the hourglass and the flow event of the sand particles describes the time.
	This icon represents the port element in PSM and PIM levels. It represents a ship that leaves from the port. It includes dotted lines to be compatible with other Indirect viewpoint's icons.
	This icon represents the SAUL element in the PSM level and SensorDevice at the PIM level. It is showed with an indicator symbol.
	This icon represents a thread element in the PSM level in the Direct viewpoint and RealTimeTask at the PIM level. The designer adds this element to the design and selects the threads that are going to be created. It is represented by a black square and a solid line. This icon is selected because it imitates a box that can include anything and the

	solid line looks like a linear, real-time graph since it is a real-time task.
	This icon represents an xtimer element in the PSM level in the Direct viewpoint and RealTimeTimer at the PIM level. The designer adds this element to the design and relates a RealTimeTask with a RealTimeTimer that are going to be created. It is represented by a modern watch. This icon is selected because it imitates the modern times watch where the time is really important.
	This icon represents the Socket element in the PSM level at the Direct viewpoint and it represents the RealTimeConnection at the PIM level. This icon is selected because it contains a brain with connection points. Since the brain is a real-time connector with the outer world, this icon is suitable to use.

Table 6.1 Platform-Independent Model icons and their definitions.

Figure 6.1 represents the Platform-Independent Model of the case studies in the sub-sections 4.2, 4.3, 4.5.4, and 4.5.5. They also represent the more abstract model of the Platform-Specific Models that are given in the sub-section 5.5.

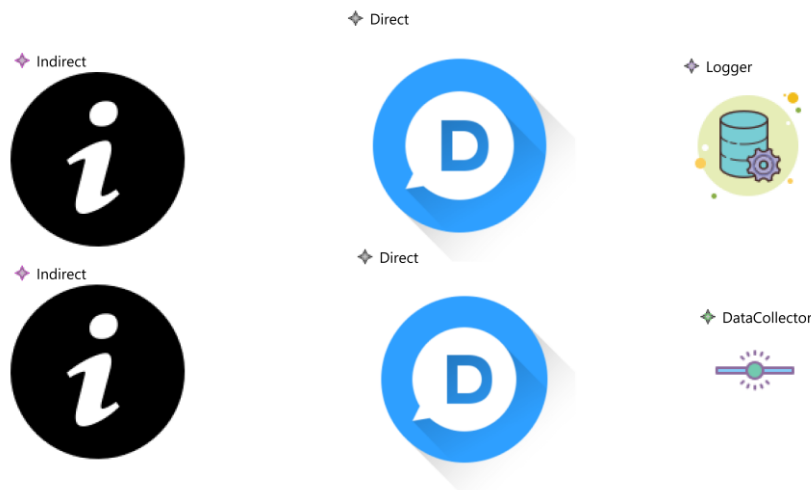


Figure 6.1 Design viewpoint that contains Direct and Indirect viewpoints.

To model the four case studies two *Direct* and two *Indirect* viewpoints are added to the *Design* viewpoint. Moreover, *Logger* and *DataCollector* elements are also added. The *DataCollector* transforms to a gateway for the IoT and WSN nodes. The *Logger* element transforms to a log manager and the topology viewpoint. Since there are two case studies implemented in the WSN side two *Indirect* elements are used. The same is made for the *Direct* side. When double-click to these icons, a

viewpoint is created and the sub-elements related to the viewpoint can be selected to create a model. The viewpoints and PIMs of the case studies are mentioned in the following sub-sections 6.1, 6.2, 6.3, and 6.4.

6.1. Indirect Viewpoint: Smart Fire Detection System

In this sub-section, the Platform-Independent Model of the smart fire detection system is mentioned. The elements that are explained in section 6 are used to create the PIM of the smart fire detection system.

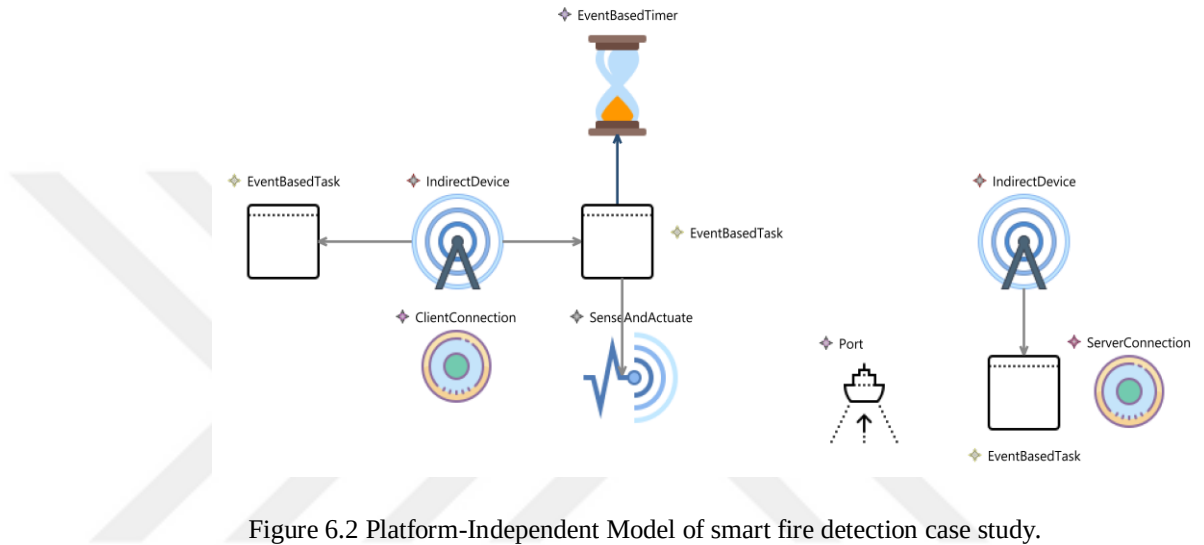


Figure 6.2 Platform-Independent Model of smart fire detection case study.

To create the model, three *EventBasedTask* elements, two *IndirectDevice* elements, one *EventBasedTimer* element, one *Port* element, one *SenseAndActuate* element as well as *ServerConnection* and *ClientConnection* elements are used. The elements are labeled with their names. On the left side of figure 6.2, the elements that are related to the client-side are shown. On the right side, the elements that are related to the server-side are given.

According to the Platform-Specific Model of the smart fire detection case study that is shown in figure 5.3, two nodes and three processes must be added in the PIM. The *Port* and *Server/Client Connection* elements must be also added. Some elements are not needed to be connected to the PIM. Because they are automatically connected during the PIM-to-PSM transformation. However, *IndirectDevice*, *EventBasedTask*, and *EventBasedTimer* elements must be specifically connected. Otherwise, all *EventBasedTask* elements are connected to all *IndirectDevice* elements. This can be a problem when the PIM is transformed to the PSM since they are many unnecessary connected relations. The PIM has a more light-weight and colorful design to guide the designer. In this way, the designer can model a system independently from the technical terms of the target platforms.

6.2. Indirect Viewpoint: Smart Lighting System

Figure 6.3 shows the Platform-Independent Model of the smart lighting case study. To create the model, four *EventBasedTask* elements, three *IndirectDevice* elements, one *EventBasedTimer* element, one *Port* element, and one *SenseAndActuate* element, as well as *Server/Client Connection* elements, are used. Three *IndirectDevice* elements are used because an actuator node had designed at the PSM level. The other two *IndirectDevice* elements are added to be transformed into *Node* elements. The *IndirectDevice* element has *EventBasedTask* elements to create processes at the PSM level. The designer can input the name for the *EventBasedTask* element. The period of the *EventBasedTimer* element can also be input by the designer. These are important attributes that have to be set by the designer. The designer can also set the sensor type using the *SenseAndActuate* element.

The attributes of the elements can be set by double-clicking on the element's icon. The properties windows are opened and the attributes are shown. These attributes are shown according to meta-models that are mentioned in section 7. Since a model element is the instance of a meta-model's EClass, then the attributes have to conform to its meta-model.

The evaluation of the Platform-Independent Model of the smart lighting system is made in Section 11.1.2. The evaluation is made according to transformation capability and manual completion of the model.

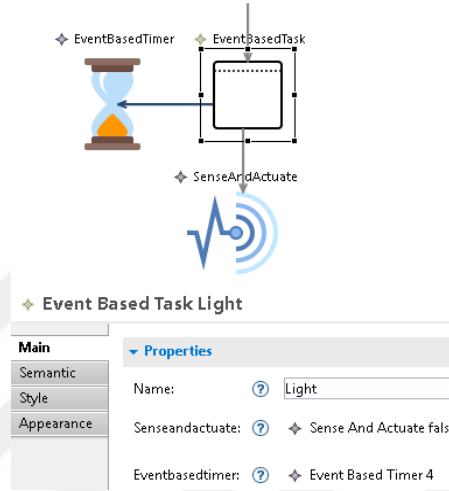


Figure 6.4 The Name attribute of the EventBasedTask is set as Light.

Figure 6.4 shows that *Name* attribute of the *EventBasedTask* is set as Light. Moreover, it also has a *SenseAndActuate* and an *EventBasedTimer* elements. These attributes are transformed to the attributes of the Platform-Specific Model elements. However, the attributes and elements that require technical and domain knowledge must be set and added in the Platform-Specific Level.

6.3. Direct Viewpoint: Smart Irrigation System

In this sub-section, the Platform-Independent Model of the smart irrigation system is explained. The elements that are mentioned in section 6 are used to create the PIM of the smart irrigation system in *Direct* viewpoint. Compared to the Platform-Independent Models of the *Indirect* viewpoint, the Platform-Independent Model of the smart irrigation system has fewer elements. Because there are few elements in the Platform-Specific Model of the smart irrigation system. As mentioned in section 7.2, this simplicity is provided due to the operating system and the design approach of the meta-model. Figure 6.5 describes the Platform-Independent Model of the smart irrigation system. It has two *DirectDevice* elements and one *RealTimeConnection* element.

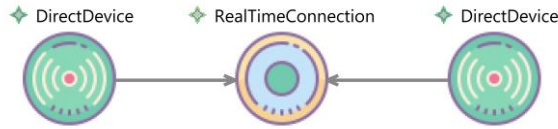


Figure 6.5 Platform-Independent Model of the Smart Irrigation System.

The *DirectDevice* elements and *RealTimeConnection* element transforms to *NodeR* and *Socket* elements in the Platform-Specific Level. Since there is not any requirement for *Thread* element in the Platform-Specific Level, *RealTimeTask* element is not added to the Platform-Independent Level. When the *RealTimeConnection* element is added to the model it automatically activates the UDP feature when it is transformed to the *Socket* element. As mentioned in the section 5.2, if the UDP feature is activated, a special thread is generated for the UDP communication, when the code generation is made based on Platform-Specific Model. Therefore, the *RealTimeTask* element is not used in PIM of the Smart Irrigation System. However, the designer can add *RealTimeTask* elements to create threads. The *RealTimeTask* can have a *RealTimeTimer* (see Figure 6.6). The *RealTimeTimer* element is used to define a delay in a thread. The *DirectDevice* element can have many *RealTimeTask* elements. The *GPIO* element has not an element in the Platform-Independent Level. Because, since it is a technical term, attribute setting and quantity of the GPIO element must be made in the Platform-Specific Level.

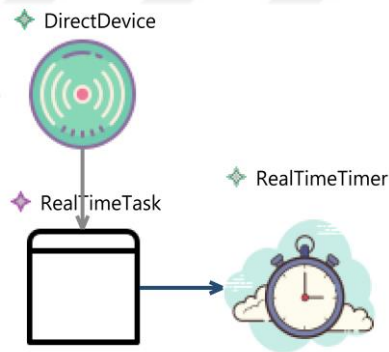


Figure 6.6 Usage of the RealTimeTask and RealTimeTimer elements.

6.4. Direct Viewpoint: Smart Lighting System

In this sub-section, the Platform-Independent Model of the smart lighting system is explained. The elements that are mentioned in section 6 are used to create the PIM of the smart lighting system in *Direct* viewpoint. Compared to the Platform-Independent Models of the *Indirect* viewpoint, the Platform-Independent Model of the smart lighting system has fewer elements. Because there are few elements in the Platform-Specific Model of the smart irrigation system. As mentioned in section 7.2, this simplicity is provided due to the operating system

and the design approach of the meta-model. Figure 6.7 shows the Platform-Independent Model of the smart lighting system. It has two *DirectDevice* elements and one *RealTimeConnection* element. As can be seen, the Platform-Independent Model of the smart fire detection case study has the same model with the smart lighting system. Since the same similarity exists in the Platform-Specific Level, it is a normal have it in a more abstract and generalized level that is Platform-Independent. When these two models are transformed to the Platform-Specific Model, it gains more specific elements and attributes to achieve the model of the target platform.

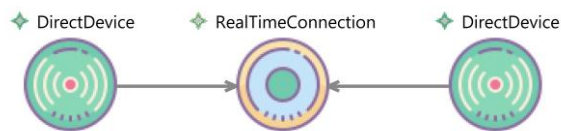


Figure 6.7 Platform-Independent Model of the Smart Irrigation System.

The designer can set the name attribute of the *RealTimeTask* element and seconds attribute of the *RealTimeTimer* element. Figure 6.8 shows that a *RealTimeTask* element's name attribute is set as *AnyTask* and *RealTimeTimer* elements' seconds attribute is set as 2.

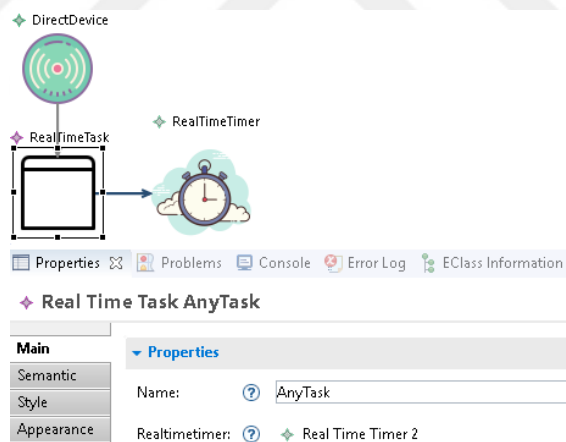


Figure 6.8 The designer can set attributes of the RealTimeTask and RealTimeTimer elements.

In conclusion, the Platform-Independent Models of the case studies are mentioned. The elements of the *Direct* and *Indirect* viewpoints are explained. The setting of the attributes are shown. The relations between PIM elements and PSM elements are given. The elements' icon and their meanings are introduced. The evaluation of the Platform-Specific Model level (DSML4Contiki and DSML4Riot) and Platform-Independent Model level are made in the Section 11.

7. META-MODELS & TRANSFORMATION RULES

A meta-model defines the classes, attributes, relations, and the cardinality constraints of a model. A model is designed according to its meta-model. Many models can be instantiated according to a meta-model. In short, a meta-model defines the scopes and the constraints of a model. In the following sub-sections Contiki, Riot, and PIM meta-models are presented. Meta-elements are explained and comparison between Contiki and Riot meta-models is made. PIM meta-model and model-to-model transformation rules are given and explained.

7.1. Meta-model of Contiki Operating System

Figure 7.1 represents the meta-model of the Contiki operating system. It includes the features of the Contiki operating system. The features are represented as EClasses and parameters are represented as attributes of these EClasses. The modeling approach that is mentioned in section 4.2.1 and 4.2.2 is applied in this meta-model. Moreover, the models of the case studies that are given in section 5.5 are instantiated according to this meta-model.

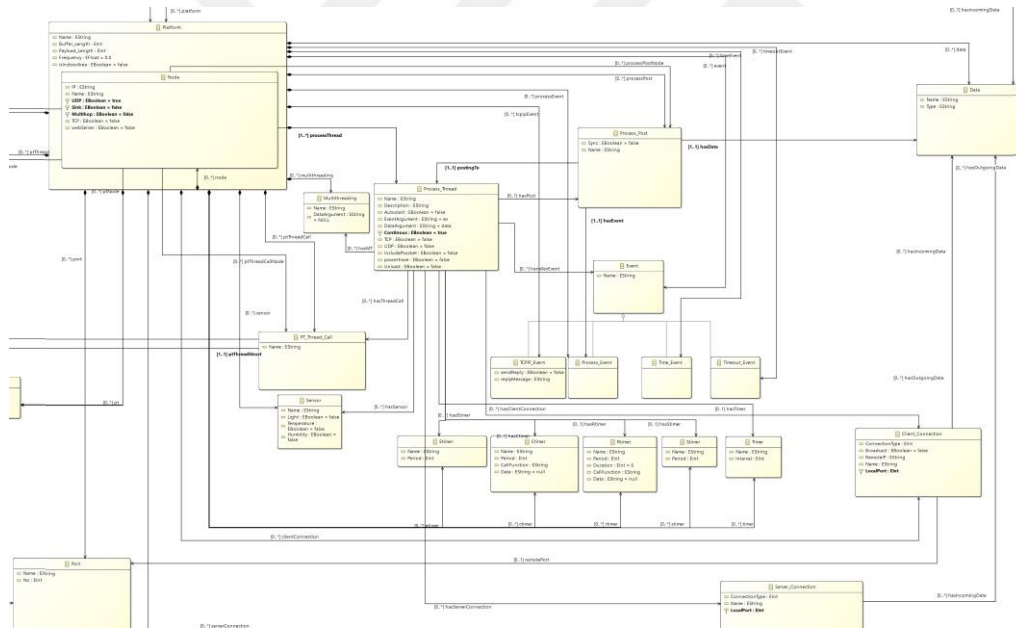


Figure 7.1 ContikiOS' meta-model.

The Contiki operating system meta-model has 17 EClass and 26 relations. It highly includes various timers and events EClasses. The timers of the Contiki operating system and their attributes are represented at the right bottom of Figure 7.1. Above them, events are represented. These EClasses are designed according to features of the Contiki operating system. Since the Contiki operating system has protothreads and protothreads have timers and events, it has reference relations to these EClasses. As can be seen from Figure 7.1, *Process_Thread* references *ETimer*, *CTimer*, *RTimer*, *STimer*, and *Timer* EClasses. Moreover, *TCPIP_Event*,

Process_Event, *Time_Event*, and *Timeout_Event* are referenced by *Process_Thread* EClass. The following Table 7.1 explains the roles of these EClasses.

Contiki Operating System's Meta-model Elements and Their Definitions	
EClass	Definition
Platform	It is the root of the whole meta-classes. All meta-classes are derived from and contained by Platform meta-class.
Node	It represents a node in the system level. Using this meta-class' instance a network also set as Single-hop or Multi-hop, TCP/UDP in the modeling environment. A node can be also set as Sink or Source node.
Event	This meta-class is the parent of 4 meta-classes. It represents the events in Contiki operating system.
TCPIP_Event	It represents the tcpip_event in Contiki operating system.
Timeout_Event	It represents the timeout_event in Contiki operating system.
Timer	This meta-class resets, sets and creates timers. It is the parent meta-class of 4 meta-classes.
Etimer	In defined interval, etimer counts down and when it expires generates a timeout event.
Stimer	In defined time interval, it counts down and uses seconds.
Ctimer	It is a callback timer. When is expired, it calls a function and can send a data parameter.
RTimer	It is a real-time timer. It is uses for real time tasks. It calls a function and sends a data parameter.
Process_Thread	Process_Thread is used to define a process. Autostart attribute describes the process starts when the hardware is powered on. Continous feature describes it is a continous process.
Process_Event	Processes have event parameter in Contiki it uses this argument to check wheter an event occured or not. Moreover this event can be send as paramter to another process.

PT_Thread	For source constrained devices it is specially designed stackless and lightweight thread. It is used with TCP/IP protocol.
Client_Connection	A node can start a connection and sends first message via Client_Connection with an outgoing data, RemoteIP, and RemotePortNumber attributes. This connection can also be used to retrieve incoming messages from the sink node.
Server_Connection	The Sink Node's connection to system is satisfied by Server_Connection. LocalPort feature achieves to Sink Node's to communicate each other. Client_Connection and Server_Connection establish a connection on Port Class.

Table 7.1 Contiki operating system's meta-model elements and their definitions.

The above table 7.1 describes the meta-classes and their roles in the Contiki operating system. In sub-section 7.2, meta-classes of the Riot operating system and their definitions are given.

7.2. Meta-model of Riot Operating System

The meta-model of the Riot operating system is introduced in this sub-section. It represents the abstract syntax of the DSML4Riot. Generally, a meta-model represents system elements, element relations, and cardinality constraints of elements and their relations. The modeling approach that is mentioned in section 4.5.2 and 4.5.3 is applied in this meta-model. Moreover, the models of the case studies that are given in section 5.5 are instantiated according to this meta-model. When it is compared, the meta-model elements and relations of DSML4Contiki are more than DSML4RiotOS. This is occurred because of Contiki operating system has a lot of event-based features and specialized timers. Moreover, in the design of the Contiki operating system's meta-model, some features are modeled as an EClass to give diversity to the designer. In Riot operating system's meta-model, its features are modeled as attributes to provide more abstraction to the designer. The same approach could not be used while developing the meta-model of the Contiki operating system, because it was not so possible to decide which and how many timers (or events) are going to be used by the designer. In Riot operating system, it is more possible to decide where a timer code should be generated. Since it is a real-time operating system, it does not have additional features for event handling. Accordingly, most of the features could be modeled as an attribute of an EClass. In Figure 7.2 elements of Riot's meta-model are shown and in the following table, they are explained.

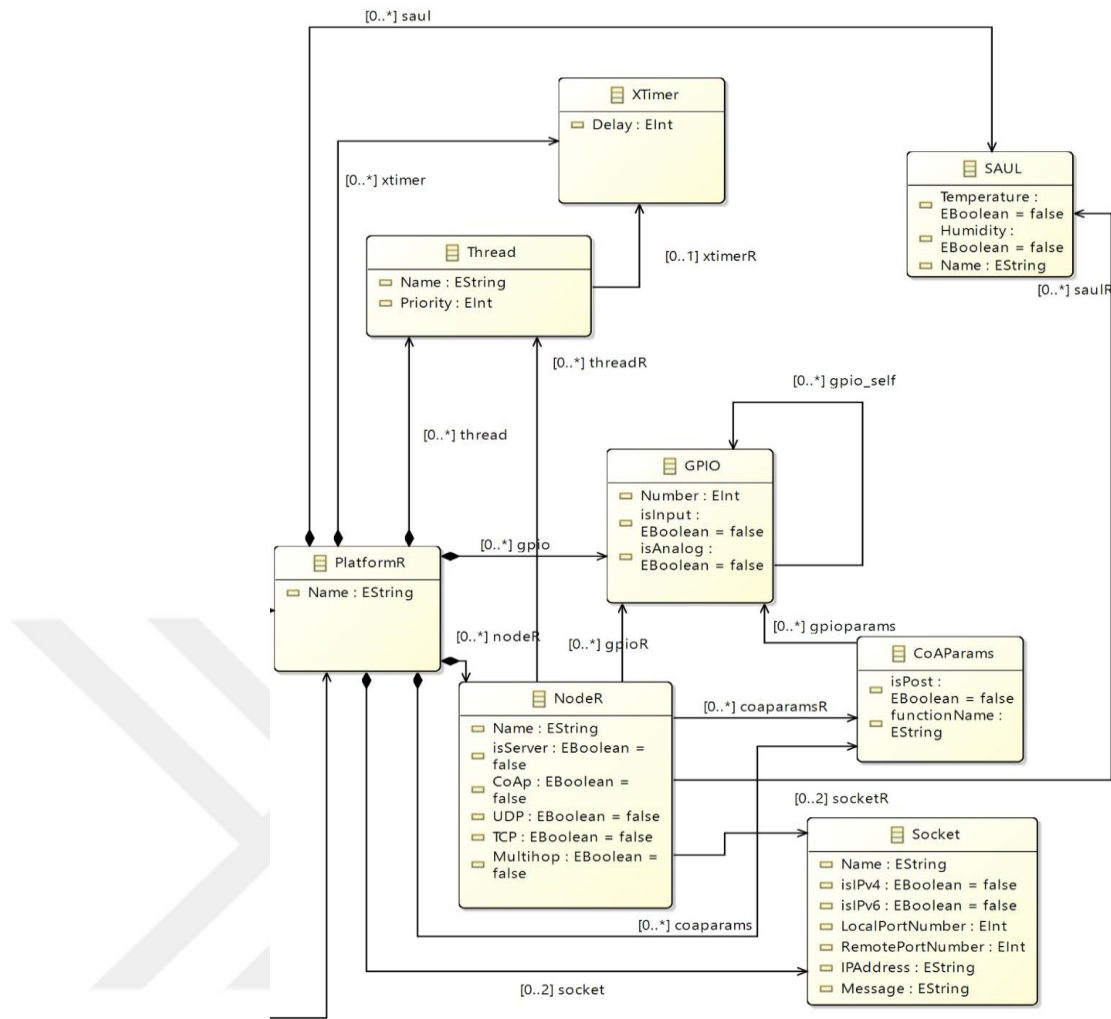


Figure 7.2 Riot operating system's meta-model.

The Riot operating system meta-model has 8 EClass and 25 attributes. The relations between EClasses and cardinality constraints can be seen. At most two *Socket* elements can be added to the model. Because a *Socket* element can be based on an *IPv4* or *IPv6*.

A *Thread* element can have at most one *XTimer* element. Because a thread should not have a lot of delays. If more than one timer is required, then the designer can add delta codes by simply copying timer's code to the necessary code lines.

A lot of *SAUL* elements can be used in the design. Because a node or a lot of nodes can have various sensors. Normally, a sensor device can have a limited amount of sensors. However, this varies according to the device specification. A sensor device can have many pins and pin extension modules to have a lot of sensors. Since it cannot be constraint to any number. It is described with zero or many cardinalities.

Some of the relations are described with a composition and a reference relation. The *PlatformR* EClass is the root element of the other elements. Every element that represents the Riot operating system's features are derived from the *PlatformR* element by a composition relation. A reference relation is simply used to refer to instances of the elements that are derived from the root EClass.

Riot Operating System's Meta-model Elements and Their Definitions	
Element	Definition
Thread	It is used to define additional threads to main thread. Name attribute is used for function name of that thread. Priority must not be set, if it is desired to run this thread equal run-time to main thread.
Xtimer	It is a delay timer and it can be used in threads. However, it is not recommended to use more than 1 seconds since Riot is real-time operating system. This feature is added to if a delay has to be used at initialization of any task and should not be used in while blocks.
PlatformR	The Platform which is root of the Riot viewpoint.
NodeR	It represents the any node (e.g ESP32) and it can be either server or client. If it is a server then can also run CoAP server.
CoapParams	Coap params is used for CoAP server. It defines if a request is POST or GET and name of the function to define URI. (e.g server.com/get_board_info)
Socket	Socket is used for UDP and TCP protocols. It defines usage of IPv4 or IPv6 and also Local and Remote ports to establish connection. Also IP adress parameter is necessary. Message attribute represents the message to be sent.
GPIO	It used to define active gpio of the board (e.g esp32) a number is the gpio number to be activated. Then input or output role is defined and also it can be analog, if not set it is digital as default. Moreover, CoapParams can have gpio and it can be toggled.
SAUL	Saul is sensor/actuator abstraction layer for RIOT. By this way, any sensor can be activated and any actuator can be interacted without adding device parameter to application code.

Table 7.2 Riot operating system's meta-model elements and their definitions.

Table 7.2 describes the meta-classes and their roles in the Riot operating system. In sub-section 7.3, the meta-model of Platform-Independent level is given. In the sub-section, their definitions are made. In sub-section 7.4, transformation rules to transform the model elements from the Platform-Independent level to the Platform-Specific level are given. To clarify the transformation mapping, the transformation mapping table is given in Figure 7.6.

7.3. Platform-Independent Meta-model of Contiki & Riot Operating Systems

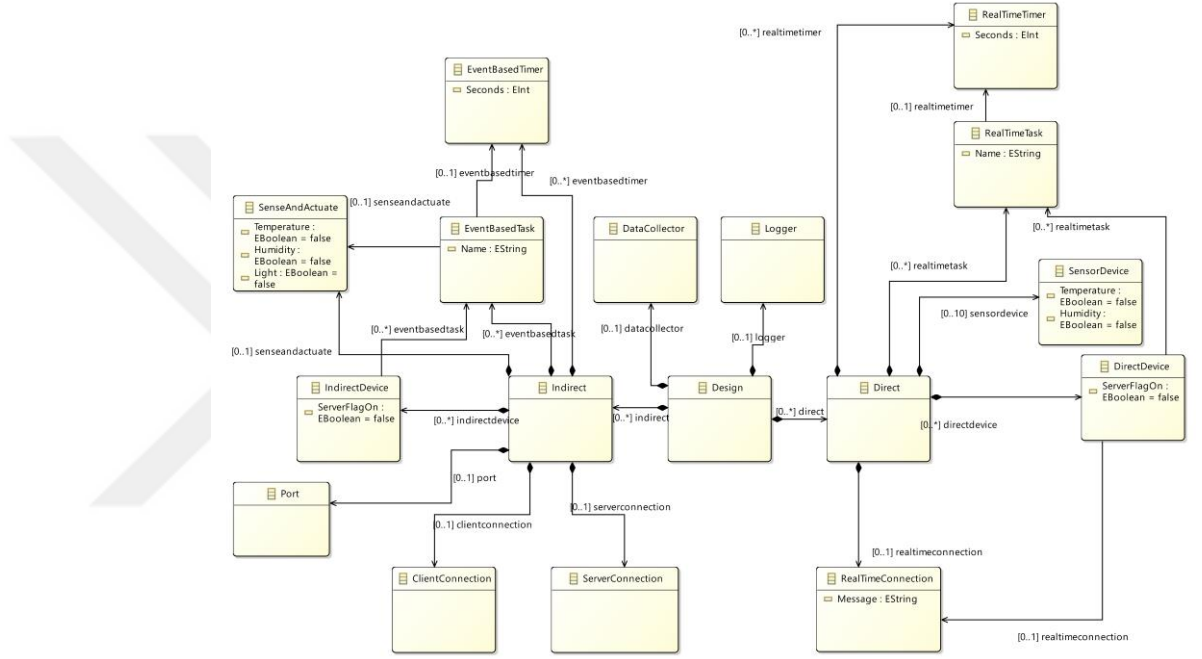


Figure 7.3 Platform-Independent Meta-model.

The PIM meta-model is divided into two ways as *Direct* and *Indirect* EClasses (see Figure 7.3). It has the root EClass of Design. It has 17 EClasses, 12 attributes, and 22 relations. *Direct* represents a direct platform that can communicate directly with a system via Internet (IoT Platform). The indirect represents the WSN Platform. Since a WSN network platform requires a gateway to send information to the Internet, they are named as Indirect. The *IndirectDevice* and *DirectDevice* are mapped to Nodes at the PSM level. In the PSM level, there have to be two nodes that are Client and Server. *ServerFlagOn* attribute represents the node that is going to be a Server Node. The PIM metamodel and transformation rules are designed to provide a server/client based architecture to the designer. The designer should add the necessary elements to the design. The designer can also define a lot of *Direct* and *Indirect* viewpoints to represent many systems. The transformation rules set the attributes automatically to establish fast development.

For example, port numbers are automatically filled. In this way, an abstraction is provided not to bother the designer with these basic configurations. Most of the PSMM (Platform-Specific Meta-Model) level design can be achieved by just adding the elements in the PIM (Platform-Independent Model) level. In *Indirect* (WSN side) the designer chooses the sensors to be activated such as Temperature, Humidity, and Light. In *Direct* (IoT) side has nearly the same elements with the *Indirect* side. During the PIMM's design, providing similarity between *Direct* and *Indirect* sides is aimed. The *Indirect* and *Direct* EClasses are designed as viewpoints in Sirius. In this way, the designer can work in different viewpoints to design any model for a system. The EClasses that require to use the instances of the *Indirect* and *Direct* EClasses are connected with reference relation. During the Platform-Independent Model to Platform-Specific Model transformation process, these relations are also transformed to PSM level's relations. In below, these transformation rules and transformation process are mentioned.

7.4. Transformation Rules for PIM-to-PSM

One of the most important operations in Model-Driven Engineering is a model transformation (Bézivin et al., 2004). A transformation is an automated method that is used for modifying an existing model or creating new models. A transformation is a defined operation. An operation may also have input and output parameters. Operation is made according to a rule. Therefore, to achieve model-to-model transformation a set of rules must be defined. Transformation rules for transforming PIM elements to PSM elements are written using Atlas Transformation Language (Jouault et al., 2008). ATL is a model transformation language that enables us to specify how one or more target model(s) can be produced from a set of source models. The model-to-model transformation is made using the above meta-models. In below, transformation rules and transformation tables are given. Figure 7.4 describes a basic model transformation process.

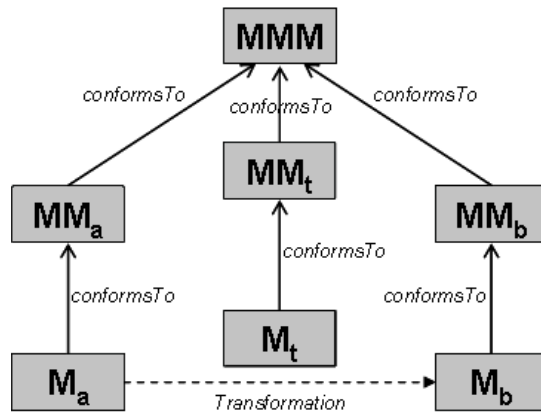


Figure 7.4 Model transformation process.

The model transformation aims to provide target models from one or more of the source model(s). To achieve this, a tool must enable developers to define source models that are going to transform into a target model. Refer to figure 7.4, a simple model transformation has to generate model *Mb* that conforms to meta-model *MMb* and has to use model *Ma* that conforms to meta-model *MMa*. Moreover, to transform model *Ma* to model *Mb*, a set of transformation rules *Mt* must be written for element mapping. These rules conform to the meta-model that defines the model transformation semantics. All of these meta-models conform to the meta-meta-model.

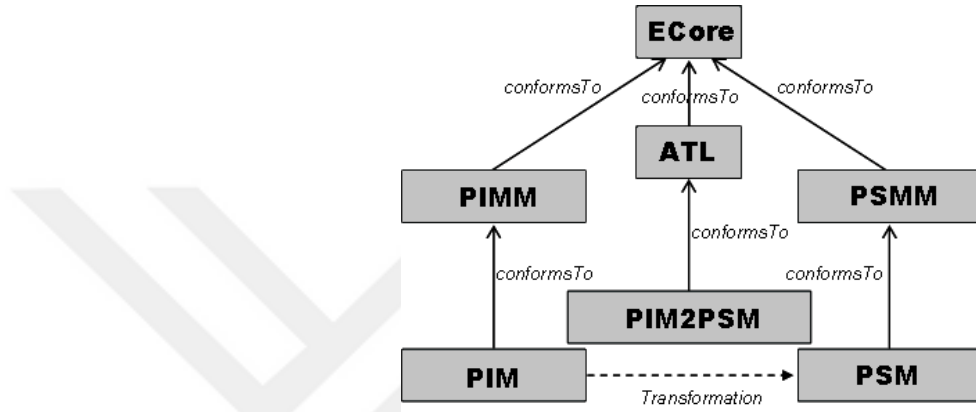


Figure 7.5 PIM to PSM transformation process.

Above figure 7.5 summarizes the model-to-model transformation process of PIM to PSM. PSM target model that conforms to PSMM is generated from the PIM input model that conforms to the PIMM. Transformation rules that conform to ATL metamodel transformation are shown below. The PIMM and PSMM are expressed using the semantics of the Ecore (Salay et al., 2007) meta-meta-model. The Platform-Independent Meta-model and Platform-Specific Meta-models (Contiki and Riot) are included to ATL file using the following definitions in Listing 7.1.

```
-- @path ContikiOS=/ContikiOS/model/PSM.ecore
-- @path PIM=/PIM/model/PIM.ecore
module myTransformation;
create OUT : PSM from IN : PIM ;
```

Listing 7.1 Meta-model inclusion to use the meta-elements.

Three helper rules are written. First and second helper rules are written to define node elements as either a server node or a client node. The third helper rule is written to numbering multiple elements. These three helper rules are shown below Listing 7.2.

```
helper context PIM!DirectDevice def: selectServer() : String =
    if (self.ServerFlagOn) then 'Server'
    else
        'Client'
```

```

endif;
helper context PIM!IndirectDevice def: selectServer() : String =
  if (self.ServerFlagOn) then 'Server'
  else
    'Client'
  endif;
helper def: var: Integer = 0;

```

Listing 7.2 The helper rules that is used for enumerating elements' labels.

The syntax to write a transformation rule in ATL is simple. A rule is defined starting by a *rule* keyword and then it requires a unique name. In following Listing 7.3 a transformation rule with a unique name is given as *DesingToIoTSystem*. *UpperCamelCase* convention is used for naming the transformation rules. The transformation's way is defined using *from* and *to* keywords. Since the transformation is made to PSM from PIM, the transformation rule must be written as *from pim: PIM!Design to iot: PSM!IoTSystem*. This defines the transformation is made from PIM's Design element to PSM's IoTSystem element. In other words, the Design element is going to be mapped with the IoTSystem element. Since elements have to be mapped, attributes of these elements have to be mapped too. In the following rule, the Name attribute of PSM's IoTSystem element is set to *DesignIoTSystem*. In ATL, relations are treated as an attribute. Therefore, the relations of PIM elements are mapped to PSM elements. The mapping operation is made using "<-" symbol.

```

rule DesignToIoTSystem {
  from
    pim: PIM!Design
  to
    iot: PSM!IoTSystem (
      Name <- 'DesignIoTSystem',
      platformR <- pim.direct,
      platform <- pim.indirect,
      logman <- pim.logger,
      raspberrypi <- pim.datacollector  })
}

```

Listing 7.3 A transformation rule to transform a PIM's Design element to PSM's IoTSystem element.

In the design of the meta-models, all elements are connected to the root element. PIM's level root element is *Design* and PSM's level root element is *IoTSystem*. Both PIM and PSM level elements are derived from their root elements and this derivation is represented by composition relation. If an element needs to use another element, then it references the required element. If an element needs to use another element, then the required element is referenced from the root element. In the following transformations rules, there is a special definition. An Ecore function *eContainer* is used to access the root element. Because the *Logger* element is derived from the *Design* element in the PIM level. In other words, the *Logger* element is composed of the *Design* element. Moreover, in the PSM level *LogMan* element has a reference relation named *raspberrypi*. In conclusion, the *Design* element's composition relation named *datacollector* is mapped to the *LogMan* element's reference relation named *raspberrypi*. The below figure 7.6 summarizes this mapping.

ViewPoint	PIM	PSM
System	Design	IoTSystem
System	Logger	LogManager
System	DataCollector	RaspberryPI
Riot	Direct	PlatformR
Riot	DirectDevice	NodeR
Riot	RealTimeConnectio	Socket
Riot	RealTimeTask	Thread
Riot	RealTimeTimer	Xtimer
Riot	SensorDevice	SAUL
Contiki	Indirect	Platform
Contiki	IndirectDevice	Node
Contiki	EventBasedTask	Process Thread
Contiki	SenseAndActuate	Sensor
Contiki	EventBasedTimer	Etimer
Contiki	ClientConnection	ClientConnection
Contiki	ServerConnection	ServerConnection
Contiki	Port	Port

Figure 7.6 PIM to PSM mapping table.

The above transformation rules represent the element to element transformations that are given in Figure 7.6. These rules simply either defines the parameters of the target element's attributes or transforms a PIM element's attribute to the PSM element's attribute. When the transformation is finished PSMs are generated. During the implementation of the transformation rules, it is aimed to define the exact parameters to create a system for the designer. It is also intended to provide less delta coding for the designer. For this purpose, some attributes are automatically defined to be an example of usage and self-documenting of the modeling environment for the designer. In one of the model transformations rule named *RealTimeConnectionToSocket* defines port numbers, IP address type, IP address number, and name of the socket. Setting the attributes of the elements automatically provides a guide for the designer in the Platform-Specific Modeling level. Moreover, it helps to achieve fast prototyping by not bothering the designer for technical configuration and technical details. However, the designer should have some technical information on the Platform-Specific Modeling level. In case of a lack of technical information, the designer can still create a system using the Platform-Specific Model since the parameters are generated automatically to create a functional system. In below Listing 7.4 the transformation rules of PIM-to-PSM are given.

When the designer transforms a Platform-Independent Model to a Platform-Specific Model, it may decide to add new elements or the designer may decide to start directly at Platform-Specific Level. In both cases, the designer has to validate the designed model to prevent upcoming problems at the system level. Therefore, a set of semantic constraints must be written to validate the designed model. In the following section 8, these semantic constraints are mentioned as well as including the syntax of the languages.

```
rule LoggerToLogMan {
  from
    log: PIM!Logger
  to
    lm: PSM!LogMan (
      URL <- 'www.example.com',
      raspberrypi <- log.eContainer().datacollector
    )
}

rule DataCollectorToRaspberryPi {
  from
    pim: PIM!DataCollector
  to
    rpi: PSM!RaspberryPi (
      Name <- 'RaspberryPI' +
      thisModule.var.toString(), portName <- '/ttyUSB0',
      RiotOS <- pim.eContainer().direct,
      PSM <- pim.eContainer().indirect
    )
}
```

```
)
do{ thisModule.var<-thisModule.var + 1;
}

rule DirectToPlatformR {
  from
    pim: PIM!Direct
  to
    psm: PSM!PlatformR (
      Name <- 'RiotPlatform',
      nodeR <- pim.directdevice,
      saul <- pim.sensorddevice,
      thread <- pim.realtimetask,
      xtimer <- pim.realtimetimer,
      socket <- pim.realtimeconnection
    )
}

rule DirectDeviceToNodeR {
```

<pre> from pim: PIM!DirectDevice to noder_server: PSM!NodeR (Name <- pim.bak(), UDP <- true, socketR <- pim.eContainer().realtimeconnection, isServer <- pim.ServerFlagOn, threadR <- pim.eContainer().realtimeTask, saulR <- pim.eContainer().sensordevice)) rule RealTimeConnectionToSocket { from pim: PIM!RealTimeConnection to Socket: PSM!Socket (Name <- 'RiotOsSocket', isIPv4 <- true, isIPv6 <- false, LocalPortNumber <- 1234, RemotePortNumber <- 12345, IPAddress <- '255.255.255.0', Message <- pim.Message)) rule RealTimeTaskToThread { from pim: PIM!RealTimeTask to Socket: PSM!Thread (Name <- pim.Name + thisModule.var.toString(), Priority <- 0, xtimerR <- pim.realtimeTimer)do{ thisModule.var<-thisModule.var + 1; }} rule RealTimeTimerToXTimer { from pim: PIM!RealTimeTimer to Socket: PSM!XTimer (Delay <- pim.Seconds)) rule SensorDeviceToSaul { from pim: PIM!SensorDevice to psm: PSM!SAUL (Name <- 'aSaulDevice', Temperature <- pim.Temperature, Humidity <- pim.Humidity)) rule IndirectToPlatform { from pim: PIM!Indirect to Platform: PSM!Platform (Name <- 'ContikiPlatform', Frequency <- 2.4, node <- pim.indirectdevice, processThread <- pim.eventbasedtask, sensor <- pim.senseandactuate, clientConnection <- pim.clientconnection, serverConnection <- pim.serverconnection, etimer <- pim.eventbasedtimer, port <- pim.port)) rule IndirectDeviceToNode { from pim: PIM!IndirectDevice to Platform: PSM!Node (Name <- pim.bak(), UDP <- true, </pre>	<pre> Sink <- pim.ServerFlagOn, Multihop <- true, processThread <- pim.eContainer().eventbasedtask)) rule EventBaseTaskToProcessThread { from pim: PIM!EventBasedTask to processthread: PSM!Process_Thread (Name <- 'Process_Thread', Autostart <- true, Continuous <- true, UDP <- true, EventArgument <- 'ev', DataArgument <- 'data', Description <- 'DescriptionIsHere', hasSensor <- pim.eContainer().senseandactuate, hasEtimer <- pim.eventbasedtimer, hasClientConnection <- pim.eContainer().clientconnection, hasServerConnection <- pim.eContainer().serverconnection)) rule SenseAndActuateToSensor { from pim: PIM!SenseAndActuate to Platform: PSM!Sensor (Name<- 'ContikiSensor', Temperature <- true, Light <- true, Humidity <- true)) rule EventBasedTimerToEtimer { from pim: PIM!EventBasedTimer to Platform: PSM!Etimer (Name <- 'Timer' + thisModule.var.toString(), Period <- pim.Seconds) do{ thisModule.var<-thisModule.var + 1;} } rule ConnectionToClientConnection { from pim: PIM!ClientConnection to cc: PSM!Client_Connection (Name <- 'ClientConn', ConnectionType <- 1, LocalPort <- 3000, remotePort <- pim.eContainer().port)) rule ConnectionToServerConnection { from pim: PIM!ServerConnection to sc: PSM!Server_Connection (Name <- 'ServerConn', ConnectionType <- 1, LocalPort <- 3000, listeningPort <- pim.eContainer().port)) rule PortToPort { from pim: PIM!Port to Platform: PSM!Port (Name <- 'Port', No <- 3001)) </pre>
---	--

Listing 7.4 Transformation rules to transform PIM elements to PSM elements.

8. SEMANTIC CONSTRAINTS

To apply the domain rules in the modeling environment and help the user to design more accurate models, some semantic constraints are implemented in the DSMLs using the Acceleo Query Language (AQL) which is built-in in the Eclipse Sirius Framework. These rules apply the domain rules in the modeling environment. The constraint checking languages are more expressive than the simple cardinality property checking rules implemented in the meta-model and they can check those properties as well. However, the constraint checking rules are generally used to implement the rules that are not possible to implement in the meta-model, such as the rules checking a property between the meta-elements that are not directly linked together. In other words, these rules are complementary to the meta-model cardinality rules.

Early design-time analyses can detect fundamental problems. Therefore, static schematics are important to prevent these fundamental problems. If the designer wants to sample sensor data, it can be forgotten to define that process as continuous. Moreover, during topology design, distance can not be considered, therefore coverage problems may occur. More importantly, messages can not be sent to the next node. This leads to time consumption even if the rest of the configurations are valid. Even if the designer has a verified topology and node distances are feasible for the environment, during modeling, port numbers might be inputted wrong. At last, all of the above problems do not occur, the designer cannot even think about power consumption, when everything works fine, after a while, starting from the relay nodes where the bottleneck is active, the network starts to fall. All of these fundamental problems which may lead to fatal problems can be detected before deployment of the system. The power consumption and bottleneck problems are discussed in section 10.

8.1. Semantic Constraints of the DSML4Contiki

In this sub-section, the semantic constraints that are implemented for the DSML4Contiki are given and are explained. These rules help the designer to check the designed model. If any rule violation occurs, the designer is warned with a message. Not to repeat the same domain rules, only general rules that are applied to every element and the rules that have critical importance for the target system are given in Listing 8.1.

```

1. self.Name.size()>0
2. self.PT_Argument_Name.size() > 0
3. self.No<= 65535 and self.No >-1
4. (self.eContainer().eContainer().oclAsType(contikiOS::IoTSystem). (self.hasSensor->size())>0
   and self.Continuous=true) or (self.hasSensor->size())=0 and (self.Continuous=true or
   self.Continuous=false)

```

Listing 8.1 Semantic constraints of the DSML4Contiki.

Rule 1 checks the size of the Name attribute of an element in case it is empty. This rule can be applied to every element that has a *Name* attribute and for the attribute that must not be left blank. Rule 2 checks the protosocket's message is inputted or not. Even if the message is not entered then the server receives the socket but prints nothing since the message is empty. This may misdirect the designer to think something is not working right. Rule 3 checks whether

a port number is entered below 0 or more than 65535. During design, the designer can enter an integer value that violates the port numbering and this may lead to port mismatch. As a constraint, if the user forgets to enter any port number or enters a number more than 65535 or enters a value lower than 0, then the designer is warned to enter a value between 65535 and 0. Rule 4 is a sophisticated rule, sampling of the sensors is a continuous activity and the process that samples a sensor value must be modeled as a continuous process. So, if there is a source node in the model, this rule checks if the continuous flag is activated or not. If it is not activated, the rule warns the user to define the process as continuous. Otherwise, the sensor process runs only once and will never sample data from the environment (see Figure 8.1).

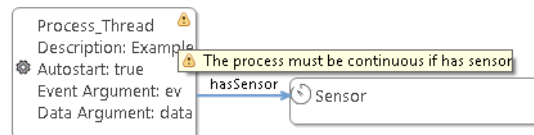


Figure 8.1 A warning message for a process that has a sensor element.

8.2. Semantic Constraints of the DSML4Riot

In this sub-section, the semantic constraints that are implemented for the DSML4Riot are given and are explained. These rules help the designer to check the designed model. If any rule violation occurs, the designer is warned with a message. Not to repeat the same domain rules, only general rules that are applied to every element and the rules that have critical importance for the target system are given in Listing 8.2.

1. (self.Number > 0)
2. (self.UDP and self.TCP and self.socketR->size()>0)
3. (self.LocalPortNumber != self.remotePortNumber)
4. (self.threadR->size() < 5)
5. (not (self.isIPv4 and self.isIPv6))

Listing 8.2 Semantic constraints of the DSML4Riot.

Constraint rule 1 checks whether the *Number* attribute of GPIO element is a negative value or not. The user may input a negative value for the GPIO and the compiler gives an error during the compilation of the code. Constraint rule 2 checks whether TCP, UDP or both of them are used in the model, but the *Socket* element is not used, then it warns the designer to include one. If this element is not included in the model, then TCP and UDP connection cannot be established. The codes that are mentioned in sub-section 5.2 would not be generated. In constraint rule 3, local and remote port numbers are compared in case they have the same number. Because, if these two ports have the same value, then messages cannot be sent. Constraint rule 4 is written to warn the designer for using a lot of threads. If the designer has more than 5 threads, the designer is warned (see Figure 8.2). Because using a lot of threads may lead to poor performance and memory overflow problems. Constraint rule 5 is implemented to direct the designer to use either IPv6 or IPv4. Because enabling these two IP protocols is not considered a wise choice.

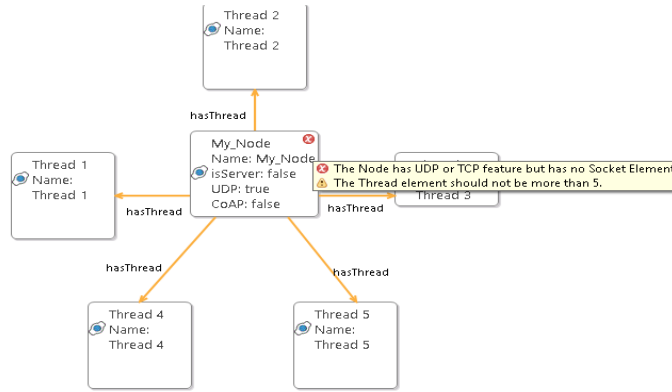


Figure 8.2 A warning and an error messages for Thread and Socket elements.

8.3. Semantic Constraints of the Topology Design Viewpoint

In the topology design viewpoint, two semantic constraints are written. These two semantic constraints are applied to the domain rules. A typical wireless sensor device that uses the IEEE 802.15.4 protocol can transmit a message at most 50 *meters* in the indoor area and at most 120, in the outdoor area (TmoteSky Datasheet, 2019). These two constraint rules check whether a distance value that exceeds the limit distance is inputted (see Listing 8.3).

1. `aql:(self.eContainer().eContainer().oclAsType(contikiOS::IoTSystem).platform.isIndoorArea<>false) and (self.Distance<50)`
2. `aql:(self.eContainer().eContainer().oclAsType(contikiOS::IoTSystem).platform.isIndoorArea<>true) and (self.Distance<120)`

Listing 8.3 Semantic Constraints of the Topology Design Viewpoint.

The first rule accesses the platform element and check the topology design is made for an indoor area or not. Since it is known that most of the WSN devices can transmit a message at most 50 *meters* indoor and 120 *meters* outdoor. This rule warns the designer if the designer inputs a distance of more than 50 *meters* (see Figure 8.3). Similarly, the second rule checks the same condition for the outdoor selection and 120-meter limit distance. These rules apply the domain rules of physical life in the modeling environment. In this way, the designer does not waste time by accidentally placing the nodes out of the range during the physical placement of the nodes. Because, if nodes are not placed in transmission range then they cannot establish a network. As a result, the designer wastes the time to find the problem.

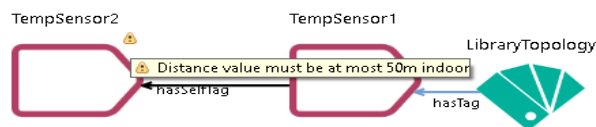


Figure 8.3 A warning message for distance violation.

9. THE OVERVIEW OF THE PROPOSED DSMLs

The process model explains how the methodology helps the user to redesign the topology before deployment and how it avoids after deployment problems (see Figure 9.1). The process model is designed according to the given method in the study of (Mustafiz et al., 2012).

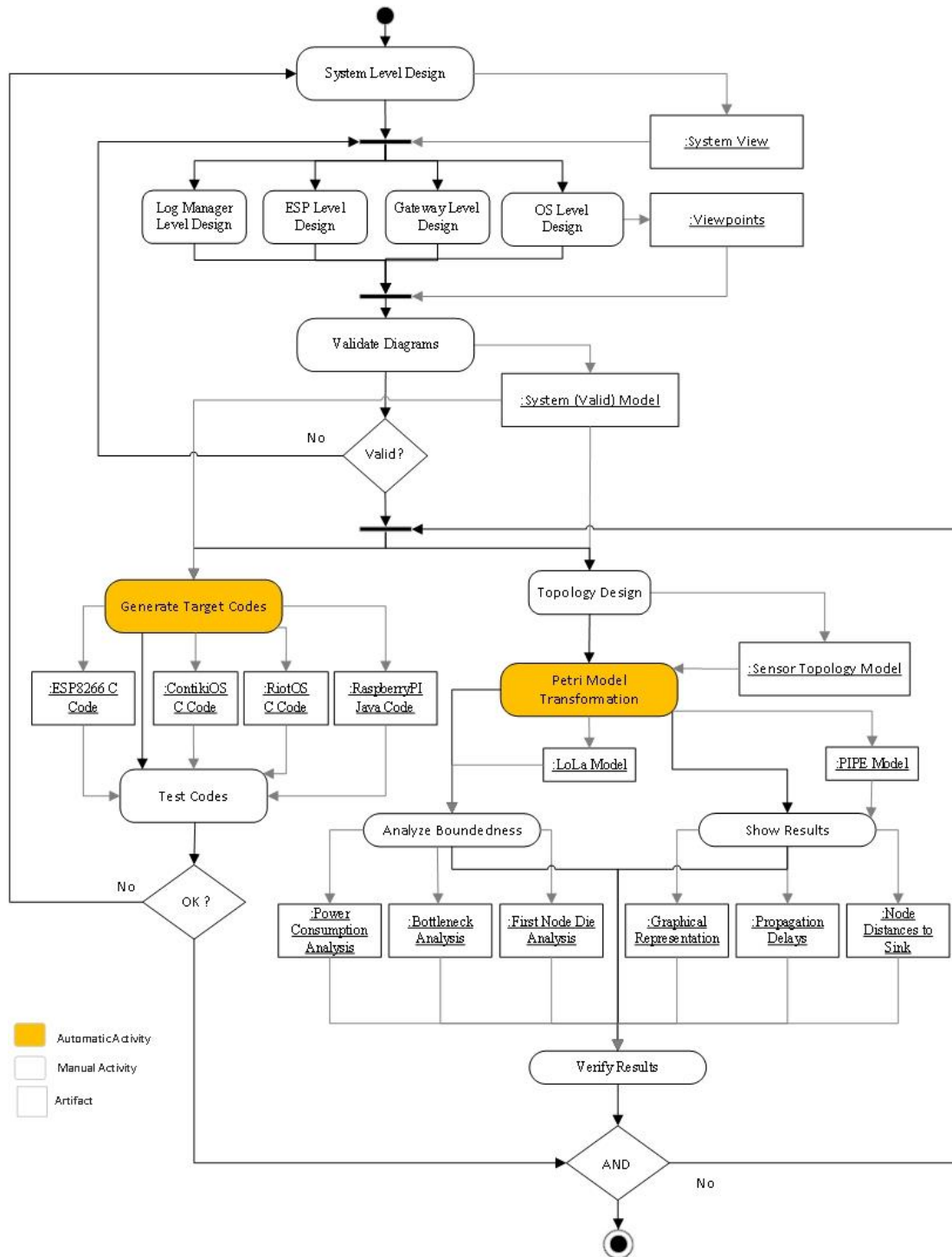


Figure 9.1 Process Model of proposed DSMLs.

The main steps and iterations in the process model are as follows:

- The designer has to design the system level with the domain concepts provided in the modeling environment. This is done after deciding on the requirements, e.g. which sensor data will be gathered or which actuators will be used. The result is a system view model for the IoT system which will be used as input for the next step.
- The next parallel step is the detailed design of the four main viewpoints (representing WSN, IoT, Gateway, and Log Manager) which are defined in the system view (in the early design of the system). This step will provide the viewpoints of the system.
- The detailed design of the system is validated by the DSML4Contiki and DSML4Riot using the constraint checking rules (which is discussed in the previous section) to control the wellformed-ness of the design models. The outcome is a valid and detailed system model in different viewpoints.
- From this point on, the designer can proceed in parallel with the analysis of the system and/or on the synthesis of the system, both from the same validated design model.
- To start with the analysis, the topological information of the system needs to be added to the system. This will extend the system model with topology details which are required for some of the analysis.
- Then the proposed DSMLs can automatically generate the Petri-net model of the system. The Petri-net model is generated both in Platform Independent Petri-net Editor (PIPE) and LoLA Low-Level Petri-net Analyzer (LoLA, 2009). This is the preferred design choice to benefit from the visual presentation of the PIPE and more low-level analysis of LoLA.
- For k-boundedness analysis, the designer decides/calculates a k value to perform power consumption analysis, bottleneck analysis, and node crash/die analysis. Also, some results are shown/reported in the PIPE framework such as propagation delays and relative distance of source nodes to the sink node. The details of these analyses are elaborated with the smart fire detection case study in the next section.

10. TRANSFORMING A TOPOLOGY MODEL TO A PETRI-NET MODEL

It is important to verify the system design model before it is implemented. This assists in the analysis and verification of the designed model in the early phases of the development, before any implementation. Because even a small error can potentially lead to catastrophic system failures that are concerned in a security-critical context. To this end, the design model needs to be transformed into a formalism in which the required properties can be easily and automatically checked. This helps the designer to inspect the model and to check that the structure and/or behavior of the system conform to domain rules. In this way, problems and errors can be detected during the design phase. Petri-nets are specific types of modeling constructs useful in data analysis, simulations, and process modeling (Petri and Reisig, 2008). It can help to plan workflows or present data of any complex system. Petri-net is a formalism used for verification of pertinent properties of automation systems, embedded systems, network systems, etc. In this thesis study, Petri-net is used to check for the occurrence of bottlenecks. To achieve this, the designer instance models of used hardware components are automatically transformed into stochastic Petri-net models in the PIPE and LoLA tools to check the bottlenecks using the k-boundedness feature. For this purpose, the instance model elements that represent hardware in system-level are transformed into places, relations are transformed to arcs, sensors' data are transformed to tokens, and transitions are added to combine two or more arcs. Figure 10.1 shows the transformation mappings.

DSL	Petri-Net
Tag (Sensor Nodes)	Place
ESP8266	Place
IoT Log Manager	Place
RaspberryPI	Place
Node messages	Token
ESP Messages	Token
Element Relations	Transition
Element Relations	Arcs

Figure 10.1 Transformation table of DSML elements to Petri-net elements.

This section focuses on the system analyses of the WSN and IoT based systems using MDE techniques. To this end, the study in (Asici et al., 2019) is also extended in this thesis study. The k-boundedness analysis is adapted to the generated Petri-net model and fire risk is formulated for the smart fire detection case study (Karaduman et al., 2019a, 2019b). Moreover, the analyses for power consumption, bottleneck, and first crashing node of the IoT system are conducted. To this end, propagation delays and nodes' distance to sink are analyzed in the Petri-net models. As a result, the proposed development approach for WSN and IoT based systems is model-centric and not only the code synthesis can be done based on design models, but also they are used to do a lot of analyses on the system, at the design phase. Besides, the engineering workflow applied in the proposed development methodology is elaborated as a step-wise process model.

In figure 10.3, the elements which are used in DSML4Contiki, DSML4Riot, and study of (Asici et al., 2019), are represented as a Petri-net model. As Figure describes, various devices send messages to a centralized computer. In the Petri-net representation, the places represent gateway, wireless devices, sensor nodes, and log manager. The paths of the messages are directed by arcs, and message transitions are added between two places into the Petri-net model. The data messages that are sent by devices are represented by tokens.

10.1. Topology Design in proposed DSML

The designer designs a topology using the *Tag* element. These *Tag* elements can be connected to each other. They represent a node in a WSN or an IoT system. To achieve the *Tag* element to the *Tag* element connection, the *Tag* Eclass is related to a self-reference relation. In this way, a *Tag* element has become capable of having another *Tag* element. Having another *Tag* element implies creating a multi-hop network since each *Tag* element represents a sensor device. Figure 10.2 describes a topology that is designed using *Tag* elements.

Each *Tag* element has a distance attribute. It is the distance relative to the neighbor node. Using this distance attribute, the farthest node to the sink node, each node's distance to the sink node, and propagation delays can be calculated for time-critic systems. To calculate child and parent nodes' distances a Java Wrapper Service is used (along with the Acceleo rules), which benefit from the self-reference relations among the nodes to generate the analysis models. Transforming the designed topology to a Petri-net model, k-boundedness feature of the Petri-net can be applied to achieve some analyses results. In the sub-section 10.2, k-boundedness feature of the Petri-net is explained and in 10.3, Java code to calculate the relative distances is explained.

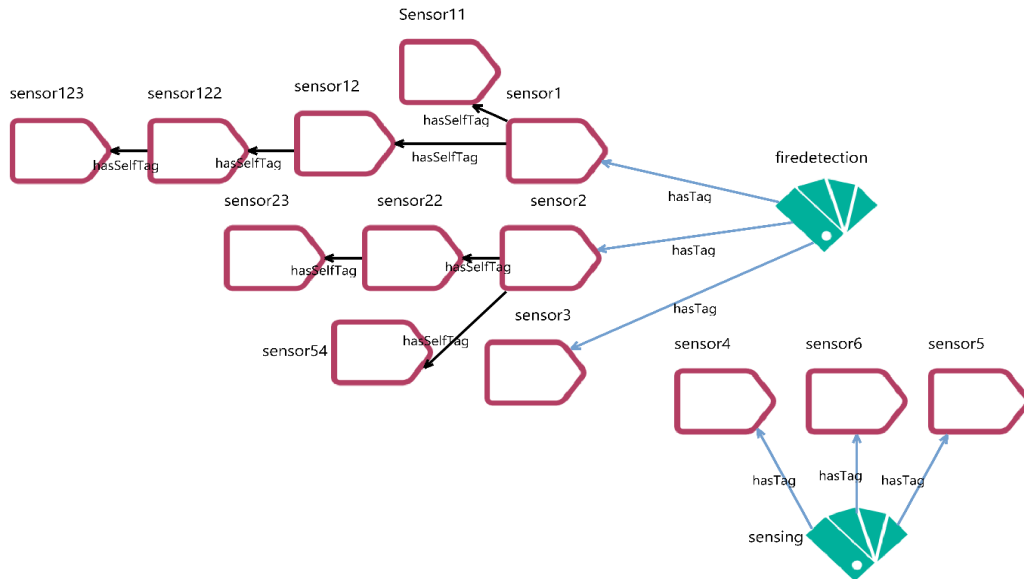


Figure 10.2 A designed topology of a network in the Topology Viewpoint.

10.2. k-boundedness feature of Petri-net

k-boundedness feature checks a place in a Petri-net model and counts the number of tokens that are passed in that place, then it compares the number of tokens with the k value. Within the knowledge, the introduced Petri-net method k-boundedness is not time-based, it is token-based. In short, the analysis is made based on the token's activity in a place. The k value represents the bottleneck of the sensor network. If the number of tokens gets closer to k value in that node (place), bottleneck problem and power consumption increases, since there are k many tokens (messages) pass through there. Therefore, an optimal k value must be found by the designer and topology design must be made according to this value. Moreover, the number of the tokens pass through in a place must be under this k value. If the number of tokens (messages) exceeds this k value in a place, then power consumption goes above of planned at that node. If the power is depleted in a node, a part of the network goes down. Therefore, k value must be reduced and this can be achieved by adding extra nodes to decrease message traffic but this results as an extra cost. Another way is increasing k value. If increase k is increased, then battery capacity has to be increased to ease the bottleneck problem, but this also results in more cost. In two ways, this trade-off must be handled by the designer.

10.3. Analyzing IoT systems with Petri-net models

For simplicity and ease of understanding the Petri-net, analyses are divided into two parts. The first part is realized in PIPE to present a graphical representation of the system, to provide the distance and propagation delay concerning the sink node, and to perform a cost analysis and arrange a list of components for the system. Propagation delays are calculated according to the distance over the speed of light formula in the air medium. The distance value is inputted by the user for each node. The propagation delays are shown in PIPE in the case of time-critic designs. In real-time applications, if there is a huge network, propagation delays might become a problem. The second part of the analysis is realized using the low-level Petri-net capability of LoLA (using k-boundedness for the places) for bottleneck analysis, power consumption analysis, and node crash (first node die) analysis. For the cost analysis, the bill of materials (BOM) is calculated. The BOM is calculated based on the number of components and their prices in the market. In Listing 10.1 the code snippet shows how the calculation is made and the results are generated.

```
<labels x="1051" y="427" width="404" height="150" border="true">
<text>
Bill of Materials
ESP8266 x [aIoTSystem.logman.esp->size()]/ = [4*aIoTSystem.logman.esp->size()]/$
RaspberryPI x [aIoTSystem.raspberrypi->size()]/ = [30*aIoTSystem.raspberrypi->size()]/$
TmoteSky x [aIoTSystem.logman.tag->size()]/ = [100*aIoTSystem.logman.tag->size()]/$
IoT Log Manager x [aIoTSystem.logman->size()]/ = [1000*aIoTSystem.logman->size()]/$
</text>
</labels>
```

Listing 10.1 Calculation of the BOM in Acceleo.

To perform the analyses, the Petri-net models (in PIPE and LoLA) need to be generated from the design models. To transform a domain model to a Petri-net model, transformation rules are implemented using Acceleo. These transformation rules get elements that are used in DSML4Contiki, DSML4Riot, study of (Asici et al., 2019) as inputs and transform them into Petri-net models for PIPE and LoLA. In below, a snippet of LoLA model generation rules (to

create the places) is demonstrated. However, calculating and transforming some of the semantic properties of the design models, such as distances of source nodes and sink node, are not straightforward and cannot be done using solely a template engine (Acceleo) and constraint checking languages (AQL). In fact, it is required an imperative language (such as Java) to do this kind of calculations and later to be transformed into the target model using a declarative transformation language such as Acceleo. The following rule uses the Java wrapper code which implements the *calculateDistance* function.

```
public LinkedHashSet<String> calculateDistance(Collection tag, float size) {

    LinkedHashSet<String> ltag= new LinkedHashSet<String>();
    ArrayList<String>names = new ArrayList<>();
    ArrayList<Tag>aTag = new ArrayList<>(tag);
    for(int i=0;i<tag.size();i++)
        System.out.println(aTag.get(i).getName());

    for(int i=0;i<aTag.size();i++){
        for(int j=0;j<aTag.get(i).getSelftag().size();j++) {

            System.out.println("\n"+aTag.get(i).getName()+"->" +aTag.get(i).getSelftag().get(j).getName()+" " +
            aTag.get(i).getSelftag().get(j).getDistance());
            names.add(aTag.get(i).getName()+"_" +aTag.get(i).getSelftag().get(j).getName());
        }
        for(int i=0;i<aTag.size();i++)
            for(int j=0;j<aTag.get(i).getSelftag().size();j++) {

                System.out.println("\n"+aTag.get(i).getName()+"->" +aTag.get(i).getSelftag().get(j).getName()+"
                "+aTag.get(i).getSelftag().get(j).getDistance());
                names.add(aTag.get(i).getSelftag().get(j).getName()+"_" +aTag.get(i).getName());
            }
        for(int i=0;i<names.size();i++) {
            ltag.add(names.get(i));
            System.out.println(names.get(i));
        }
        return ltag;
    }
}
```

Listing 10.2 Java code of calculateDistance function.

The function simply uses two for loops and access every *Tag* element in the model. For each of the *Tag* element it access the children of a *Tag* elements, then it calculates the distances. Moreover, this function is used to transform a topology model to a Petri-net model. Because a sensor network topology has a tree structure, therefore each element in Figure 10.2 must be transformed to a Petri-net model. To achieve this, each element is accessed using Java code and nested loops. However, to call the *calculateDistance* function, it must be defined in the Acceleo code. To call a Java function in the Acceleo, a query is used (see Listing 10.3).

```
[query public calculateDistance( arg0:OrderedSet(Tag), arg1: Integer ):OrderedSet(String)=
invoke('ContikiOS.acceleo.module.names', ' calculateDistance (java.util.Collection,int)'
,Sequence{arg0,arg1}) /]
```

Listing 10.3 Acceleo Query to trigger calculateDistance function.

This query introduces the Java function to the Acceleo. As it is seen from the code, the Java function returns a variable named *ltag* that is type of *LinkedHashSet*. When a function is called in the Acceleo, its return values must be contained in a variable.

```
[let abc : OrderedSet(String) = aIoTSystem.logman.tag.asd(aIoTSystem.logman.tag,aIoTSystem.logman.tag->size())->asOrderedSet()
[/let]
```

Listing 10.4 A variable in Acceleo code to store return values of the calculateDistance function.

The above code in Listing 10.4, stores the return values. A variable in Acceleo is defined using *let* keyword. Acceleo typecasts the objects to the *String* type. In this way using the *String* functions, any implementation on to these objects can easily made.

10.4. Petri-Net Case Study: Smart Fire Detection System for a Library

As mentioned, IoT and WSN systems are power critical systems. If a node loses its battery power, a part of the network dies off. Most of the battery power is consumed by the antenna. If the antenna is used frequently, the battery will be depleted fast. The antenna is used when a packet is received or it is sent. Therefore, important criteria to analyze these systems is the data sampling and sending period. Generally, when data is sampled, it is sent immediately. To reduce power consumption and increase battery life, designers can increase the sampling and sending period. However, this may violate system requirements in some cases such as sampling temperature sensors for a fire detection case study where the fire situation must be reported in a limited amount of time. However, setting the sampling time very high can bring the risk of bottleneck and also resulting some of the sensors to discharge their batteries fast, resulting in losing the coverage of part of the system in the fire detection example. Therefore, analyzing the trade-off between these two ends and setting an appropriate sampling and sending period is very essential. Specifically, realizing these analyses in the early design phase of the system is important, as it decreases the cost of system development and avoids unforeseen critical errors in the system after deployment. Another parameter that needs to be analyzed is the distances between the nodes in the topology of the IoT system. In this sub-section, these analyses, as well as some other analyses (such as cost analyses and propagation delay analysis), are elaborated using Petri-net models generated from the design models and the data provided by the designer in the design models. To apply the proposed method, the *k* value must be calculated by the designer to apply *k*-boundedness feature of Petri-net. The *k* value can be calculated using the formulas that are given in the next sub-section 10.4.1.

10.4.1. Generating Petri-net Model for the Fire Detection System

In Figure 10.3, the generated Petri-net model of the IoT system for fire detection is shown in PIPE (due to no graphical user interface, the LoLA version is not shown). Figure represents the places, transitions, and arcs for a multi-hop WSN for the system. The initialization of the tokens is made according to the sensing activity of the nodes. If a node (place) has a sensor data and has to transmit this data, then that node (place) must have a token. The information to generate this Petri-net model is transformed from the domain models that are designed in a topology, e.g. the diagram in Figure 10.2. The bill of materials is shown on the right-hand side of Figure 10.3 with the number of materials that are used in the design. Moreover, concerning the sink node, the distance and propagation delay results are shown in a box in the given figure.

These calculations are done based on the information provided by the designer in the design models.

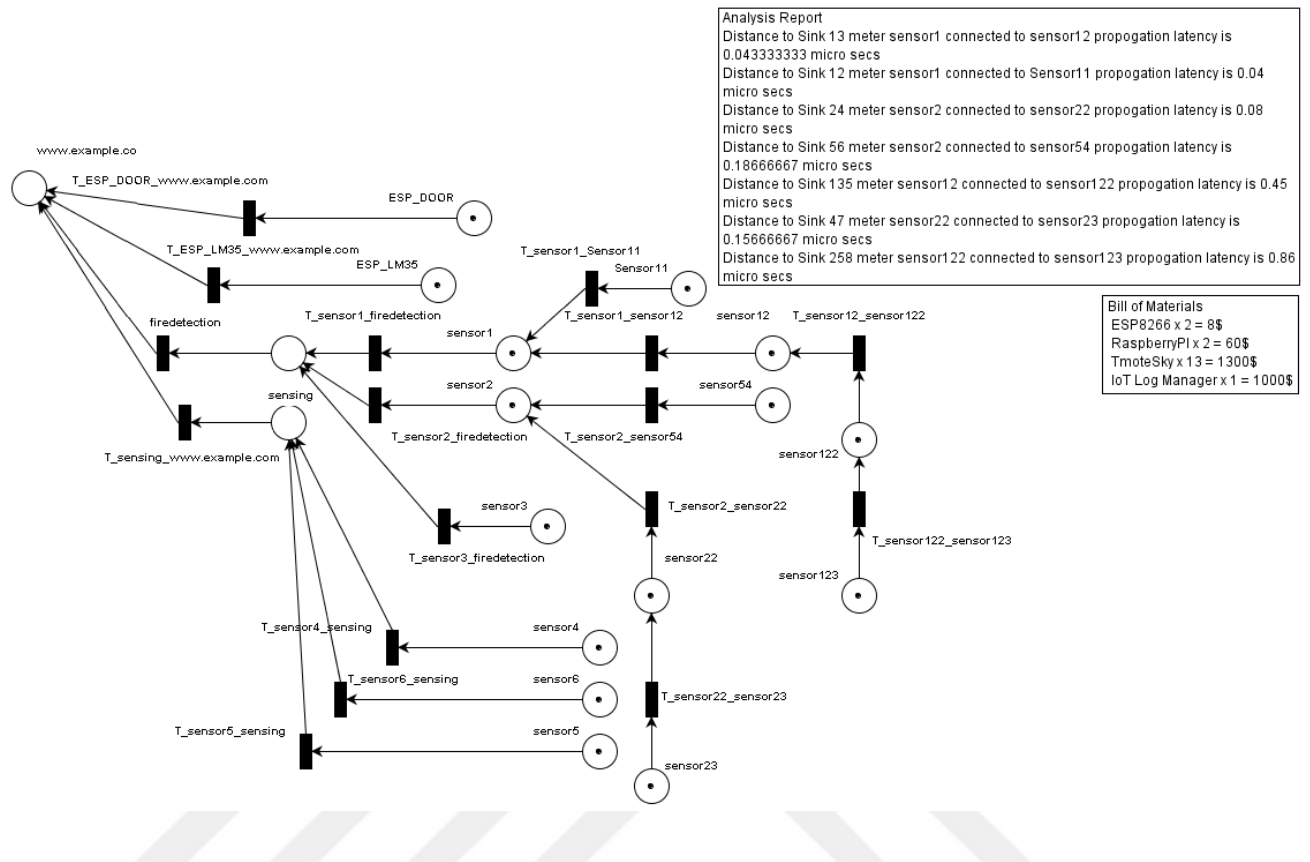


Figure 10.3 Petri-Net model of a designed topology.

The following sub-section analysis method of a designed topology using the Petri-net model is explained. Firstly, transformation rules are given to transform a topology model to a LoLA configuration file and LoLA's file format to create a configuration file is mentioned. Secondly, the same steps are mentioned and explained for the PIPE tool. Finally, the exemplary use of these two tools is described in detail via a fire detection system.

10.5. LoLA Model-to-Code Transformation

As mentioned before, to analyze the designed topology, it must be transformed into a configuration file that can be interpreted by LoLa. For this purpose, the *Tag* element that represents a node in the topology viewpoint is transformed to places, relations are transformed to arcs, sensor data is transformed to a token, and transitions are added to combine two or more arcs. The code generation rule to have a configuration file starts with the rule of all places. They are separated by a comma and terminated by a semicolon. Then the initial token is defined using the same syntax. The numbers that represent the token is generated after a colon. To transfer a token to another place, the number of the token that is sent must be defined using *PRODUCE* and *CONSUME* keywords. Code snippet in Listing 10.5 represents the code generation rules of the configuration file for the LoLA low-level Petri-net analyzer. Even if the

LoLA does not present a graphical user interface, it provides many analyses to achieve a lot of results for the domain.

```
[file (Name.concat('.lola'),false,'UTF-8')]

PLACE

[for (l:LogMan | aIoTSystem.logman)][l.URL/].[for (e:ESP | aIoTSystem.esp)]
[e.Name/],
[/for]
[/for]

[for (l:LogMan | aIoTSystem.logman)][for (t : Tag | tag)][t.Name/],
[/for]
[/for]

[for(t:Tag|aIoTSystem.raspberrypi.tags )]
[if (c.Name=r.Name)]
[if(c.tagss->at(i).Name.toString()<>'invalid')]

TRANSITION [r.Name+'_'+c.tagss->at(i).Name.toString()/]
CONSUME [c.tagss->at(i).Name/]:1;
PRODUCE [r.Name/]:1;
[/if]
[/if]
[/for]

[for (l:LogMan | aIoTSystem.logman)]
[for (r:RaspberryPi | aIoTSystem.raspberrypi)]

TRANSITION [r.Name+'_'+l.URL/]
CONSUME [r.Name/]:1;
PRODUCE [l.URL/]:1;

[/for]
[/for]

[let abc : OrderedSet(String) = aIoTSystem.logman.tag.calculateDistance(aIoTSystem.logman.tag,aIoTSystem.logman.tag-
>size())->asOrderedSet()]
[for(t:String|abc)]
[if (i<=abc->size()/2)]

TRANSITION [t.substring(1, t.index('_')-1)/_[t.substring(t.index('_')+1, t.size())]/]
CONSUME [t.substring(t.index('_')+1, t.size())/]:1;
PRODUCE [t.substring(1, t.index('_')-1)/]:1;
[/if]
[/for]
[/let]
[/file]
```

Listing 10.5 Rules to generate LoLA code.

In the following sub-section 10.6, code generation rules for the PIPE Petri-net tool is mentioned. Since the PIPE provides a graphical user interface, the same topology model is transformed to PIPE's configuration file. The consistency between the two transformations was tested by counting the number of tokens in each place in both models.

The k-boundedness feature is applied in LoLA and the tokens were transferred manually in the PIPE for a topology model. As a result, when the number of tokens reached the last place in PIPE was equal to the same number of tokens of the last place in the LoLA model, the

consistency between the two models is satisfied since they have given the same result for all places.

10.6. PIPE Model-to-Code Transformation

As mentioned before, to visually represent the designed topology, it must be transformed into a configuration file that can be interpreted by PIPE. For this purpose, the *Tag* element that represents a node in the topology viewpoint is transformed to places, relations are transformed to arcs, sensor data is transformed to a token, and transitions are added to combine two or more arcs. The code generation rule to have a configuration file starts with the generation of places. They are defined using *place id* label. To define the position of a place *position* label is used. The initialization number of token is written between *value* labels. The code snippet in Listing 10.6 represents the code generation rules of the configuration file for the PIPE Petri-net analyzer.

```
[for (l:LogMan | aIoTSystem.logman)]
[for (t : Tag | tag)]

<place id="[t.Name/]">
<graphics>
<position x="[760+(i*20)/]" y="[300+(i*50)/]" />
</graphics>
<graphics>

<value>1</value>
</place>
[/for]
[/for]

[for (c: Channel | channel)]
<transition id="T_[c.Name/]_[l.URL/]">

<graphics>
<position x="435" y="[315+(i*50)/]" />
</graphics>

<value>T_[c.Name/]_[l.URL/]</value>
<graphics>
</transition>

<arc id="[c.Name/] to T_[l.URL/]" source="[c.Name/]" target="T_[c.Name/]_[l.URL/]">
</arc>

<arc id="[c.Name/] to T_[l.URL/]" source="T_[c.Name/]_[l.URL/]" target="[l.URL/]">
</arc>

[/for]
```

Listing 10.6 Rules to generate PIPE code.

To transfer a token to another place, two arcs and a transition must be defined between two places. The token is directed via these arcs and two arcs are connected by a transition. Since the PIPE tool presents a graphical user interface, it provides a visual representation of the designed topology and presents animation modes there the tokens can be moved manually. Using the animation mode, the path of the tokens can be tracked and token changes in the places can be observed.

10.7. Analyzing the Fire Detection System with Petri-net

The bottleneck can be detected using k -boundedness, the system can be transformed to another model like Petri-net using model-to-model transformation without modeling a system from scratch and power analysis can be made using Petri-net. The nodes can be analyzed for power consumption as they may go out of battery and cause a part of the network to go down. Therefore, the designer can go back to the design and can devise alternatives, such as adding new connections, new nodes or increasing battery capacity. Moreover, if there is a bottleneck problem on a node, the network lifetime, which has already planned, may not be valid anymore. Because of the complexity of node paths, in a distributed network the designer cannot predict how many times a node receives and transmits message packets. Therefore, to run a system for the defined period, the battery capacity must be increased or extra nodes must be added for creating new paths to ease the message traffic, thus the impact of the bottleneck problem is fixed. To keep the planned lifetime of the network, the bottleneck must be analyzed before establishing the network in the physical environment. The Petri-net model is power-balanced if the number of tokens passes through in each place does not exceed one, which means a 1-bounded model. This corresponds to a single-hop sensor network. However, due to environmental constraints, a single-hop network is not always establish-able. Therefore, a multi-hop network is usually designed, in which case the network will be k -bounded. If k increases, the bottleneck problem, power consumption, and battery cost are increased. A k value must be found according to provide the desired lifetime with planned battery capacity in the system. Some of these analyses are discussed below for the fire detection system. The load in a node is the number of messages to be received and/or transmitted to the next node. Naturally, the message load of a node increases when the node has more sub-nodes (or neighbor nodes). This creates a bottleneck and results in more power consumption, which can result in a node die earlier than the planned lifetime. Moreover, a node can have message overflow due to the limited buffer (as these devices are constraint system with limited resources). To solve these two problems, an optimal k value must be calculated and the whole network must be bounded by this k value. k represents the maximum value that must be passed through in a node (place). To find the k value, the number of messages which are received and transmitted in a node should be considered, and it can be calculated using the formula 1. The value of k is multiplied with two since a node that receives a message should transmit it to the next node and the node sends its message. In some cases, a node can only receive a message and it may drop the packet to save energy. However, in the fire detection system, the nodes transmit all the messages they receive. Power consumption depends on the number of operations of each node makes. Therefore, to bound the network with a value of k , the designer first should decide on the nodes' lifetime, the battery capacity and the message sampling and sending period.

$$1. \text{NumberOfOperations} = (2 * k) + 1$$

If the designer increases battery capacity, then the lifetime of the node increases. The power consumption of receiving and transmitting operations are averaged ($RxTxAvgmA$) since a node transmits what it receives. In short, a node makes a transmission for a reception. Therefore, the average of receiving and transmitting operations can be averaged to ease the calculation. T represents the hourly usage period of the antenna and it is bound to the sampling period and can be calculated using the following formula. Since our Petri-net analyses are based on tokens instead of time, if the system's lifetime is determined then k value can be found to analyze the Petri-net model in LoLa.

$$2. \quad T = \frac{3600}{\text{SamplingPeriodseconds}}$$

TmoteSky nodes are used for the library fire detection system and a multi-hop ad-hoc network is implemented. To achieve the above formulas, the power consumption of transmission and reception values are found from the TmoteSky datasheet. In the fire detection case study, the nodes are set to sample and send the temperature value each 120-second. According to the above equation, the nodes use their antenna 30 times in an hour. In the case study, a 5-volt (10000mAh) power-bank was used as the power source, which can provide 33000mAh since TmoteSky runs at a voltage level of 3.3v. The battery capacity is 33000mAh in this case study. To ease the calculations on the power consumption side, the average of transmitted and received current consumption for TmoteSky is calculated (based on the data-sheet), which is approximately 20mA. In 2, if k is 5, a node with 5 sub-nodes (see formula 1) can run for 5 hours. It should be considered that k = 5 is a very highly connected network with very high reliability. These manual calculations show how to perform the trade-off analyses between k and lifetime of the network (via power consumption). The lifetime of a node can be calculated using formula 2.

$$3. \quad \text{LifeTimehour} = \frac{(\text{BatteryCapacitymAh})}{\text{NumberOfOperations} * \text{RxTxAvg(mA)} * T}$$

In the system design phase, the network size can be huge for where it is not easy to track the paths and do the analyses manually, so to test the k value and to bound the network, the proposed automatic analyses in LoLA can be used. If a node cannot be bounded with given k value, the battery capacity must be increased or the number of child nodes of that node must be reduced. The lifetime of the nodes can also be increased with logical controls, e.g if the temperature value reaches a certain set point, the node sends a message once, not periodically. If the distance to the sink node is too far for a node in a multi-hop network, the system's total propagation delay for a message might become a problem. If the distance increases, the propagation delay increases. Although there is a processing delay in the nodes, in Contiki operating system, the messages are processed in a trivial amount of time. Consequently, the time to trigger the fire alarm needs to be calculated concerning the node which has the furthest distance to the sink node. Anyway, the time required to realize an action, e.g. send a notification to the fire department, is another delay which should be considered by the designer. These kinds of trade-off analyses can be done by the designer.

11. EVALUATION

Evaluation of modeling languages is done by assessing its code generation and PIM-to-PSM transformation capabilities. This was done by comparing the lines of generated code and final code (the code after adding the end-user's delta code) and number of PIM elements and number of PSM elements. Moreover, to have more accurate result, numbers of attributes that are not modified by the designer is evaluated as a successful transformation. A focus on both the code generation and model generation performance is made. Following tables show the comparison results of the number of generated lines of code with the final number of lines of code (Challenger et al., 2016). In the awareness of the evaluation with a single case study may not achieve a comprehensive evaluation, it is done with multiple case studies. In the following listings "... " means that the delta codes are not added consecutively.

11.1. Evaluation of DSML4Contiki for Model-to-Text Transformation

To evaluate more comprehensive, the code generation capability of the DSML4Contiki two case studies were implemented. Codes are generated according to given models and rules in section 5. UDP, Event, and Event Timer features are used mostly in these two case studies. Most of the code to create the smart irrigation case study and smart lighting system are generated by DSML4Contiki.

11.1.1. Evaluation of Smart Fire Detection System

Using the elements of the DSML4Contiki on the server-side and on the client-side, most of the code could be generated for the smart fire detection case study. The following listing 11.1 shows delta codes that are added to the server-side.

```
PRINT6ADDR(&UIP_IP_BUF->srcipaddr);
PRINTF("\n");
PRINT6ADDR(&Server_Connection->ripaddr);
PRINTF(" local port and remote port %u and %u \n", UIP_HTONS(Server_Connection->lport), UIP_HTONS(Server_Connection->rport));
..
...
else
{
PRINTF("Repairing the Topology\n");
rpl_repair_root(RPL_DEFAULT_INSTANCE);
}
```

Listing 11.1 Delta codes of the server side.

Print6Addr function is used to print the sender's IP address. Next, the *Printf* function is called to print a space between the texts. *Print6Addr* function is again is called to device's IP address. These are important parameters for debugging purposes. Because when a packet arrives to the server, the sender's IP address of that packet must be equal to the connection's remote IP address. In this way, a reply message can be sent to the correct address. The *print* function is called for the last time to print local and remote port addresses. *Else* statement and in its scope, the network topology is repaired in case of addition or subtraction of a node. This *else* statement runs when any event is generated.

```

PROCESS_WAIT_EVENT();
printf(" (msg: %s)\n", buf);
PRINT6ADDR(&Client_Connection->ripaddr);
..
....
if(etimer_expired(&Etimer_Broadcast))
{
    Timeout_Event_handler();
    etimer_reset(&Etimer_Broadcast);
}
if(ev==tcpip_event)
{
    tcpip_event_handler();
}

```

Listing 11.2 Delta codes of the client side.

On the client-side, *PROCESS_WAIT_EVENT* function is added to *BroadCast Process* (see Listing 11.2). In this way, the process waits until an event is generated. The *printf* function is added to inform the incoming message and *Print6Addr* function is used to print the sender's IP address. The rest of the delta code is moved to other lines. Additionally, *Timeout_Event_handler* function is moved into the scope of the *If* statement.

During the evaluation of the code generation success of smart fire detection case study, moving a code line is treated as a delta code line. 54 lines of client-side code are generated successfully. 12 lines of code are either added by hand or moved to another line. As a result, 54 lines of code are generated successfully over 66 lines. The code generation success is %81 on the client-side.

In the server-side, 52 lines of code are generated successfully. 9 lines of code are either added by hand or moved to another line. As a result, 52 lines of code are generated successfully over 61 lines. The code generation's success is %85. It shows that most of the code to create the smart fire detection system case study could be generated using DSML4Contiki.

11.1.2. Evaluation of Smart Lighting System

Using the elements of the DSML4Contiki on the server-side and in the client-side, most of the code could be generated for the smart lighting case study. The following listing 11.3 shows delta codes that are added to the server-side. The delta codes of the server-side are the same as the delta codes of the smart fire detection system. Since in the two case studies, the server-side has the same role, they have the same delta codes. In general, the server-side collects the data, creates an RPL network, and sends a reply to the clients. Therefore code success is %85 for the server-side of the smart lighting system.

```

PRINT6ADDR(&UIP_IP_BUF->srcipaddr);
PRINTF("\n");
PRINT6ADDR(&Server_Connection->ripaddr);
PRINTF(" local port and remote port %u and %u \n", UIP_HTONS(Server_Connection->lport), UIP_HTONS(Server_Connection->rport));
..
...
else
{
PRINTF("Repairing the Topology\n");
rpl_repair_root(RPL_DEFAULT_INSTANCE);
}

```

Listing 11.3 Server side's delta codes.

Delta codes of the smart lighting system's client-side are same as the smart fire detection's (see Listing 11.4). Additionally, an *add_u* variable is defined. This variable contains the IP address of the actuator node. Even if the IP address of the server is sent to clients and the clients send the data packet to that IP address of the server, the actuator node actually another server that is registered with another port number in the topology. Therefore, a packet must be sent to a specific IP address and port. To achieve this, the client-side use *simple_udp_register* function to create a connection with the given IP address and port. Next, an information message and IP address of the Actuator node are printed. Lastly, an *If* statement checks which message is going to be sent according to the intensity level.

```

PROCESS_WAIT_EVENT();
printf(" (msg: %s)\n", buf);
PRINT6ADDR(&Client_Connection->ripaddr);
..
...
if(etimer_expired(&Etimer_Broadcast))
{
Timeout_Event_handler();
etimer_reset(&Etimer_Broadcast);
}
if(ev==tcpip_event)
{
tcpip_event_handler();
}
..
...
static uip_ipaddr_t addr_u;
simple_udp_register(&unicast_connection, 8888, NULL, 8888, receiver);
uip_ip6addr(&addr_u, 0xfe80, 0, 0, 0, 0xc30c, 0x0000, 0x0000, 0x0003);
printf("sending unicast to ");
PRINT6ADDR(&addr_u);
printf("\n");
if(value[2]>100)
simple_udp_sendto(&unicast_connection, "on", strlen("on") + 1, &addr_u);
else
simple_udp_sendto(&unicast_connection, "off", strlen("off") + 1, &addr_u);

```

Listing 11.4 Client side's delta codes.

Listing 11.5 shows the delta codes of the actuator node. The text variable is defined to store the incoming message. If and else if statements check which keyword is received. According to the incoming keyword, the green LED is either turn on or turn off.

```

char *text=0;
text=(char*)data;
if(strcmp(text,"on")==0){
leds_on(LED_GREEN);
printf("it is on");
}

```

```

}
else if(strcmp(text,"off")==0){
    leds_off(LED_GREEN);
    printf("it is off");
}

```

Listing 11.5 Actuator nodes' delta codes.

During the evaluation of the code generation success of smart lighting case study, moving a code line is treated as a delta code line. 57 lines of client-side code are generated successfully. 22 lines of code are either added by hand or moved to another line. As a result, 57 lines of code are generated successfully over 79 lines. The code generation success is %72 on the client-side. On the server-side, 52 lines of code are generated successfully. 9 lines of code are either added by hand or moved to another line. As a result, 52 lines of code are generated successfully over 61 lines. The code generation's success is %85. In the actuator node, 10 lines of code added manually and 32 lines of code are generated. As a result, 32 lines of code is generated successfully over 42 lines. The code generation success is %76 for the actuator node. It shows that most of the code to create the smart lighting case study could be generated using DSML4Contiki.

11.2. Evaluation of DSML4Riot for Model-to-Text Transformation

To evaluate the code generation capability of the DSML4Riot, two case studies were implemented. Codes are generated according to given models and rules in section 5. UDP, IPv6, and GPIO features are used in these two case studies. Most of the code to create the smart irrigation case study and smart lighting system are generated by DSML4RIOT.

11.2.1. Evaluation of the Smart Irrigation Case Study

Only using a few elements in server side and in client side, most of the code could be generated for the irrigation case study. The following listing 11.6 shows delta codes that are added to the server side.

```

#include <string.h>
puts("A message is received ");
puts(message);
if(strcmp(message,"Dry")==0)
{
    gpio_set(g5);
    xtimer_sleep(3);
    gpio_clear(g5);
}

```

Listing 11.6 Delta codes of the server side.

The *string.h* library is included to use the *strcmp* function. The *puts* function is called to print the incoming messages to the screen. This also helps for testing purposes. The generated *GPIO* functions are moved to the necessary lines in the target code. The *xtimer_sleep* function is called for three seconds delay. If client sends *Dry* keyword to the server, then server sets HIGH GPIO number 5, waits for 3 seconds then sets the GPIO number 5 to LOW. Listing 11.7 shows client side's delta codes.

```

if(gpio_read(g15)==0)
{
}
xtimer_sleep(1);
gpio_toggle(g2);
printf("%d \n",gpio_read(g15));
xtimer_sleep(1);

```

Listing 11.7 Delta codes of the client side.

In the client-side, GPIO number 15 is checked, if the moisture sensor sends digital LOW. When GPIO number 15 equals zero then the client sends *Dry* keyword to the server. GPIO number 2 is toggled to inform about system activity. The *xtimer_sleep* function is generated by DSML, but its place is moved to another line. During the evaluation of the code generation success of the smart irrigation case study, moving a code line is treated as a delta code line. 65 lines of client-side code are generated. 10 lines of code are either added by hand or moved to another line. Additionally, the *strcmp* function also counts as an extra delta code. As a result, 65 lines of code are generated successfully over 75 lines. The code generation's success is %86.6. On the server-side, 49 lines of code are generated successfully. 7 lines of code are either added by hand or moved to another line. As a result, 49 lines of code are generated successfully over 56 lines. The code generation's success is %87.5. It shows that most of the code to create the smart lighting case study could be generated using DSML4Riot. Compared to the smart irrigation case study, the smart lighting case study has more delta codes.

11.2.2. Evaluation of the Smart Lighting Case Study

Only using a few elements in server-side and in client-side, most of the code could be generated for the smart lighting case study. The following listing 11.2 shows delta codes that are added to the server-side.

```

#include <string.h>
puts("A message is received ");
puts(message);
if(strcmp(message,"o")==0)
{
    gpio_set(g2);//open the lights
}
else if(strcmp(message,"f")==0)
{
    gpio_clear(g2);//close the lights
}

```

Listing 11.8 Delta codes of the server side.

In the server-side of the smart lighting case study, if the server receives *o* character then it sets the GPIO number to HIGH to open the light and it closes the light when it receives *f* character. The following listing 11.9 shows the delta codes on the client-side.

```

char msg=0;
xtimer_sleep(3600);//sample period is 1 hour.
if(gpio_read(g15)==0)
{
    msg="o";
}
else
{

```

```
msg="f";
}
```

Listing 11.9 Delta codes of the client side.

According to the above listing 11.9, it samples the environment light intensity one hour periodically and checks the result of the light-dependent resistor from GPIO number 15. According to the result, it decides which message is sent to the server. During the evaluation of code generation success of DSML4Riot in the smart lighting system, moving a code line is treated as a delta code line. Additionally, the *gpio_read* function also counts as an extra delta code. 49 lines of the client-side code are generated. 11 lines of code are either added by hand or moved to another line. On the server-side, 46 lines of code are generated. 12 lines of code are either added by hand. Additionally, the *strcmp* function also counts as an extra delta code. The generation performance of the DSML4Riot for this case study is %81.6 on the client-side and %79 on the server-side.

11.3. Evaluation of Platform-Independent Model to Platform-Specific Model Transformation

To evaluate the case studies, the generated elements and their attributes are counted. The generated element itself is also counted as an attribute. An attribute that does not require manual completion or any input is count as a successful transformation. Since any relation is known where to be connected, they are not included in the evaluation process.

11.3.1. DSML4Contiki: Evaluation of Smart Fire Detection System Model

The below table 11.1 shows the evaluation table of the smart fire detection system model. It should be known that only the attributes that are actively used are evaluated and compared with the generated PIM model from PIM. Additionally, element's itself is counted as an attribute. The total 15 elements are used in the PSM and 23 attributes are set correctly over 36. The model generation's success is %63.8.

Element Name & Type	Number of attributes used in Element	Number of attributes generated and set correctly
Client Node	2	2
Server Node	2	2
BroadCast Process	3	3
MyServer Process	3	3
MeasureTemperature Process	3	3
Temperature Sensor	2	2
ETimer Broadcast	2	2
Server Connection	2	2
Client Connection	2	2
Sink Port	2	2
TCPIP Event (Client Node)	4	0
TCPIP Event (Server Node)	4	0
Data	2	0

Timeout Event	2	0
Broadcast_Wait_Event_Until	1	0
15	36	23

Table 11.1 Evaluation table of the smart fire detection system model.

Since events and data elements are technical and platform-specific terms, these do not correspond to any element in the PIM level. Therefore they are not generated. However, even if the model generation success is moderate, the code generation success is more important since it has a determinant role for the realization of the target system.

11.3.2. DSML4Contiki: Evaluation Smart Lighting System Model

The below table 11.2 shows the evaluation table of the smart lighting system model. It should be known that only the attributes that are actively used are evaluated and compared with the generated PIM model from PIM. Additionally, element's itself is counted as an attribute. The total 14 elements are used in the PSM and 25 attributes are set correctly over 36. The model generation's success is %69.4.

Element Name & Type	Number of attributes used in Element	Number of attributes generated and set correctly
Client Node	2	2
Server Node	2	2
Actuator Node	2	2
Actuator Process	4	3
Process Thread (Server)	3	3
Process Thread (Client)	3	3
Light Sensor	2	2
ETimer (Client)	2	2
Server Connection	2	2
Client Connection	2	2
MyPort Port	2	2
TCPIP Event (Client Node)	4	0
TCPIP Event (Server Node)	4	0
Timeout Event	2	0
14	36	25

Table 11.2 Evaluation table of the smart lighting system model.

Since *Actuator Node* elements require manual activation of the unicast feature to listen to the incoming packets on a specific port, three of the features are set correctly. Moreover, because of events and data elements are technical and platform-specific terms, they are manually added in the Platform-Specific level. However, even if the model generation success is moderate, the code generation success is more important since it has a determinant role for the realization of the target system.

11.3.3. DSML4Riot: Evaluation of Smart Irrigation System Model

The below table 11.3 shows the evaluation table of the smart irrigation system model. It should be known that only the attributes that are actively used are evaluated and compared with the generated PIM model from PIM. Additionally, element's itself is counted as an attribute. The total 6 elements are used in the PSM and 10 attributes are set correctly over 20. The model generation's success is %50.

Element Name & Type	Number of attributes used in Element	Number of attributes generated and set correctly
Server Node	3	3
Client Node	3	3
Socket	5	4
GPIO 5	3	0
GPIO 15	3	0
GPIO 2	3	0
6	20	10

Table 11.3 Evaluation table of the smart irrigation system model.

Since the *Socket* element's message must be inputted by the designer in the PSM level, it cannot be automatically set. Moreover, because of GPIO element requires technical information of the target platform, the GPIO elements must be manually added to the Platform-Specific level. Even if the model generation success is moderate, the code generation success is more important since it has a determinant role for the realization of the target system.

11.3.4. DSML4Riot: Evaluation of Smart Lighting System Model

The below table 11.4 shows the evaluation table of the smart lighting system model (see Figure. It should be known that only the attributes that are actively used are evaluated and compared with the generated PIM model from PIM. Additionally, element's itself is counted as an attribute. The total 5 elements are used in the PSM and 10 attributes are set correctly over 17. The model generation's success is %58.8.

Element Name & Type	Number of attributes used in Element	Number of attributes generated and set correctly
Server Node	3	3
Client Node	3	3
Socket	5	4
GPIO 5	3	0
GPIO 15	3	0
6	17	10

Table 11.4 Evaluation table of the smart lighting system model.

Since the *Socket* element's message must be inputted by the designer in the PSM level, it cannot be automatically set. Since the GPIO element requires technical information on the target platform, the GPIO elements have to be manually added to the Platform-Specific Model. Even if the model generation success is moderate, code generation success is more important since it has a determinant role for the implementation of the target system.

11.4. Evaluation Results of Code Generation

In this sub-section evaluation results of code generation of DSMLs were illustrated and discussed. The evaluation was made according to numerical results that were given in sub-section 11.1 and 11.2. Firstly, the code generation capability of each DSML on their related case study was evaluated. Secondly, the success rate of the code generation capability of the DSMLs for each target node was calculated and illustrated by the bar charts. Lastly, the average and success rate of the code generation for each target nodes' end code, generated code, delta code, generated code, delta code was illustrated by bars.

11.4.1. Evaluation Results of Code Generation of the DSML4Contiki

Figure 11.1 shows the end code, generated code, delta code, and success rate. Blue bars represent the server-side, red bars represent the client-side, and green bars represent the average of the two sides.

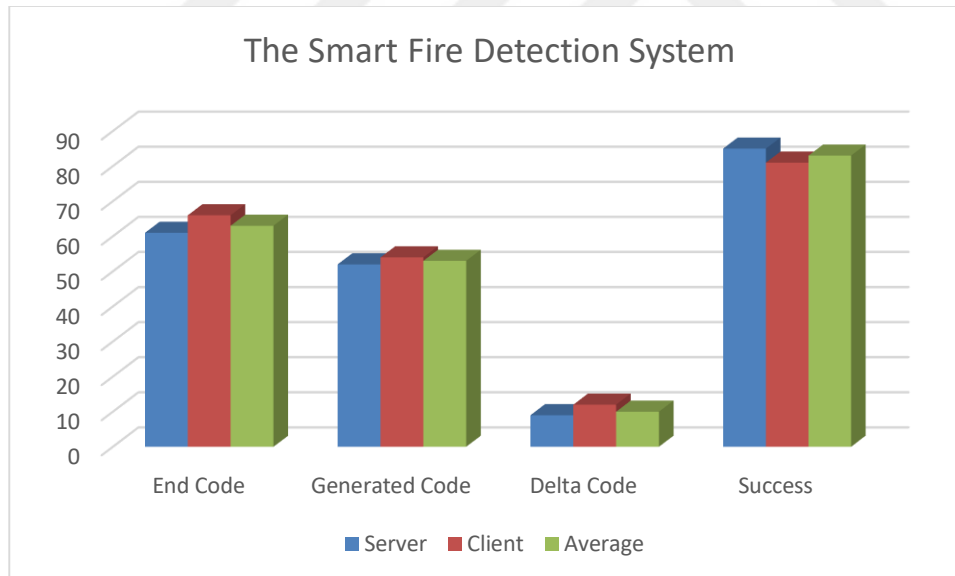


Figure 11.1 Bar chart of the smart fire detection system.

The red bar of the generated code shows that more code lines were generated for the client-side compared to the server-side. On the other hand, it required more delta codes compared to the server-side. The success rate shows that the code generation for the server-side is a little bit more than the client-side.

Figure 11.2 shows the end code, generated code, delta code, and success rate. Blue bars represent the server-side, red bars represent the client-side, green bars represent the actuator side, and purple bars represent the average of the three sides.

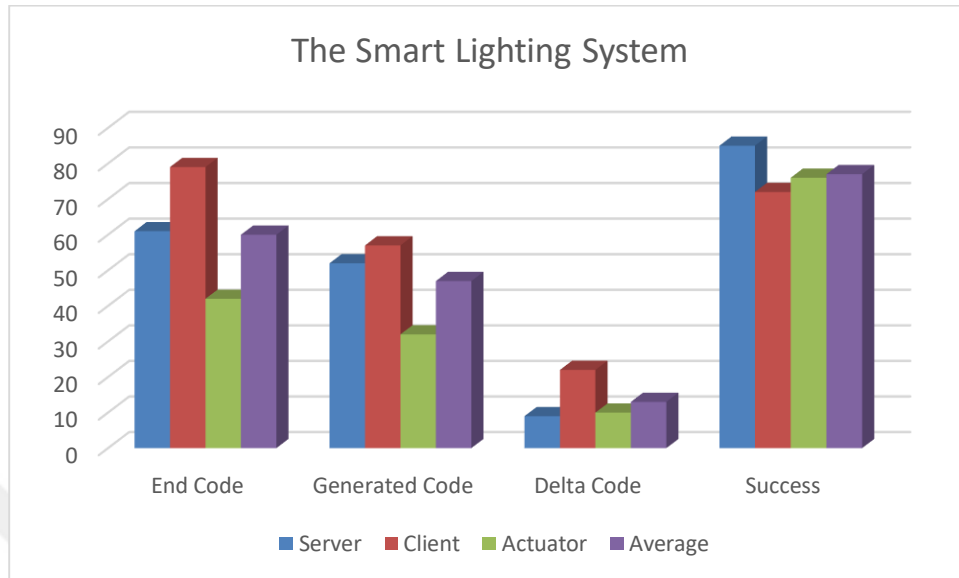


Figure 11.2 Bar chart of the smart lighting system.

The red bar of the generated code shows that more code lines were generated for the client-side compared to the server-side and actuator-side. On the other hand, it required more delta codes compared to the server-side and actuator-side. The success rate shows that the code generation for the server-side is a little bit more than the client-side. On the actuator side, even if the least code is generated for it, it required lesser delta code compared to the server-side. These results show that the need for the delta code depends on the requirements of the target domain and the roles of the target elements such as server, client, and actuator. Moreover, a specific role of a sensor node may require more or fewer delta codes compared to typical roles like server-client architecture.

If the common requirements of the roles of the sensor devices can be determined successfully, the more successful code generation can be made. This can be achieved by focusing on the backbone of WSN systems. In this thesis study, the focus is mostly made on, communication protocols, threads, and sensors on the WSN side. These are the main requirements to create a DSML for WSN based systems. Moreover, a high success rate also requires providing modular code generation and executable code including these features. Because without modularity, the designer has to move the generated code to the necessary code lines or try to delete unnecessary code lines. Surely, this can result in adding more delta codes and lowering the success rate.

11.4.2. Evaluation Results of Code Generation of the DSML4Riot

Figure 11.3 shows the end code, generated code, delta code, and success rate. Blue bars represent the server-side, red bars represent the client-side, and green bars represent the average of the two sides.

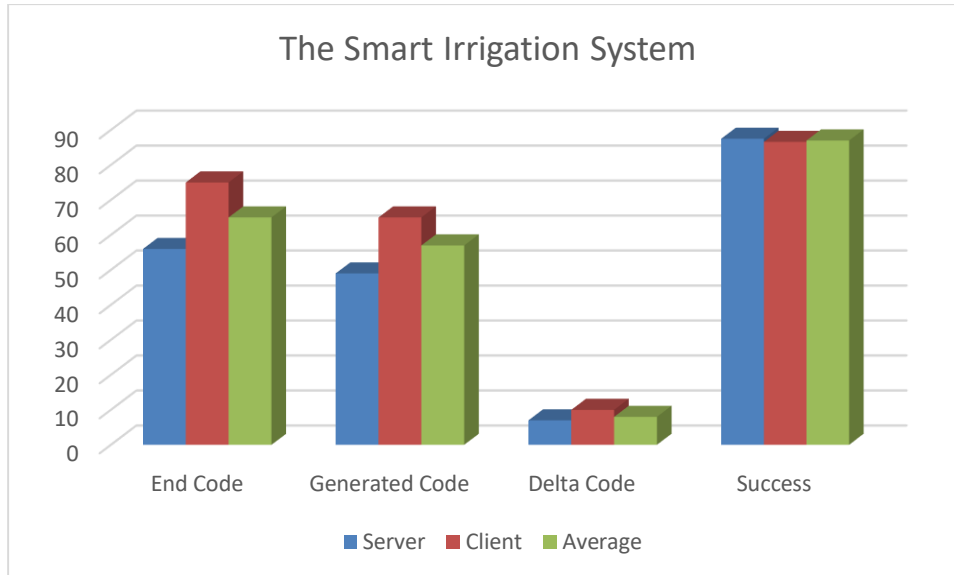


Figure 11.3 Bar chart of the smart irrigation system.

Figure 11.3 shows a comparison of the number of generated lines with the number of delta code lines for the smart irrigation system. More code is generated for the client-side compared to the server-side and the client-side required more delta codes. However, the success rate of the client-side and the server-side is so close. As a result, a high percentage of success rate was achieved with the number of codes generated by DSML4Riot, even if the client-side required more delta codes. In other words, the number of the generated codes overcame the number of the required delta codes.

The same result can be observed for the smart fire detection system in sub-section 11.4.1. Even if the client-side required more delta codes, the code generation success overcame delta codes. On the other hand the same result cannot be observed for the smart lighting system. Even if the number of generated code lines are much more than the delta codes, the success percentage is low when it is compared to the server-side and the actuator-side. Because, the client-side required more specific configurations such as, logical controls, manual IP address definition, and moving a generated code line to a necessary line.

A DSML that is designed for WSN and IoT based systems should always consider generating the network application codes, sensor sampling functions, and library inclusions for the designer. Because, these are the burdensome configurations to cause time-consumption, errors, and complexity. The specific configurations can be easily made when the generated code provides sufficiency to overcome the burdensome processes, it can leave the specific and basic configurations to the designer.

Figure 11.4 shows the end code, generated code, delta code, and success rate. Blue bars represent the server-side, red bars represent the client-side, and green bars represent the average of the two sides.

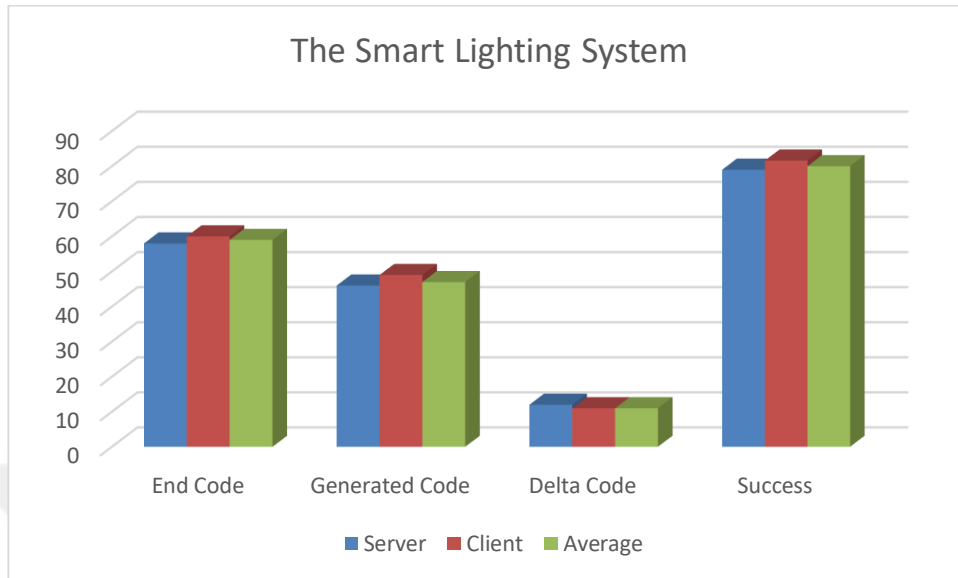


Figure 11.4 Bar chart of the smart lighting system.

The bars of the end code shows the code lines required to build the smart lighting system at the server-side and the client-side are close to each other. Since the server-side has less generated code lines and required more delta codes compared to the client-side the success rate of the server-side is low according to the client-side. The generated code lines of the client-side are more than the server-side and it required fewer delta codes in this case study. The same result could not be observed in the smart irrigation system. This can be concluded as the requirements of a case study changes according to its target domain and given functionalities. Therefore, the roles of the server-side and client-side (or a specific role e.g actuator-side) are defined according to the behavior of the target system.

As a result, the common requirements of the target domains to create a system should be met by the DSML and the specific and basic configurations should be left to the designer. To achieve this the DSML should focus on the requirements that create the backbone of the IoT systems. In this thesis study, this focus is mostly made on, communication protocol, threads, and GPIO during the development of the DSML4Riot.

So far, the evaluation of the case studies is made individually and comparisons with the discussions of each other are given. In the following sub-section 11.4.3, the average success rates of the case studies are made and the success rate of the DSMLs is given.

11.4.3. Average of Evaluation Results of DSMLs' Code Generation Success

In this sub-section average of evaluation results of DSMLs' code generation success were illustrated and discussed. These averages show the success of the DSMLs on both case studies and it also shows the success of the DSMLs. The evaluation was made according to numerical results that were given in sub-section 11.4.1 and 11.4.2.

Firstly, the case studies the average success rate that is given in Figures 11.1, 11.2, 11.3, 11.4 are considered. Secondly, the averages of these success rates are also shown in Figures 11.5 and 11.6. Lastly, the averages of these success rates' were averaged to find the success rate of the DSMLs.

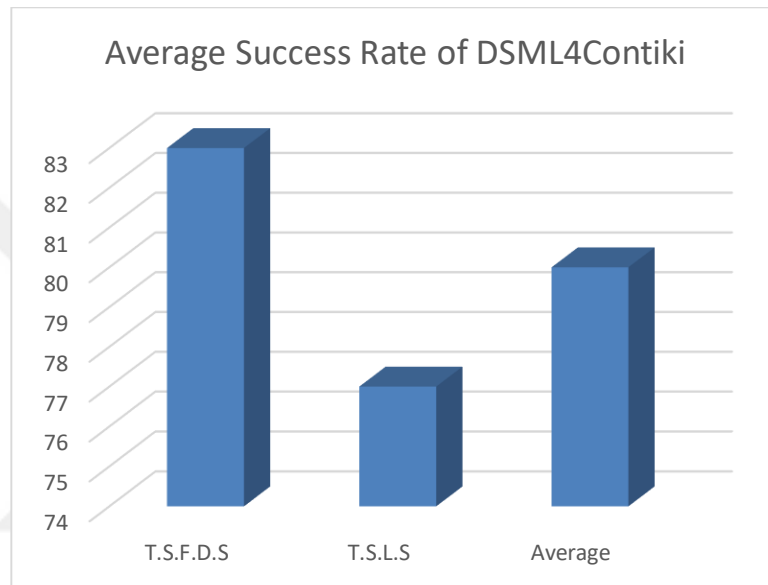


Figure 11.5 Bar chart of average success rate of the DSML4Contiki.

Figure 11.5 shows the code generation success rate on the smart fire detection system and the smart lighting system are %83 and %77, respectively. Therefore the success rate of the DSML4Contiki is %80.

In the awareness of the evaluation with a single case study may not achieve a comprehensive evaluation, the evaluation of the DSML4Contiki is made with two case studies. The number of the generated code lines, the number of delta code lines, and the number of end code lines are given. Using the ratio of numbers of generated code lines to the numbers of end code lines the success rate of the code generation is calculated for both case studies. The average of success rates of each case study's each side (server, client, also actuator for the smart lighting system) is calculated. This results in average code generation success on a case study.

To calculate the success rate of the DSML4Contiki, the average success rates on the smart library fire detection system and the smart lighting system are averaged.

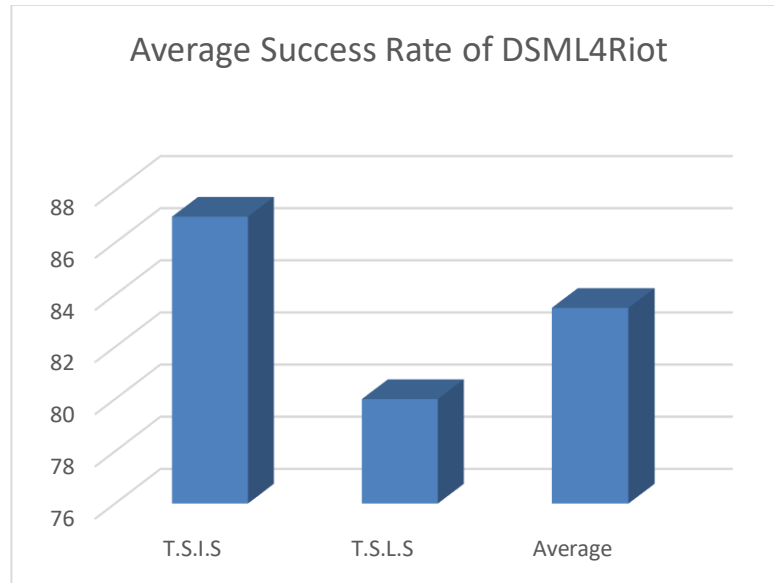


Figure 11.6 Bar chart of average success rate of the DSML4Riot.

Figure 11.5 shows the code generation success rate on the smart irrigation system and the smart lighting system are %87 and %80, respectively. Therefore the success rate of the DSML4Riot is %83.5.

In the awareness of the evaluation with a single case study may not achieve a comprehensive evaluation, the evaluation of the DSML4Riot is made with two case studies. The number of the generated code lines, the number of delta code lines, and the number of end code lines are given. Using the ratio of numbers of generated code lines to the numbers of end code lines the success rate of the code generation is calculated for both case studies. The average of success rates of each case study's each side (server and client) are calculated. This results in average code generation success on a case study. To calculate the success rate of the DSML4Riot, the average success rates on the smart irrigation system and the smart lighting system are averaged.

As mentioned in Section 1, the smart lighting system was implemented as a common case study. Since this case study is implemented both using DSML4Riot and DSML4Contiki, code generation capabilities of DSMLs are also evaluated using a common case study too. As figure 11.5 shows the success rate of the DSML4Contiki on the smart lighting case study is %77 and as figure 11.6 shows the success rate of the DSML4Riot on the smart lighting case study is %80. As can be seen, the results are close to each other. Surely, the results could be more close to each other. However, a different architecture was wanted to be used by adding an actuator node on the Contiki side. In the Riot side, traditional server and client architecture is used. In this way, also the success of the code generation is showed when the designer decides to use a node that has another role in the design (different than a server or a client). Moreover, this also shows the DSMLs' consistent success and code generation capability by supporting multiple node designs. Further, this advances the idea of when the common requirements that are the backbones of the WSN and IoT based systems are satisfied, a high success rate can be reached. Next, specific and basic configurations can be left to the designer by adding delta codes.

11.5. Code Generation Success Rates with PIM-to-PSM Success Rates

In this sub-section, the PIM-to-PSM success rates with the DSMLs' code generation success rates are given and discussed. The designer may use only the Platform-Specific Model level to generate codes. However, since this highly requires technical knowledge and consumes time, an upper-level abstraction helps the designer to create designs independently from the platforms. In the end, a more experienced designer has to extend the generated PSMs in the Platform-Specific level to have a full coverage model of the target system. Next, the generated codes can be completed to have a fully functional system by a software designer.

Figure 11.7 shows code generation success and model transformation success for DSML4Contiki. It should be noted that the code generation successes belong to the complete PSMs of the case studies.

- Blue and red bars of the smart fire detection system can be interpreted as to realize this case study the %63.8 of the PSM could be generated using the PIM and %83 of the full code of the system could be generated using DSML4Contiki.
- Blue and red bars of the smart lighting system can be interpreted as to realize this case study the %69.4 of the PSM could be generated using the PIM and %77 of the full code of the system could be generated using DSML4Contiki.
- On average, the success rate of the PIM transformation is %66 with %80 code generation success of the DSML4Contiki.

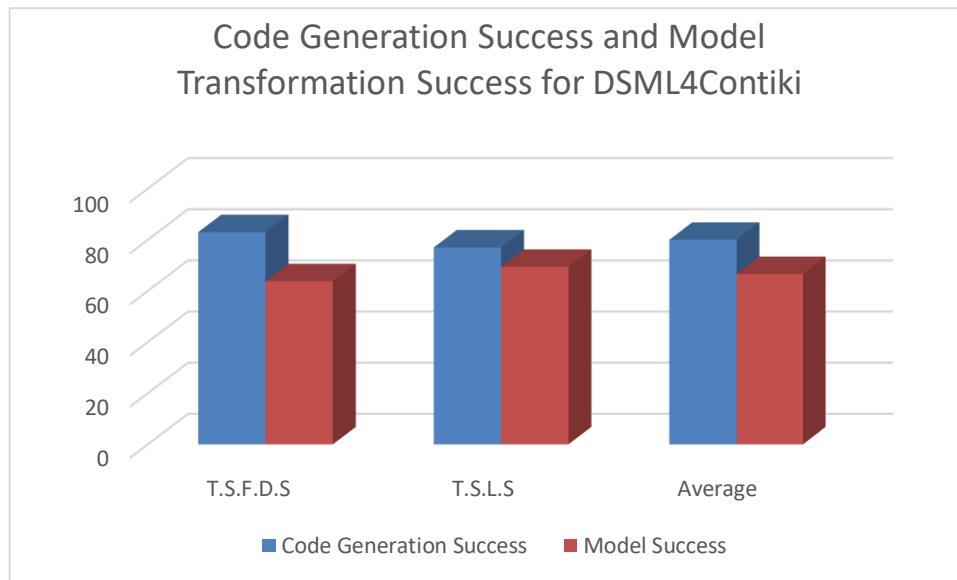


Figure 11.7 Code generation success and model transformation success for DSML4Contiki.

Figure 11.8 shows code generation success and model transformation success for DSML4Riot. It should be noted that the code generation successes belong to the complete PSMs of the case studies.

- Blue and red bars of the smart irrigation system can be interpreted as to realize this case study the %50 of the PSM could be generated using the PIM and %87 of the full code of the system could be generated using DSML4Riot.
- Blue and red bars of the smart lighting system can be interpreted as to realize this case study the %58.8 of the PSM could be generated using the PIM and %80 of the full code of the system could be generated using DSML4Riot.
- On average, the success rate of the PIM transformation is %54.4 with %83.5 code generation success of the DSML4Riot.

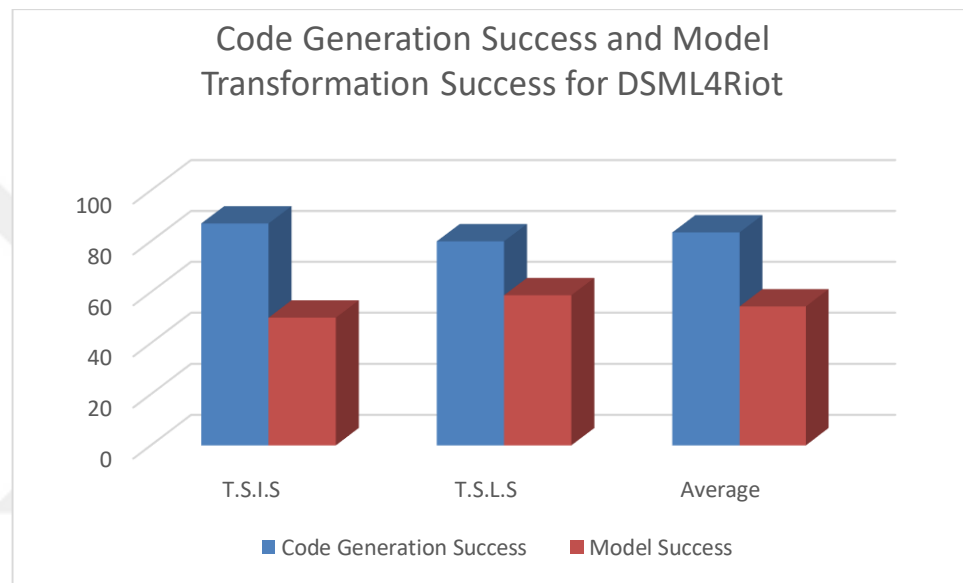


Figure 11.8 Code generation success and model transformation success for DSML4Riot.

As figure 11.7 and figure 11.8 shows the model transformation and code generation successes, more model elements could be generated for the DSML4Contiki. The average model generation success for DSML4Contiki is %66.6 and the average model generation success for DSML4Riot is %54.4. Even if there is not a great difference, the results are not so close to each other. As mentioned in section 7, this difference is due to the variety of the model elements and design choices of the meta-models.

As a result, the evaluation of the study is concluded and the results are illustrated. The delta codes are given. The comparison between DSML4Contiki and DSML4Riot is made. Moreover, the comparison between case studies is also made. The end codes (final codes) can be found in Appendix B.

11.6. Challenges & Issues

Before implementing code generation rules, necessary libraries and features of the operating systems were tested and some problems were popped-up. Therefore, in this sub-section, the challenges and issues that were dealt with during the implementation of the DSMLs is mentioned. This sub-section contributes to the related researchers and students how to overcome these problems when they study on Contiki and Riot operating systems.

During the implementation of the DSML4Contiki and DSML4Riot operating systems, many problems are encountered. Most of the problems are based on bugs and incomplete libraries. Even if Contiki operating system is a mature operating system, it has some issues that may direct to false results its user. During the simulation of the protosocket library for TCP communication in CooJa, the protosocket codes did not work with the TmoteSky motes, alternatively, Z1 motes are used to test the code (Uwase et al., 2014). In the physical programming of the TmoteSky motes, protosocket codes run without any problem. Moreover, during the simulation process, sometimes the CooJa simulator fails to run any code and it terminates the simulation process. It should be noted that the simulation must be saved before starting the simulation in case of the CooJa closes itself. Furthermore, since TmoteSky is a source-constraint device, the EEPROM limit during the development of an application must be considered.

In the Riot operating system side, there were more critical problems since it is an under-development operating system. The first issue is about, reading a value from a sensor module using the SAUL interface. Because of this, some functions related to SAUL could not be used and alternative functions of the SAUL interface are used as a solution (RiotOS GitHub Issue, 2019a). This issue still in progress to be solved. Another issue was about the TCP library's compatibility for the ESP32 device (RiotOS GitHub Issue, 2019b). This issue is solved by the developers of the Riot operating system. The rest of the issues that are opened and solved for the contribution of this thesis study and the Riot operating system's users can be accessed with this reference (RiotOS GitHub Issue, 2019c). Since the Riot operating system does not have an official simulator, it was hard to implement the case studies and to test the application codes. Because, if any change had to be made in the code, then the code had to be flashed to the device again. This was a highly time-consuming process. In the next section, the conclusion and the future work of this thesis study are mentioned.

12. CONCLUSION & FUTURE WORK

In this thesis study, two domain-specific modeling languages and their supporting tools are developed to improve the design and implementation of IoT and WSN based systems. The complexity and difficulty in these systems are addressed. The proposed DSMLs provide a transformation to Petri-nets for design verification as well as a code generation mechanism to synthesize the executable codes for Contiki and Riot operating systems. Meta-models are designed for the Contiki operating system and Riot operating system. These meta-models are used as the abstract syntax of DSML4Contiki and DSML4Riot. A set of graphical notations, as well as some domain constraints, are used to advance graphical editors for the DSMLs. The developed DSMLs have an important role in providing code generation, fast development, and verification for the devices which are used in Contiki-based WSNs (based on the IEEE 802.15.4 protocol) and IoT boards such as the ESP32 transceiver module (based on the IEEE 802.11 protocol).

Since the proposed DSMLs automatically generates a Petri-net model, this assists in the analysis and verification of the design model in the early phases of the development, before any implementation. The automatic generation of the Petri-net models from the topology design model and applying different analyses on the topology design model using k-boundedness property of Petri-net models are provided a lot of beneficial results. Using this development approach, the designers can apply analyses such as network lifetime (first crashing nodes), bottleneck, and power consumption analyses in the early design phase (before the implementation and deployment) of the system development. This prevents the extra iterations of re-implementation and re-deployment for the system instead, the system is re-designed and re-generated at the early phase of development considering the results of the analysis. Additionally, a bill of materials table is generated to inform the designer about the cost of the components. Propagation delays are calculated if the designer has a time-critic use case. In conclusion, the proposed approach not only can generate a high ratio of the codes from the system model but also can apply analyses based on the same domain-specific models. The Petri-net model is evaluated using the smart fire detection system.

The four case studies are implemented to show the capabilities of the DSML4Contiki and DSML4Riot. A common case study, the smart lighting system, is selected to show both the implementation and differences of both DSMLs. This difference is both caused by the operating system and the modeling approach of the DSMLs. During the development of the DSML4Contiki, the meta-model was aimed to have more EClasses. This gives the design flexibility to the designer but the designer has to use more design elements in the model. In the DSML4Riot side, the fewer elements usage was aimed by modeling some features of the operating system as attributes, additionally, this hardened the implementation of the code generation rules. Certainly, the operating systems itself have an effect on the design of a Domain-Specific Modeling Language. Since Contiki is an event-based operating system, it has more specialized features such as event-timers, events, and functions. These specialized features create variety in the DSML's elements.

The modeling approach is mentioned and the features that are not covered in the case studies are explained in the additional sections. It is aimed to explain how DSML4Contiki and DSML4Riot are implemented. The code generation rules are written according to given modeling approaches. This may also guide to a relevant developer, how a DSML can be implemented both graphically and syntactically.

Platform-Specific Models of the case studies are given in sub-section 5.5. These PSMs represent these case studies and case studies (target systems) conform to their PSMs. Using the Platform-Specific Models the target codes are generated for the target systems. The code generation success is evaluated in section 11. The smart lighting system is chosen as a common case study. The evaluation of the DSMLs is also made in a common case study. Furthermore, the DSMLs' description of a common target system as a model is also shown.

The modeling approach is explained in order to clarify how these two operating systems are modeled. The idea is intended to given in sub-sections 4.2.1, 4.2.2, 4.4, 4.5.2, 4.5.3, and 4.6. According to the modeling approach the DSML4Contiki and DSML4Riot are designed and code generation rules are implemented.

The Platform-Independent level is designed to create an environment where the designer can design a model without dealing with technical terms and technical operations. The main advantage of this method is to derive various alternative representations of the Platform-Specific level using the same PIM. In this way, the derivation process of PSMs can be partially automated. In this way, the designer can create many design viewpoints and can transform them to one or more Platform-Specific Models. It reduces the development time and implementation efforts. Moreover, models at all levels can be traced by the designer and stakeholders in the development process. During the transformation, the best configurations are made automatically for the designer to have a more consistent model that represents a target system. Therefore, the designer can focus on the more specific parts of the model. Certainly, the designer may decide to change these automatically set configurations. When the designer completes a PSM that is generated by a PIM, the code generation is made for the target system.

The PIM elements are described by meaningful icons. These icons are selected carefully both to guide the designer to simplify the design process and to provide a user-friendly environment. The models can be transformed to another model representations (e.g. Petri-net). Therefore the model at any level can be extended in another platform, in this way many analyses results can be achieved.

The transformation rules are implemented and given in sub-section 7.4. A transformation is an automated method that is used for modifying an existing model or creating new models. In this thesis study, transformation rules are used to transform a Platform-Independent model to one or many Platform-Specific Models. To achieve PIM-to-PSM

transformation, a set of rules is defined. Moreover, helper rules are written to provide more accurate results. These helper rules help to differentiate and to name some elements' attributes. The semantic constraints are given in section 8. These rules apply the domain rules in the modeling environment. The constraint checking languages are more expressive than the simple cardinality property checking rules implemented in the meta-model and they can check those properties as well. However, the constraint checking rules are generally used to implement the rules that are not possible to implement in the meta-model, such as the rules checking a property between the meta-elements that are not directly linked together. In other words, these rules are complementary to the meta-model cardinality rules. Therefore to provide the sustainability of the target system in the after-deployment phase, these rules must validate the designed model.

To represent the overview of the proposed DSMLs FTG+PM diagram's Process Model is given. The PM specifies an MDE process using these formalisms and transformations as well as using the instances of those formalisms. The Process Model is explained and the design steps are shown in section 9. Using the Process Model an inexperienced designer can be navigated starting from the system-level design to code generation and verification of the topology model using Petri-net as well as including analyses results.

The thesis study is evaluated comprehensively in section 11. Evaluation of modeling languages is done by assessing its code generation and PIM-to-PSM transformation capabilities. This was done by comparing the lines of generated code and final code (end code) and the number of PIM elements and the number of PSM elements. Moreover, to have a more accurate result, numbers of attributes that are not modified by the designer is evaluated as a successful transformation. A focus on both the code generation and model generation performance is made.

On the DSML4Contiki side, %63.8 of the Platform-Specific Model of the smart fire detection system could be generated. When the Platform-Specific Model of the smart fire detection system is completed, %83 of the final code of the case study could be generated. %69.4 of the Platform-Specific Model of the smart lighting system could be generated. When the Platform-Specific Model of the smart lighting system is completed, the %77 of the final code of the case study could be generated. According to these results, the DSML4Contiki has a %66 success rate for the PIM-to-PSM transformation and has a %80 success rate for code generation.

On the DSML4Riot side, %50 of the Platform-Specific Model of the smart irrigation system could be generated. When the Platform-Specific Model of the smart irrigation system is completed, %87 of the final code of the case study could be generated. %58.8 of the Platform-Specific Model of the smart lighting system could be generated. When the Platform-Specific Model of the smart lighting system is completed, %80 of the final code of the case study could be generated. According to these results, the DSML4Riot has a %54.4 success rate for the PIM-to-PSM transformation and has a %83.5 success rate for the code generation.

In conclusion, this thesis study shows the provided DSMLs, can significantly reduce the development complexity, time and cost as well as increase the quality of the resulting system. Moreover, the proposed approach not only can generate a high ratio of the codes from the system model but also can apply analyses based on the same domain-specific models. Also, this study discusses the workflow of the proposed development methodology using a process model in the FTG+PM framework to increase the reusability of the framework. The study is evaluated using four case studies. Using the smart fire detection system various analyses are achieved using Petri-nets and the results are demonstrated. Future work will extend this study to support the modeling of other operating systems and will provide more functionalities in the Platform-Independent Level. As another future work, it is aimed to use Multi-agent Systems in the modeling, analysis, and implementation of WSN and IoT based systems and to use Digital Twin methodology for the after-deployment phase of these systems.



REFERENCES

- Adda, M., & Saad, R.** (2014). A data sharing strategy and a DSL for service discovery, selection and consumption for the IoT. *Procedia Computer Science*, 37, 92-100.
- Asici, T. Z., Karaduman, B., Eslampanah, R., Challenger, M., Denil, J., & Vangheluwe, H.** Applying model driven engineering techniques to the development of contiki-based IoT systems. In *Proceedings of the 1st International Workshop on Software Engineering Research & Practices for the Internet of Things* (pp. 25-32), (2019, May). IEEE Press.
- Bagherzadeh, M., Bordeleau, F., Bruel, J. M., Dingel, J., Gérard, S., Hili, N., & Voss, S.** Summary of Workshop on Model-Driven Engineering Tools (MDETools' 17).
- Beaulaton, D., Said, N. B., Cristescu, I., Fleurquin, R., Legay, A., Quilbeuf, J., & Sadou, S.** (2018, June). A language for analyzing security of IoT systems. In *2018 13th Annual Conference on System of Systems Engineering (SoSE)* (pp. 37-44). IEEE.
- Bézivin, J.** (2005). On the unification power of models. *Software & Systems Modeling*, 4(2), 171-188.
- Bézivin, J., Jouault, F., & Valduriez, P.** (2004, October). On the need for megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.
- Bonet, P., Lladó, C. M., Puijaner, R., & Knottenbelt, W. J.** (2007, October). PIPE v2. 5: A Petri net tool for performance modelling. In *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*.
- Bruneliere, H., Burger, E., Cabot, J., & Wimmer, M.** (2019). A feature-based survey of model view approaches. *Software & Systems Modeling*, 18(3), 1931-1952.
- Challenger, M., Kardas, G., & Tekinerdogan, B.** (2016). A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. *Software Quality Journal*, 24(3), 755-795.
- ContikiOS GitHub.** <https://github.com/contiki-os/contiki/blob/master/doc>. (2006) (Acces Date: 15 November 2019)

REFERENCES Cont'd

- Dantas, P., Rodrigues, T., Batista, T., Delicato, F. C., Pires, P. F., Li, W., & Zomaya, A. Y.** (2013, May). LWiSSy: A domain specific language to model wireless sensor and actuators network systems. In 2013 4th International Workshop on Software Engineering for Sensor Network Applications (SESENA) (pp. 7-12). IEEE.
- De Lara, J., & Guerra, E.** (2010, October). Generic meta-modelling with concepts, templates and mixin layers. In International Conference on Model Driven Engineering Languages and Systems (pp. 16-30). Springer, Berlin, Heidelberg.
- De Lara, J., Guerra, E., Kienzle, J., & Hattab, Y.** (2018, October). Facet-oriented modelling: open objects for model-driven engineering. In Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (pp. 147-159). ACM.
- Doddapaneni, K., Ever, E., Gemikonakli, O., Malavolta, I., Mostarda, L., & Muccini, H.** (2012, June). A model-driven engineering framework for architecting and analysing wireless sensor networks. In Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications (pp. 1-7). IEEE Press.
- Doddapaneni, K., Ever, E., Gemikonakli, O., Malavolta, I., Mostarda, L., & Muccini, H.** (2012, June). A model-driven engineering framework for architecting and analysing wireless sensor networks. In Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications (pp. 1-7). IEEE Press.
- Dunkels, A., Gronvall, B., & Voigt, T.** (2004, November). Contiki-a lightweight and flexible operating system for tiny networked sensors. In 29th annual IEEE international conference on local computer networks (pp. 455-462). IEEE.
- Egesoy, A.,** (2017). Model Driven Software Engineering as the Next Paradigm. International Journal on Advanced Research in Computer Science and Electronics Engineering, vol.6, 104-116.
- Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, MatthiasWahlisch, and Thomas C Schmidt.** 2013. RIOT OS: Towards an OS for the Internet of Things. In 2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS). IEEE, 79–80.
- Espressif Systems. ESP32-WROOM-32 Datasheet.** (2019).
https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf. (Access Date: 21 November 2019)

REFERENCES Cont'd

- Essaadi, F., Maissa, Y. B., & Dahchour, M.** (2016, May). MDE-based languages for wireless sensor networks modeling: a systematic mapping study. In *International Symposium on Ubiquitous Networking* (pp. 331-346). Springer, Singapore.
- Kühlwein, A., Paule, A., Hielscher, L., Rosenstiel, W., & Bringmann, O.** (2019, September). Firmware Synthesis for Ultra-Thin IoT Devices Based on Model Integration. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)* (pp. 339-346). IEEE.
- Kardas, G., Demirezen, Z., & Challenger, M.** (2010, June). Towards a DSML for semantic web enabled multi-agent systems. In *Proceedings of the International Workshop on Formalization of Modeling Languages* (p. 5). ACM.
- Golra, F. R., Beugnard, A., Dagnat, F., Guerin, S., & Guychard, C.** (2016, March). Addressing modularity for heterogeneous multi-model systems using model federation. In *Companion Proceedings of the 15th International Conference on Modularity* (pp. 206-211). ACM.
- Harrand, N., Fleurey, F., Morin, B., & Husa, K. E.** (2016, October). 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* (pp. 125-135). ACM.
- Hussein, M., Li, S., & Radermacher, A.** (2017, September). Model-driven Development of Adaptive IoT Systems. In *MODELS (Satellite Events)* (pp. 17-23).
- Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I.** (2008). ATL: A model transformation tool. *Science of computer programming*, 72(1-2), 31-39.
- Karaduman, B., Aşıcı, T., Challenger, M., & Eslampanah, R.** (2018, June). A Cloud and Contiki based Fire Detection System using Multi-Hop Wireless Sensor Networks. In *Proceedings of the Fourth International Conference on Engineering & MIS 2018* (p. 66). ACM.
- Karaduman, B., Challenger, M., & Eslampanah, R.** (2018, May). ContikiOS based library fire detection system. In *2018 5th International Conference on Electrical and Electronic Engineering (ICEEE)* (pp. 247-251). IEEE.

REFERENCES Cont'd

- Karimpour N., Karaduman B., Ural A., Challenger M., Dagdeviren O.**, IoT based Hand Hygiene Compliance Monitoring, International Symposium on Networks, Computers and Communications (ISNCC'19), Istanbul-Turkey, Mar 19, 2019 publication description
- Kim, S., Cho, J., & Park, D.** (2018). Accelerated DEVS Simulation Using Collaborative Computation on Multi-Cores and GPUs for Fire-Spreading IoT Sensing Applications. *Applied Sciences*, 8(9), 1466.
- Lekidis, A., Stachtari, E., Katsaros, P., Bozga, M., & Georgiadis, C. K.** (2018). Model-based design of IoT systems with the BIP component framework. *Software: Practice and Experience*, 48(6), 1167-1194.
- Liu, S., Zeng, R., & He, X.** (2011). PIPE+-A Modeling Tool for High Level Petri Nets. In *SEKE* (pp. 115-121).
- LoLA.** (2009). <https://www2.informatik.hu-berlin.de/top/lola/loladoku/node4.html>. (Access Date: 26 October 2019).
- Lucio, L., Mustafiz, S., Denil, J., Meyers, B., & Vangheluwe, H.** (2012). The formalism transformation graph as a guide to model driven engineering. School of Computer Science, McGill University, Tech. Rep. SOCS-TR2012, 1.
- Malavolta, I., & Muccini, H.** (2014, August). A study on MDE approaches for engineering wireless sensor networks. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications* (pp. 149-157). IEEE.
- Mülders, P., Gruner, S., & Thang, N. X.** (2011, May). Model-driven design plus artificial intelligence for wireless sensor networks software development. In *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications* (pp. 63-64). ACM.
- Mulligan, G.** (2007, June). The 6LoWPAN architecture. In *Proceedings of the 4th workshop on Embedded networked sensors* (pp. 78-82). ACM.
- Mustafiz, S., Denil, J., Lúcio, L., & Vangheluwe, H.** (2012, October). The FTG+ PM framework for multi-paradigm modelling: An automotive case study. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling* (pp. 13-18). ACM.

REFERENCES Cont'd

- Nakahori, K., & Yamaguchi, S.** (2017, January). A support tool to design IoT services with NuSMV. In 2017 IEEE International Conference on Consumer Electronics (ICCE) (pp. 80-83). IEEE.
- Negash, B., Westerlund, T., Rahmani, A. M., Liljeberg, P., & Tenhunen, H.** (2017). DoS-IL: A domain specific Internet of Things language for resource constrained devices. *Procedia Computer Science*, 109, 416-423.
- Object Management Group.** OMG Unified Modeling Language Version 2.5.1 Technical Report, OMG. (December 2017). <https://www.omg.org/spec/UML/2.5.1/PDF>. (Access Date: 7 December 2019)
- Österlind, F., Dunkels, A., Eriksson, J., Finne, N., & Voigt, T.** (2006). Cross-level sensor network simulation with cooja. In First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006).
- Peterson, J. L.** (1981). Petri net theory and the modeling of systems. Prentice-hall.
- Petri, C. A., & Reisig, W.** (2008). Petri net. *Scholarpedia*, 3(4), 6477.
- RiotOS Docs.** (2019). https://riot-os.org/api/group_net_sock.html. (Access Date: 11 November 2019).
- RiotOS GitHub Issue.** <https://github.com/RIOT-OS/RIOT/issues/12057>. (2019) (Access Date: 12 December 2019).
- RiotOS GitHub Issue.** <https://github.com/RIOT-OS/RIOT/issues/11910>. (2019) (Access Date: 12 December 2019).
- RiotOS GitHub Issue.** <https://github.com/RIOT-OS/RIOT/issues?utf8=%E2%9C%93&q=is%3Aclosed+is%3Aissue+author%3Ammaxus35+>. (2019) (Access Date: 12 December 2019).
- Rodrigues, T., Delicato, F. C., Batista, T., Pires, P. F., & Pirmez, L.** (2017). An approach based on the domain perspective to develop WSN applications. *Software & Systems Modeling*, 16(4), 949-977.

REFERENCES Cont'd

- Salay, R., Chechik, M., Easterbrook, S., Diskin, Z., McCormick, P., Nejati, S., Viriyakattiyaporn, P.** (2007, October). An Eclipse-based tool framework for software model management. In Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange (pp. 55-59). ACM.
- Schmidt, D. C.** (2006). Model-driven engineering. COMPUTER-IEEE COMPUTER SOCIETY-, 39(2), 25.
- Shelby, Z., Hartke, K., Bormann, C., & Frank, B.** (2014). RFC 7252: The constrained application protocol (CoAP). Internet Engineering Task Force.
- Tei, K., Shimizu, R., Fukazawa, Y., & Honiden, S.** (2014). Model-driven-development-based stepwise software development process for wireless sensor networks. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 45(4), 675-687.
- Thramboulidis, K., Bochalos, P., & Bouloumpasis, J.** (2017, October). A framework for MDE of IoT-based manufacturing cyber-physical systems. In Proceedings of the seventh international conference on the internet of things (p. 11). ACM.
- TmoteSky** **Datasheet.**
https://www.advanticsys.com/shop/documents/1322650507_MSP430F1611.zip. (2019)
 (Access Date: 26 November 2019)
- Tomasic, I., Khosraviani, K., Rosengren, P., Jörntén-Karlsson, M., & Lindén, M.** (2018, May). Enabling IoT based monitoring of patients' environmental parameters: Experiences from using OpenMote with OpenWSN and Contiki-NG. In 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) (pp. 0330-0334). IEEE.
- Uwase, M. P., Long, N. T., Tiberghien, J., Steenhaut, K., & Dricot, J. M.** (2014). Outdoors range measurements with Zolertia Z1 Motes and Contiki. In Real-world wireless sensor networks (pp. 79-83). Springer, Cham.
- Vangheluwe, H., Amaral, V., Giese, H., Broenink, J., Schätz, B., Norta, A., & Vallecillo, A.** (2016). MPM4CPS: multi-paradigm modelling for Cyber-Physical Systems. In CEUR workshop proceedings (Vol. 1675, pp. 40-47).

REFERENCES Cont'd

- Wagenknecht, G., Anwander, M., & Braun, T.** (2009, May). Hop-to-hop reliability in IP-based wireless sensor networks-A cross-layer approach. In International Conference on Wired/Wireless Internet Communications (pp. 61-72). Springer, Berlin, Heidelberg.
- Wimmer, M., & Langer, P.** (2013). A benchmark for model matching systems: The heterogeneous metamodel case. *Softwaretechnik-Trends*: Vol. 33, No. 2.
- Winter, T.** (2011). RPL: IPv6 routing protocol for low power and lossy networks. RFC6550.



ACKNOWLEDGMENTS

I would like to thank Asst. Prof. Dr. Moharram Challenger and Assoc. Prof. Dr. Orhan Dağdeviren for their supervision, support, and motivation during my academic studies. Further, I present my gratitude to Prof.Dr. Hans Vangheluwe, Asst. Prof. Dr. Joachim Denil and Dr. Raheleh Eslampanah. I have completed this thesis study with their guidance and mentoring.

Furthermore, I would like to thank my friends, Tansu Zafer Aşıcı, Ömer Faruk Alaca, Ercüment Türk, and Sadık Arslan that I worked together in the Scientific Research Project. Additionally, I would like to thank my friends, Mustafa Abshir Mohamed, Hussein Masoud Marah, and Malath Thaer Nejres that I worked together in the Ege-SERLab.

Lastly, I would like to thank Scientific Research Projects Directorate of Ege University for funding our Scientific Research Project named “A Study on the Model-driven Development of Internet of Things Software based on Contiki Operating System” under grant 17-UBE-002.



Burak KARADUMAN

CURRICULUM VITAE

Name & Surname: Burak KARADUMAN

E-mail: bburakkaraduman@gmail.com

LinkedIn Profile: <https://www.linkedin.com/in/burak-karaduman-7382bb140/>

ResearchGate Profile: https://www.researchgate.net/profile/Burak_Karaduman2

Personal Statement: Electrical&Electronics Engineer

Work Experience:

1. 01/01/2017–Present Teaching Assistant in Software Engineering

Given courses:

- Object-Oriented Programming (Java)
- Software System Analysis
- Software Construction
- Logic Design
- Introduction to Programming (Python)
- Introduction to Programming (C)

2. 08/08/2017–08/08/2019 Researcher in Scientific Research Project in Ege University

A Study on the Model-driven Development of Internet of Things Software based on ContikiOS

The project is completed with the supervision of following researchers:

- Asst. Prof. Dr. Moharram CHALLENGER
- Assoc. Prof. Dr. Geylani KARDAS
- Assoc. Prof. Dr. Orhan DAGDEVIREN
- Dr. Raheleh ESLAMPANAH

3. 08/06/2016–08/06/2017 Scientific Research Project in Yaşar University
 - The project is completed with the supervision of Dr. Mahir Kutay.
 - Development of Electronic Guitar for Disabled and Elderly People.
 - The project is made using Python, RaspberryPI and Arduino Uno.

Publications:

1. “Applying Model Driven Engineering Techniques to the Development of Contiki-based IoT Systems”, 1st International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT 2019) Mar 27, 2019 , Asici, T. Z., **Karaduman B.**, Eslampanah R.,Challenger M.,Denil J., Vangheluwe H.
2. "IoT based Hand Hygiene Compliance Monitoring", International Symposium on Networks, Computers and Communications (ISNCC'19), Istanbul-Turkey ,Mar 19, 2019 publication description Karimpour N., **Karaduman B.**, Ural A., Challenger M., Dagdeviren O.
3. “A Cloud and Contiki based Fire Detection System using Multi-Hop Wireless Sensor Networks”, Jun 19, 2018, International Conference on Engineering & MIS (ICEMIS 2018), 2018,International Conference on Engineering & MIS (ICEMIS 2018), **Karaduman B.**, Asici T.Z., Challenger M., Eslampanah R.
4. "ContikiOS based library fire detection system", Mar 1, 2018, 5th International Conference on Electrical and Electronic Engineering (ICEEE), **Karaduman, B.**, Challenger, M.,& Eslampanah,R.

APPENDIXES

Appendix A: Figures that belong the first phases and previous versions of the case studies.

Appendix B: The codes that are used for the implementation of the case studies using DSML4Contiki and DSML4Riot as well as including code generation rules.



Appendix A

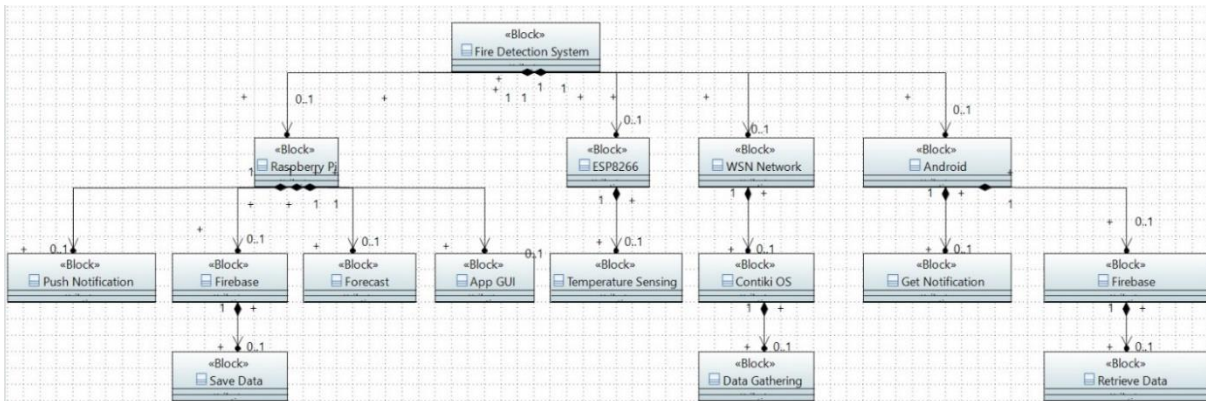


Figure A.1 Block Diagram of the fire detection system (Karaduman et al., 2018a; 2018b).

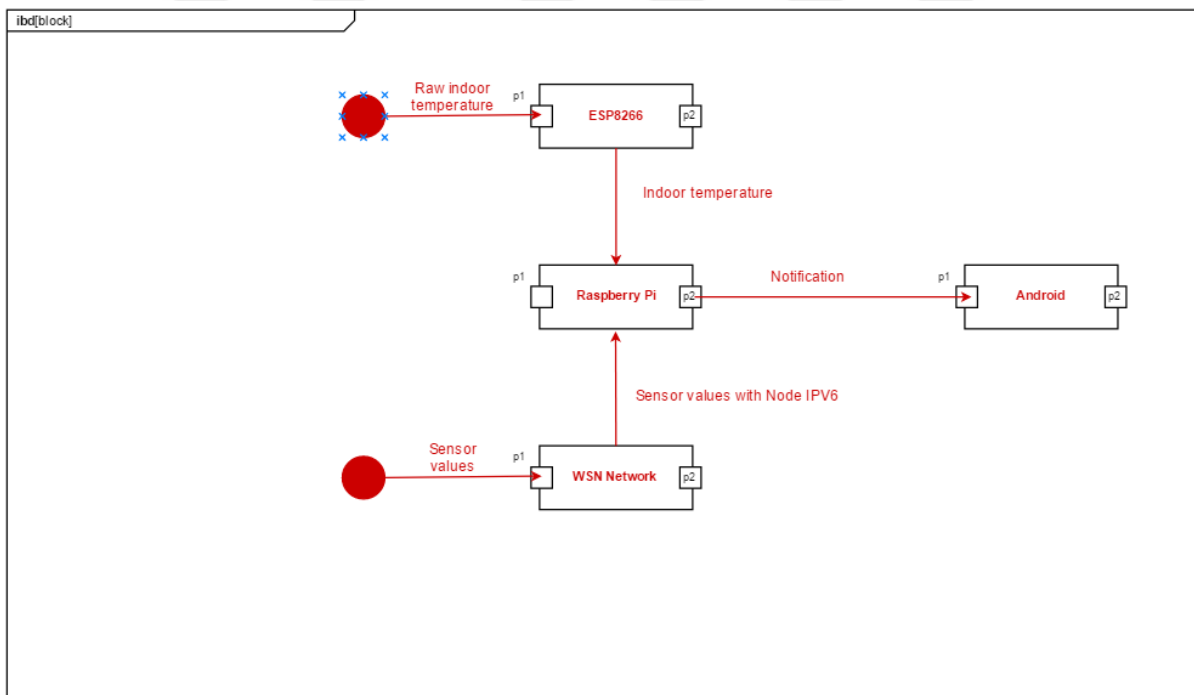


Figure A.2 Internal Block Diagram of the fire detection system (Karaduman et al., 2018a; 2018b)



Figure A.3 First configurations of the study (Karaduman et al., 2018a; 2018b).

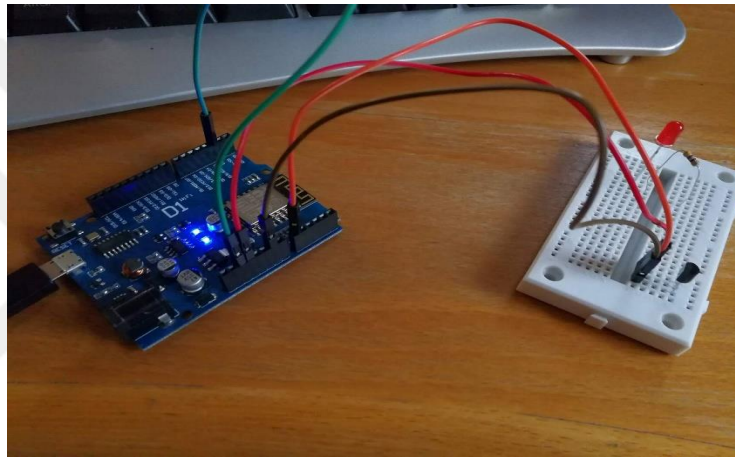


Figure A.4 ESP development board (Karaduman et al., 2018a; 2018b).

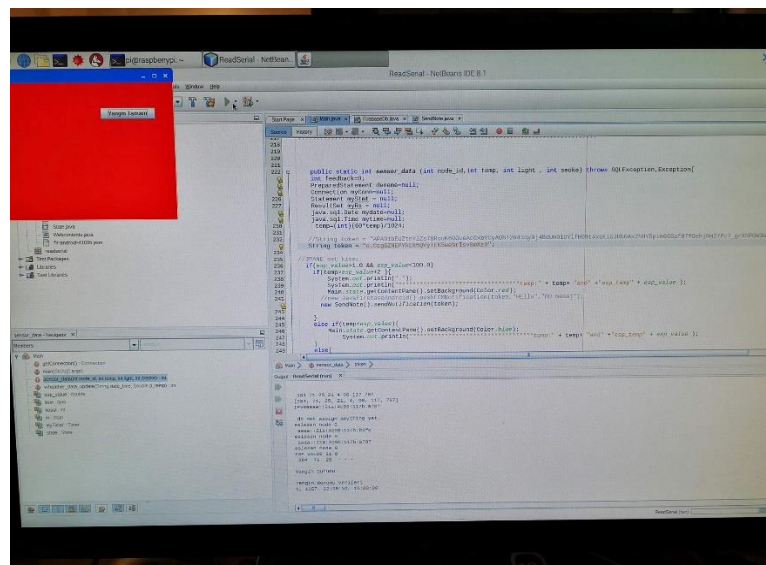


Figure A.5 Filtering source nodes' sensor data and IP addresses (Karaduman et al., 2018a; 2018b).



Figure A.6 RaspberryPI as a gateway (Karaduman et al., 2018a, 2018b)

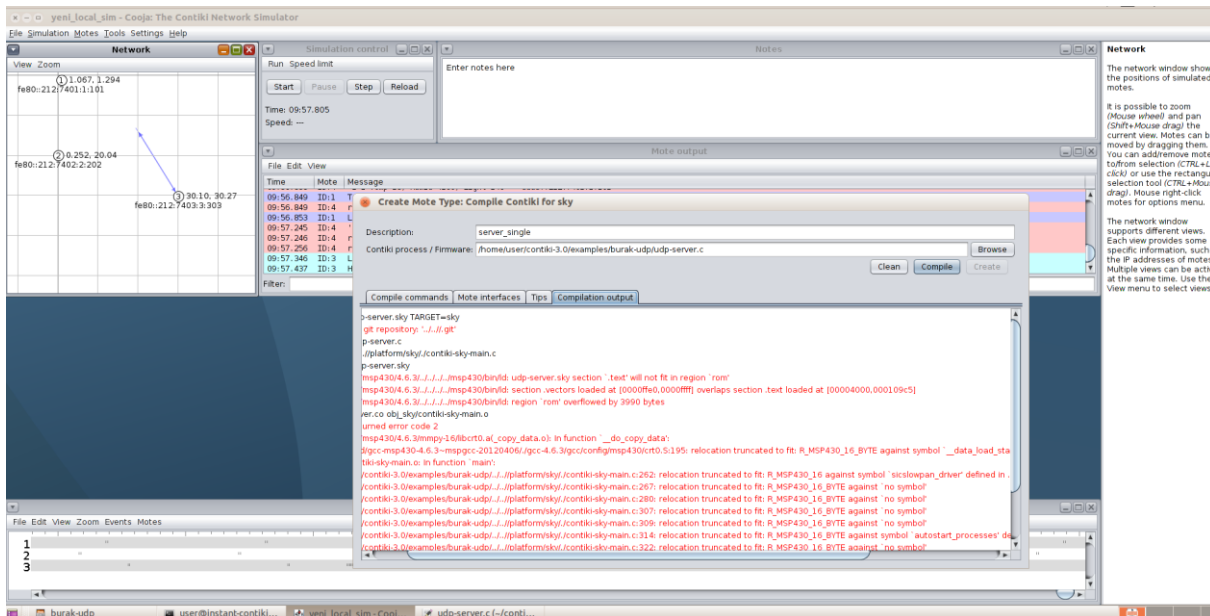


Figure A.7 Code size and EEPROM limitations of TmoteSky mote.

Appendix B

```

#include "contiki.h"
#include "contiki-lib.h"
#include "contiki-net.h"
#include <string.h>
#define DEBUG DEBUG_PRINT
#include "net/ip/uiip.h"
#include "net/ip/uiip-debug.h"
#define UIP_IP_BUF ((struct uip_ip_hdr *)&uip_buf[UIP_LLH_LEN])
/* Server */
PROCESS(MY_Server_Process,"Server Process");
AUTOSTART_PROCESSES(&MY_Server_Process);
static struct uip_udp_conn *Server_Connection;
uip_ipaddr_t ripaddr;
static void TCPIP_Event_handler(void){
char buf[50];
if(uip_newdata())
{
((char *)uip_appdata)[uip_datalen()] = 0;
PRINTF("Server received: '%s' from ", (char *)uip_appdata);
PRINT6ADDR(&UIP_IP_BUF->srcipaddr);
PRINTF("\n");
PRINTF("DATA sending reply\n");
uip_ipaddr_copy(&Server_Connection->ripaddr, &UIP_IP_BUF->srcipaddr);
uip_udp_packet_send(Server_Connection, "asss", sizeof("asss"));
uip_create_unspecified(&Server_Connection->ripaddr);
memset(&Server_Connection->ripaddr, 0, sizeof(Server_Connection->ripaddr));
}
}
PROCESS_THREAD(MY_Server_Process, ev, data)
{
// NOTE: Add ServerConnection to model to create TCP(w/o psock) and UDP Connection
PROCESS_BEGIN();
PRINTF("UDP server started\n");
Server_Connection = udp_new(NULL, UIP_HTONS(5555), NULL);
if(Server_Connection == NULL)
{
PRINTF("No UDP connection available, exiting the process!\n");
}
udp_bind(Server_Connection, UIP_HTONS(5556));
PRINTF("Created a server connection with remote address ");
PRINT6ADDR(&Server_Connection->ripaddr);
PRINTF(" local/remote port %u/%u\n", UIP_HTONS(Server_Connection->lport),
UIP_HTONS(Server_Connection->rport));
while(1)
{
PROCESS_YIELD();
if(ev==tcpip_event)
{
TCPIP_Event_handler();
}
}
PROCESS_END();
}

```

Listing B.1 Final server code for both case studies.

```

#include "contiki.h"
#include "contiki-lib.h"
#include "contiki-net.h"
#include "net/ip/uiplib.h"
#include "net/ipv6/uiplib-ds6.h"
#include "net/ip/uiplib-udp-packet.h"
#include <stdio.h>
#include <string.h>
#define DEBUG DEBUG_PRINT
#include "net/ip/uiplib-debug.h"
#include "dev/light-sensor.h"
#include "dev/sht11/sht11-sensor.h"
static uip_ipaddr_t server_ipaddr, static int i, static uint8_t state, static int value[4];
PROCESS(Measure_Temperature,"Measure Temperature");
PROCESS(Broadcast_Process,"Broadcast Process");
AUTOSTART_PROCESSES(&Measure_Temperature,&Broadcast_Process);
static struct uip_udp_conn *Client_Connection, static char buf [50];
static void Timeout_Event_handler(void) {
static int seq_id;
printf("Client sending to: ");
sprintf(buf,"Temp %d, Humid %d, Light %d seq_id %d ",value[0],value[1],value[2],++seq_id);
printf(" (msg: %s)\n", buf);
uip_udp_packet_sendto(Client_Connection, buf, strlen(buf),&server_ipaddr, UIP_HTONS (3001));
static void tcpip_event_handler(void){
char *str;
if(uip_newdata()) {
str = uip_appdata;
str [uip_datalen()] = '\0';
printf("DATA recv \"%s\n", str);
PRINT6ADDR(&Client_Connection->ripaddr); } } //end of tcpip_handler
PROCESS_THREAD(Measure_Temperature,ev,data){
PROCESS_BEGIN();
static struct etimer Etimer_Temperature;
etimer_set(&Etimer_Temperature,3*CLOCK_SECOND);
while(1){
PROCESS_WAIT_EVENT();
SENSORS_ACTIVATE(sht11_sensor);
value[0]=(((sht11_sensor.value(SHT11_SENSOR_TEMP)/10)-396)/10)-6;
printf("Temp: %d\n", value[0]);
SENSORS_DEACTIVATE(sht11_sensor); } // end of while loop
PROCESS_END(); } // end of process.
PROCESS_THREAD(Broadcast_Process,ev,data){
PROCESS_BEGIN();
uip_ipaddr_t ipaddr;
uip_ip6addr(&ipaddr,0xfe80, 0, 0, 0, 0xc30c,0x0000,0x0000,0x0002); //write server address here.
printf("connection server address is %s\n");
PRINT6ADDR(&ipaddr);
Client_Connection=udp_new(&ipaddr,UIP_HTONS(3001),NULL);
udp_bind(Client_Connection,UIP_HTONS(3000));
PRINTF("Created a connection with the server ");
PRINT6ADDR(&Client_Connection->ripaddr);
PRINTF(" local/remote port %u%u\n", UIP_HTONS(Client_Connection->lport), UIP_HTONS(Client_Connection->rport));
static struct etimer Etimer_Broadcast;
etimer_set(&Etimer_Broadcast,3*CLOCK_SECOND);
while(1){
PROCESS_YIELD(); //faster than process_pause
if(!(uip_closed() || uip_aborted() || uip_timedout())) {
if(etimer_expired(&Etimer_Broadcast))
{
Timeout_Event_handler();
etimer_reset(&Etimer_Broadcast);
}
if(ev==tcpip_event)
{
tcpip_event_handler();
} } } PROCESS_END(); } // end of process.

```

Listing B.2 Final code for the smart fire detection system's client side.

```

#include "contiki.h"
#include "contiki-lib.h"
#include "contiki-net.h"
#include "net/ip/uiplib.h"
#include "net/ipv6/uiplib-ds6.h"
#include "net/ip/uiplib-udp-packet.h"
#include <stdio.h>
#include <string.h>
#define DEBUG DEBUG_PRINT
#include "net/ip/uiplib-debug.h"
#include "dev/light-sensor.h"
#include "dev/sht11/sht11-sensor.h"
#include "dev/leds.h"
PROCESS(unicast_receiver,"");
AUTOSTART_PROCESSES(&unicast_receiver);
static void receiver(struct simple_udp_connection *c, const uip_ipaddr_t *sender_addr, uint16_t sender_port, const uip_ipaddr_t
*receiver_addr, uint16_t receiver_port, const uint8_t *data, uint16_t datalen) {
    char *text=0;
    text=(char*)data;
    printf("Data received from ");
    uip_debug_ipaddr_print(sender_addr);
    printf(" on port %d from port %d with length %d: %s\n", receiver_port, sender_port, datalen, data);
    if(strcmp(text,"on")==0){
        leds_on(LEDS_GREEN);
        printf("it is on");
    }
    else if(strcmp(text,"off")==0){
        leds_off(LEDS_GREEN);
        printf("it is off");
    }
    data=0;
}
static struct simple_udp_connection unicast_connection;
PROCESS_THREAD(unicast_receiver, ev, data){
    PROCESS_BEGIN();
    simple_udp_register(&unicast_connection, 8888, NULL, 8888, receiver);
    while(1) {
        PROCESS_WAIT_EVENT();
    }
    PROCESS_END();
}

```

Listing B.3 Final code for the Actuator node of the smart lighting system case study.

```

#include "contiki.h"
#include "contiki-lib.h"
#include "contiki-net.h"
#include "net/ip/uiplib.h"
#include "net/ipv6/uiplib-ds6.h"
#include "net/ip/uiplib-udp-packet.h"
#include <stdio.h>
#include <string.h>
#define DEBUG DEBUG_PRINT
#include "net/ip/uiplib-debug.h"
#include "dev/light-sensor.h"
#include "dev/sht11/sht11-sensor.h"
static int i;
static uint8_t state;
static int value[4];
PROCESS(Process_Thread,"light process");
AUTOSTART_PROCESSES(&Process_Thread);
static struct uip_udp_conn *Client_Connection;
static char buf[100];
static void Timeout_Event_handler(){
    static int seq_id;
    printf("Client sending to: ");
    sprintf(buf,"Light %d ",value[2]);
    printf(" (msg: %s)\n", buf);
    uip_udp_packet_send(Client_Connection, buf, strlen(buf));
}
static void tcpip_event_handler(void){
    char *str;
    if(uip_newdata()) {

```

```

str = uip_appdata;
str[uip_datalen()] = '\0';
printf("DATA recv %s\n", str);
PRINT6ADDR(&Client_Connection->ripaddr); } //
} //end of tcpip_handler
PROCESS_THREAD(Process_Thread,ev,data){
PROCESS_BEGIN();
uip_ipaddr_t addr_u;
uip_ipaddr_t ipaddr;
uip_ip6addr(&ipaddr,0xfe80, 0, 0, 0,0xc30c,0x0000,0x0000,0x0001); //write server address here.
printf("connection server address is \r\n");
PRINT6ADDR(&ipaddr);
printf("done \r\n");
PRINTF("UDP client process started\n");
Client_Connection=udp_new(&ipaddr,UIP_HTONS(5556),NULL);
udp_bind(Client_Connection,UIP_HTONS(5555));
PRINTF("Created a connection with the server ");
PRINT6ADDR(&Client_Connection->ripaddr);
PRINTF(" local and remote port %u and %u\n", UIP_HTONS(Client_Connection->lport), UIP_HTONS(Client_Connection->rport));
static struct etimer Etimer;
etimer_set(&Etimer,4*CLOCK_SECOND);
simple_udp_register(&unicast_connection, 8888, NULL, 8888, receiver);
uip_ip6addr(&addr_u,0xfe80, 0, 0, 0,0xc30c,0x0000,0x0000,0x0003);
char buf[20]; // the data which will be sent
while(1){
if(!(uip_closed() || uip_aborted() || uip_timedout())) {
PROCESS_YIELD(); //faster than process_pause
SENSORS_ACTIVATE(light_sensor);
value[2]=(int)light_sensor.value(0)*0.7;
printf("Light: %d\n",value[2]);
SENSORS_DEACTIVATE(light_sensor);
if(etimer_expired(&Etimer)) {
Timeout_Event_handler();
etimer_reset(&Etimer);
printf("sending unicast to ");
PRINT6ADDR(&addr_u);
printf("\n");
if(value[2]>100)
simple_udp_sendto(&unicast_connection, "on", strlen("on") + 1, &addr_u);
else
simple_udp_sendto(&unicast_connection, "off", strlen("off") + 1, &addr_u);
}
else if(ev==tcpip_event) {
tcpip_event_handler();
}
}}
PROCESS_END();}

```

Listing B.4 Final code for the client side of the smart lighting system case study.

```

#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include "od.h"
#include "shell.h"
#include "xtimer.h" // always include this.
#include "thread.h" //for threads
#include "msg.h" //for threads
#include "periph/gpio.h"
#include "net/ipv6/addr.h" //for udp and tcp
#include "net/af.h" //for udp
#include "net/protnum.h" //for udp
#include "net/sock/udp.h" //for udp
gpio_t g5 = GPIO5;
char udp_stack[THREAD_STACKSIZE_MAIN];
uint8_t buf[128];
char message[130];
extern int _gnrc_netif_config(int argc, char **argv);
void *udp_thread(void)
{
sock_udp_ep_t local = SOCK_IPV6_EP_ANY;
sock_udp_t Node_Socket;

```

```

local.port = 1212;
if (sock_udp_create(&Node_Socket, &local, NULL, 0) < 0)
{
puts("Problem while creating UDP socket");
return NULL;
}
puts("Configured network interfaces are shown below:");
_gnrc_netif_config(0, NULL);
while (1)
{
sock_udp_ep_t remote;
ssize_t res;
if ((res = sock_udp_recv(&Node_Socket, buf, sizeof(buf), SOCK_NO_TIMEOUT,
&remote)) >= 0)
{
sprintf(message, "%s", buf);
puts("A message is received by this server");
puts(message);
if (strcmp(message, "Dry") == 0)
{
printf("match");
gpio_set(g5);
xtimer_sleep(3);
gpio_clear(g5);
}
if (sock_udp_send(&Node_Socket, buf, res, &remote) < 0)
{
puts("Error during sending a reply to client");
}
}
}
return NULL;
}
int main(void)
{
thread_create(udp_stack, sizeof(udp_stack),
THREAD_PRIORITY_MAIN, THREAD_CREATE_WOUL_YIELD,
udp_thread, NULL, "udp_thread");
if (gpio_init(g5, GPIO_OUT) < 0) {
puts("Error while initializing GPIO's \n");
return 1;
}
else
puts("leds init ok");
gpio_clear(g5);
while(1){
//printf("main\n");
thread_yield();
}
return 0;
}

```

Listing B.5 Final code of the server side of the smart irrigation case study.

```

#include <stdio.h>
#include <string.h>
#include "xtimer.h" //for udp and tcp and processes
#include "thread.h" //for threads
#include "msg.h" //for threads
#include "net/ipv6/addr.h" //for udp and tcp
#include "net/af.h" //for udp and tcp
#include "net/protnum.h" //for udp
#include "net/sock/udp.h" //for udp
#include "periph/gpio.h"
char udp_stack[THREAD_STACKSIZE_MAIN];
gpio_t g15 = GPIO15;
gpio_t g2 = GPIO2;
uint8_t buf[30];
void *udp_thread(void)
{
sock_udp_ep_t local = SOCK_IPV6_EP_ANY;
sock_udp_t Client_Socket;
local.port = 1212;

```

```

if (sock_udp_create(&Client_Socket, &local, NULL, 0) < 0) {
puts("Error creating UDP sock");
return NULL;
}
while (1) {
sock_udp_ep_t remote = { .family = AF_INET6 };
ssize_t res;
remote.port = 12345;
ipv6_addr_from_str((ipv6_addr_t *)&remote.addr.ipv6, "fe80::4c2c:2f99:9ae8:73cb");
if (gpio_read(g15) == 0) {
if (sock_udp_send(&Client_Socket, "Dry", sizeof("Dry"), &remote) < 0) {
puts("Error sending message to server");
sock_udp_close(&Client_Socket);
return NULL;
}
if ((res = sock_udp_rcv(&Client_Socket, buf, sizeof(buf), 1 * US_PER_SEC,
NULL)) < 0) {
if (res == -ETIMEDOUT) {
puts("Timed out");
}
else {
puts("Error receiving message");
}
}
else {
printf("Received message is prompting: \n");
for (int i = 0; i < res; i++) {
printf("%c", buf[i]);
}
printf("\n");
}
xtimer_sleep(1);
return NULL;
}
int main(void)
{
thread_create(udp_stack, sizeof(udp_stack),
THREAD_PRIORITY_MAIN, THREAD_CREATE_WOUL_YIELD,
udp_thread, NULL, "udp_thread");
if (gpio_init(g15, GPIO_IN) || gpio_init(g2, GPIO_OUT) < 0) {
puts("Error while initializing GPIO's \n");
return 1;
}
else
puts("leds init ok");
while(1){
gpio_toggle(g2);
printf("%d \n", gpio_read(g15));
xtimer_sleep(1);
thread_yield();
}
return 0;
}

```

Listing B.6 Final code of the client side of the smart irrigation case study.

```

#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include "od.h"
#include "shell.h"
#include "xtimer.h" // always include this.
#include "thread.h" //for threads
#include "msg.h" //for threads
#include "periph/gpio.h"
#include "net/ipv6/addr.h" //for udp and tcp
#include "net/af.h" //for udp
#include "net/protnum.h" //for udp
#include "net/sock/udp.h" //for udp
gpio_t g2 = GPIO2;
char udp_stack[THREAD_STACKSIZE_MAIN];
uint8_t buf[128];
char message[130];
extern int _gnrc_netif_config(int argc, char **argv);

```

```

void *udp_thread(void){
sock_udp_ep_t local = SOCK_IPV6_EP_ANY;
sock_udp_t Node_Socket;
local.port = 1212;
if (sock_udp_create(&Node_Socket, &local, NULL, 0) < 0) {
puts("Problem while creating UDP socket");
return NULL;}
puts("Configured network interfaces are shown below:");
_gnrc_netif_config(0, NULL);
while (1) {
sock_udp_ep_t remote;
ssize_t res;
if ((res = sock_udp_recv(&Node_Socket, buf, sizeof(buf), SOCK_NO_TIMEOUT,&remote)) >= 0) {
sprintf(message,"%s",buf);
puts("A message is received by client ");
puts(message);
if(strcmp(message,"o")==0){
gpio_set(g2);//open the lights
}else if(strcmp(message,"f")==0){
gpio_clear(g2);}//close the lights
if (sock_udp_send(&Node_Socket, msg, res, &remote) < 0) {
puts("Error during sending a reply to client");
} } }
return NULL;}
int main(void){
thread_create(udp_stack, sizeof(udp_stack),
THREAD_PRIORITY_MAIN , THREAD_CREATE_WOUL_YIELD,
udp_thread, NULL, "udp_thread");
if(gpio_init(g2,GPIO_OUT)< 0){
puts("Error while initializing GPIO's \n");
return 1; }
else
puts("leds init ok");
gpio_clear(g2);
while(1){
//printf("main\n");
thread_yield();}
return 0;}

```

Listing B.7 Final code of the server side of the smart lighting case study.

```

#include <stdio.h>
#include <string.h>
#include "xtimer.h" //for udp and tcp and processes
#include "thread.h" //for threads
#include "msg.h" //for threads
#include "net/ipv6/addr.h" //for udp and tcp
#include "net/af.h" //for udp and tcp
#include "net/protnum.h" //for udp
#include "net/sock/udp.h" //for udp
#include "periph/gpio.h"
char udp_stack[THREAD_STACKSIZE_MAIN];
gpio_t g15 = GPIO15;
uint8_t buff[30];
void *udp_thread(void *arg){
(void) arg;
sock_udp_ep_t local = SOCK_IPV6_EP_ANY;
sock_udp_t Client_Socket;
local.port = 1212;
if (sock_udp_create(&Client_Socket, &local, NULL, 0) < 0) {
puts("Error creating UDP sock");
return NULL;
} while (1) {
sock_udp_ep_t remote = { .family = AF_INET6 };
ssize_t res;
remote.port = 12345;
ipv6_addr_from_str((ipv6_addr_t *)&remote.addr.ipv6,"fe80::32ae:a4ff:fe19:60e8");
char msg=0;
xtimer_sleep(3600); // wait for 1 hour
if(gpio_read(g15)==0){
msg="o"; }
else{
msg="f";}

```

```

if (sock_udp_send(&Client_Socket, msg, sizeof(msg), &remote) < 0) {
puts("Error sending message to server");
sock_udp_close(&Client_Socket);
return NULL;}
if ((res = sock_udp_rcv(&Client_Socket, buf, sizeof(buf), 1 * US_PER_SEC,NULL)) < 0){
if (res == -ETIMEDOUT){
puts("Timed out");}
else {
puts("Error receiving message");}
}else{
printf("Received message is prompting: \n");
for (int i = 0; i < res; i++) {
printf("%c", buf[i]); }
printf("\n\n");}
}
xtimer_sleep(1);
return NULL;}
int main(void)
{thread_create(udp_stack, sizeof(udp_stack),THREAD_PRIORITY_MAIN , THREAD_CREATE_WOUL_YIELD,udp_thread, NULL,
"udp_thread");
if(gpio_init(g15,GPIO_IN)< 0){
puts("Error while initializing GPIO's \n");
return 1; }
else {
puts("leds init ok"); }
while(1){
thread_yield();
return 0;}

```

Listing B.8 Final code of the client side of the smart lighting case study.

```

all: Server Client
APPS+=powertrace
CONTIKI=../..
APPS += webbrowser
PROJECT_SOURCEFILES += httpd-simple.c
WITH_UIP6=1
CFLAGS+= -DUIP_CONF_IPV6_RPL
UIP_CONF_IPV6=1
UIP_CONF_TCP=1
HAVE_BANKING=1
UIP_CONF_IPV6=1
UIP_CONF_RPL=1
WITH_UIP6=1
UIP_CONF_IPV6=1
CFLAGS+= -DUIP_CONF_IPV6_RPL
include $(CONTIKI)/Makefile.include

```

Listing B.9 General Makefile to build generated Contiki codes.

```

APPLICATION = client
BOARD ?= esp32-wroom-32
RIOTBASE ?= $(CURDIR)/../..
USEMODULE += gnrc_netdev_default
USEMODULE += auto_init_gnrc_netif
USEMODULE += gnrc_icmpv6_error
USEMODULE += gnrc_ipv6_router_default
USEMODULE += gnrc_udp
USEMODULE += gnrc_sock_check_reuse
USEMODULE += gnrc_sock_udp
USEMODULE += gnrc_ipv6
USEMODULE += ipv4_addr
USEMODULE += gnrc_rpl
USEMODULE += auto_init_gnrc_rpl
USEMODULE += gnrc_pktdump
USEMODULE += gnrc_icmpv6_echo
USEMODULE += shell
USEMODULE += shell_commands
USEMODULE += ps
USEMODULE += netstats_l2
USEMODULE += netstats_ipv6
USEMODULE += netstats_rpl

```

```
USEMODULE += esp_wifi
DEVELHELP ?= 1
QUIET ?= 1
include $(RIOTBASE)/Makefile.include
```

Listing B.10 General Makefile to build generated Riot codes.

```
#!/bin/bash
echo Enter the file name.
read file
echo file : $file
make $file.c;
make TARGET=sky MOTES=/dev/ttyUSB0 $file.upload login;
```

Listing B.11 A script to build and flash Contiki applications.

```
USEMODULE=esp_wifi \
CFLAGS=-DESP_WIFI_SSID=\"SSID_NAME\" -DESP_WIFI_PASS=\"SSID_PASS\" \
make flash-only BOARD=esp32-wroom-32 PORT=/dev/ttyUSB0
```

Listing B.12 Commands to build and flash Riot applications.

```
[comment encoding = UTF-8 /]
[** * The documentation of the module generate. Burak
Karaduman Ege University */]
[module
generate('http://www.eclipse.org/sirius/contikiOS')]
[query public getter( feature : ESP) : String = 'get'+
feature.Name.toUpperFirst () /]
[query public getterPrefix(feature : ESP) : String = if
feature.eClass().name = 'aa' then 'ok' else 'not' endif/]
[query public getAll( arg0:OrderedSet(Tag), arg1: Integer
):OrderedSet(Tag)=
invoke('ContikiOS.accelele.module.deneme','getAll(java.u
til.Collection,int)' ,Sequence{arg0,arg1}) /]
[query public getConnection( arg0:OrderedSet(Tag),
arg1: Integer ):OrderedSet(String)=
invoke('ContikiOS.accelele.module.names','getConnection
(java.util.Collection,int)' ,Sequence{arg0,arg1}) /]
[query public asd( arg0:OrderedSet(Tag), arg1: Integer
):OrderedSet(String)=
invoke('ContikiOS.accelele.module.names','getConnection
(java.util.Collection,int)' ,Sequence{arg0,arg1}) /]
[query public calculateDistance( arg0:OrderedSet(Tag),
arg1: Real ):OrderedSet(String)=
invoke('ContikiOS.accelele.module.names','calculateDista
nce(java.util.Collection,float)' ,Sequence{arg0,arg1}) /]
[template public generateElement(aIoTSystem :
IoTSystem) post (replaceAll("\n", "").trim())]
[comment @main/]
[for(n:Node | aIoTSystem.platform.node)]
[if(n.Sink)]
[file (n.Name.concat('.c'), false, 'UTF-8')]
#include "contiki.h"
#include "contiki-lib.h"
#include "contiki-net.h"
#include <string.h>
#define DEBUG DEBUG_PRINT
#include "net/ip/uip.h"
#include "net/ip/uip-debug.h"
#define UIP_IP_BUF ((struct uip_ip_hdr
*)&uip_buf[1]/UIP_LLH_LEN])
[if (n.Multihop)]
#include "net/rpl/rpl.h"
#include "net/netstack.h"
[/if]
/* Server */
[ if (n.webServer)]
#include "httpd-simple.h"
#include "dev/sht11/sht11-sensor.h"
```

```
#include "dev/light-sensor.h"
#include "dev/leds.h"
#include <stdio.h>
[/if]

[if (n.processThread.hasMT->size() > 0 )]
#include "sys/mt.h"
[/if]

[if (n.webServer)]
PROCESS(web_sense_process, "Sense Web Demo");
PROCESS(webserver_nogui_process, "Web server");
PROCESS_THREAD(webserver_nogui_process, ev,
data){
PROCESS_BEGIN();
httpd_init();
while(1) {
PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
httpd_appcall(data);
}

PROCESS_END();
}
[/if]
[for (p : Process_Thread | n.processThread)]
PROCESS([p.Name/],[p.Description/]);
[/for]
AUTOSTART_PROCESSES(
[for (p : Process_Thread | n.processThread)]
[if (i<>1)]
,
[/if]&[p.Name/]
[/for]]if
(n.webServer)],&webserver_nogui_process,&web_sense_
process[/if]
);
[if (n.webServer)]

/*-----
----*/
static int
get_light(void)
{
return 10 *
light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHET
IC) / 7;
}
```

```

/*-----
----*/
static int
get_temp(void)
{
return ((sht11_sensor.value(SHT11_SENSOR_TEMP) /
10) - 396) / 10;
}
/*-----
----*/
static int
get_humid(void)
{
return
(sht11_sensor.value(SHT11_SENSOR_HUMIDITY));
}
static const char *TOP = "<html><head><title>Contiki
Web Sense</title></head><body>\n";
static const char *BOTTOM = "</body></html>\n";
/*-----
----*/
static char bufs['256']/;
static int blen;
#define ADD(...) do {
blen += sprintf(&bufs['blen']/, sizeof(bufs) - blen,
__VA_ARGS__); \
} while(0)
static
PT_THREAD(send_values(struct httpd_state *s))
{
PSOCK_BEGIN(&s->sout);
SEND_STRING(&s->sout, TOP);
if(strncmp(s->filename, "/index", 6) == 0 ||
s->filename['1']/ == '\0') {
blen = 0;
ADD("<h1>Current readings</h1>\n"
"Light: %u<br>"
"Temperature: %u<deg> C <br>"
"Humid: %u<br>"
"msg: %s",
get_light(), get_temp(), get_humid(), "User Message
Here");
SEND_STRING(&s->sout, bufs);
} else if(s->filename['1']/ == '0') {
/* Turn off leds */
leds_off(LED_ALL);
SEND_STRING(&s->sout, "Turned off leds!");
} else if(s->filename['1']/ == '1') {
/* Turn on leds */
leds_on(LED_ALL);
SEND_STRING(&s->sout, "Turned on leds!");
} else {
if(s->filename['1']/ != 't') {
SEND_STRING(&s->sout, bufs);
}
if(s->filename['1']/ != 'l') {
SEND_STRING(&s->sout, bufs);
}
}
SEND_STRING(&s->sout, BOTTOM);
PSOCK_END(&s->sout);
}
/*-----
----*/
httpd_simple_script_t
httpd_simple_get_script(const char *name)
{
return send_values;
}

```

```

}
/*-----
----*/
PROCESS_THREAD(web_sense_process, ev, data)
{
static struct etimer et;
PROCESS_BEGIN();
etimer_set(&et, CLOCK_SECOND * 2);
SENSORS_ACTIVATE(light_sensor);
SENSORS_ACTIVATE(sht11_sensor);
while(1) {
PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et)
);
etimer_reset(&et);
}

PROCESS_END();
}
[/i] [comment]end of the webservercode[/comment]
[for (p : Process_Thread | n.processThread)]
[for (hasServerConnection : Server_Connection |
p.hasServerConnection)]
[if (p.hasServerConnection.Name.toString()<>' ')]
static struct uip_udp_conn
*[hasServerConnection.Name];
uip_ipaddr_t ipaddr;
[/i]
[/for]
[/for]
[for (p : Process_Thread | n.processThread)]
[for (event : Event | n.processThread.handlesEvent)]
[if p.hasServerConnection.Name->size()>0] [comment]to
send reply server conn must have [/comment]
static void [event.Name/]_handler(void)
{
//static int seq_id;
char buf [ '1' /][50][ '1' /];
[if (event.eClass().name='TCPIP_Event')]
if(uip_newdata()) {
((char *)uip_appdata)['uip_datalen()']/ = 0;
PRINTF("Server received: '%s' from ", (char
*)uip_appdata);
PRINT6ADDR(&UIP_IP_BUF->srcipaddr);
PRINTF("\n");
[/i]
[comment]ilk once Generalization yapilmis eventlerden
TCPIP eClass isimdekileri seç, her birinin at(i)
indexindeki
replyMessage özelliğini al ve son olarak a stringine
eşitle.[/comment]
[let b:Boolean= n.processThread.handlesEvent->select(
event | event.eClass().name='TCPIP_Event')-
>at(i).eGet('sendReply')]
[if (b)]
// Adding UDP Reply
[let a:String =n.processThread.handlesEvent->select(
event | event.eClass().name='TCPIP_Event')-
>at(i).eGet('replyMessage')]
// NOTE: Add ServerConnection to model
[if(p.UDP)]
PRINTF("Server sending reply\n");
uip_ipaddr_copy(&[p.hasServerConnection.Name/-
>ripaddr, &UIP_IP_BUF->srcipaddr);
uip_udp_packet_send([p.hasServerConnection.Name/,
[a/], sizeof([a/]));
uip_create_unspecified(&[p.hasServerConnection.Name/-
->ripaddr);
memset(&[p.hasServerConnection.Name/->ripaddr, 0,
sizeof([p.hasServerConnection.Name/->ripaddr));
[/i]
[if(p.TCP )]
//TCPIP Reply
sprintf(buf, [a/ ]("(%d)" , 1);
PRINTF(buf);
PRINTF("\n\r");
}

```

```

uip_send(buf, strlen(buf));
[/if][/let][if][/let]
} // end of newdata
[/if][/for][/for]
}
int i;
uint8_t state;
PRINTF("Server IPv6 addresses: ");
for(i = 0; i < UIP_DS6_ADDR_NB; i++) {
state = uip_ds6_if.addr_list[ i ][i].state;
if(uip_ds6_if.addr_list[ i ][i].isused &&
(state == ADDR_TENTATIVE || state ==
ADDR_PREFERRED)) {
PRINT6ADDR(&uip_ds6_if.addr_list[ i ][i].ipaddr);
PRINTF("\n");
}}}
[for (pt : PT_Thread | n.ptThreadNode)]
static struct psock ps;
static uint8_t buffer[50];
static
PT_THREAD([pt.Name/]_connection(struct psock *p))
{
PSOCK_BEGIN(p);
PSOCK_READTO(p, '\n');
printf("RCV: %s\n", buffer);
// PSOCK_SEND(p, buffer, PSOCK_DATALEN(p)); use
this if you need a buffered message
PSOCK_CLOSE(p);
PSOCK_END(p);
}
[/for]
[for (p : Process_Thread | n.processThread)]
PROCESS_THREAD([p.Name/],[p.EventArgument/],[p.
DataArgument/]){
[if (n.MultiHop) and (p.UDP)]
struct uip_ds6_addr *root_if;
[/if]
// NOTE: Add ServerConnection to model to create
TCP(w/o psock) and UDP Connection
PROCESS_BEGIN();
[if (n.MultiHop) and (p.UDP)]
uip_ds6_addr_add(&ipaddr, 0, ADDR_MANUAL);
root_if = uip_ds6_addr_lookup(&ipaddr);
if(root_if != NULL) {
rpl_dag_t *dag;
dag =
rpl_set_root(RPL_DEFAULT_INSTANCE,(uip_ip6addr
_t *)&ipaddr);
uip_ip6addr(&ipaddr, 0xaa, 0, 0, 0, 0, 0);
rpl_set_prefix(dag, &ipaddr, 64);
PRINTF("created a new RPL dag\n");
} else {
PRINTF("failed to create a new RPL DAG\n");
}
NETSTACK_MAC.off(1);
[/if]
[if p.hasServerConnection.Name->size()>0]
[if(p.TCP)]
/* printf("TCP server started\n\r");
#endif UIP_CONF_ROUTER
uip_ip6addr(&ipaddr, [n.IP/]);
uip_ds6_set_addr_iid(&ipaddr, &uip_lladdr);
uip_ds6_addr_add(&ipaddr, 0, ADDR_AUTOCONF);
#endif /* UIP_CONF_ROUTER */
tcp_listen(UIP_HTONS([p.hasServerConnection.LocalPo
rt/]));
[/if]
[if (p.UDP)]
PRINTF("UDP server started\n");
[p.hasServerConnection.Name/] = udp_new(NULL,
UIP_HTONS([p.hasServerConnection.LocalPort/]),
NULL);
if([p.hasServerConnection.Name/] == NULL)
{
PRINTF("No UDP connection available!\n");

```

```

}
udp_bind([p.hasServerConnection.Name/],
UIP_HTONS([p.hasServerConnection.listeningPort.No/])
);
PRINTF("Created a server connection with remote
address ");
PRINT6ADDR(&[p.hasServerConnection.Name/]-
>ripaddr);
PRINTF(" local/remote port %u/%u\n",
UIP_HTONS([p.hasServerConnection.Name/]->lport),
UIP_HTONS([p.hasServerConnection.Name/]->rport));
[/if]
[comment] end of server_conn check [/comment]
[if (p.Continuous)]
while(1)
{
PROCESS_YIELD();
[/if]
[for (pt : PT_Thread | n.ptThreadNode)]
[comment]serverdada psock olmalı [/comment]
if(uip_connected()) {
PSOCK_INIT(&ps, buffer, sizeof(buffer));
while(!uip_aborted() || uip_closed() || uip_timedout()) {
PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
printf("\n Accepted client.\n");
[pt.Name/]_connection(&ps);
}
}
[/for]
[for (event:Event | p.handlesEvent)]
[if (event.eClass().name="TCPIP_Event")]
if([p.EventArgument/]==tcpip_event)
{
[event.Name/]_handler();

}
[/if]
}
[/if]
[comment]end of event handling[/comment]
[if (p.Continuous)]
}
[/if]
PROCESS_END();
[/for] [comment]End of the Process Thread[/comment]
[/file] [comment]End of the File[/comment]
[/if]
[/for]
[comment] END OF THE SINK FILE [/comment]
[for(n:Node | aIoTSystem.platform.node)]
[if not(n.Sink)]
[file (n.Name.concat('.c'), false, 'UTF-8')]
#include "contiki.h"
#include "contiki-lib.h"
#include "contiki-net.h"
#include "net/ip/uip.h"
#include "net/ipv6/uip-ds6.h"
#include "net/ip/uip-udp-packet.h"
#include <stdio.h>
#include <string.h>
#define DEBUG DEBUG_PRINT
#include "net/ip/uip-debug.h"
[if (n.processThread.hasSensor-
>select(p/p.Name.toString()<>'invalid')-
>at(1).Name<>'invalid')]
#include "dev/light-sensor.h"
#include "dev/sht11/sht11-sensor.h"
[/if]
[if (n.MultiHop)]
static uip_ipaddr_t server_ipaddr;
[/if]
[for (p : Process_Thread | n.processThread)]

```

```

[if (p.powertrace.toString()='true')]
#include "powertrace.h"
[/if]
[/for]
/* Client Code*/
static int i;
static uint8_t state;
[if (n.processThread.hasSensor->size()>0)]
static int value['[4]'];
[/if]

[for (p : Process_Thread | n.processThread)]
PROCESS([p.Name/],[p.Description/]);
[/for]
AUTOSTART_PROCESSES(
[for (p : Process_Thread | n.processThread)]
[if (i<>1)]
,
[/if]&[p.Name/]
[/for]
);
[comment] Define connections TCP ve UDP burada
ayrilmali [/comment]
[for (p : Process_Thread | n.processThread)]
[for (hasClientConnection : Client_Connection |
p.hasClientConnection)]
[if (p.hasClientConnection.Name.toString()<>' ')]
static struct uip_udp_conn
*[hasClientConnection.Name/];
[/if]
[/for]
[/for]
static char buf [ ' ' /][100/];
[for (event : Event | n.processThread.handlesEvent)]
[comment] it is ok so far [/comment]
[if (event.eClass().name='TCPIP_Event')]
static void
[event.eClass().name.toLowerCase()/]_handler(void)
{
char *str;
if(uip_newdata())
{
str = uip_appdata;
str ['[uip_datalen()']/] = '\0';
printf("DATA recv '%s'\n", str);

[for (p : Process_Thread | n.processThread)]
[for (hasClientConnection : Client_Connection |
p.hasClientConnection)]
[if (p.hasClientConnection.Name.toString()<>' ')]
PRINT6ADDR(&[hasClientConnection.Name/]-
>ripaddr);
[/if]
[/for]
[/for]
} //
} //end of tcpip_handler

[elseif (event.eClass().name='Time_Event')]
[elseif (event.eClass().name='Process_Event')]
[elseif (event.eClass().name='Timeout_Event')]
static void [event.Name/]_handler(void){
static int seq_id;
printf("Client sending to: ");
sprintf(buf,"Temp %d, Humid %d, Light %d seq_id %d
\n",value['[0]'/],value['[1]'/],value['[2]'/],++seq_id);
[comment] sprintf(buf,"Hello %d from the
client",++seq_id);[/comment]
printf(" (msg: %s)\n", buf);
[for (p : Process_Thread | n.processThread)]
[for (hasClientConnection : Client_Connection |
p.hasClientConnection)]
[if (p.hasClientConnection.Name.toString()<>' ')]
[if not(n.Multihop)]

```

```

uip_udp_packet_send([hasClientConnection.Name/],n,
buf, UIP_APPDATA_SIZE);
[else]
uip_udp_packet_sendto([hasClientConnection.Name/],
buf, strlen(buf),
&server_ipaddr,
UIP_HTONS([p.hasClientConnection.remotePort.No/]));
[/if]
[/if]
[if (p.TCP)]
uip_send(buf, strlen(buf));
[/if]
[/for]
[/for]
}
[elseif (event.eClass().name<>'NULL')]
static void [event.Name/]_handler(void)
{
}
[/if][[/for]
//----- PROTOTHEADS STARTS-----
//-----
[for (p : Process_Thread | n.processThread)]
[if (p.includePsocket)]
[for (pt : PT_Thread | n.ptThreadNode)]
static struct psock [pt.Name/];
static uint8_t buffer_[pt.Name/][100/]; [comment] this
buffer is required for process_thread which includes TCP
conn for
psock[/comment]
[/for]
[for (pt : PT_Thread | n.ptThreadNode)]
static int
[pt.Name/]_connection(struct psock
*[pt.hasPT_ArgumentType.Name/])
{
PSOCK_BEGIN([pt.hasPT_ArgumentType.Name/]);
//
PSOCK_SEND_STR([pt.hasPT_ArgumentType.Name/],
"GET / HTTP/1.0\r\n");
//
PSOCK_SEND_STR([pt.hasPT_ArgumentType.Name/],
"Server: Contiki example protosocket client\r\n");
[for (psock : Psock | aloTSystem.platform.psock->at(i))]
[if (psock.hasOutgoingData.Name.toString()<>'invalid')]
sprintf(buf,"Temp %d, Humid %d, Light %d msg %s \n
",value['[0]'/],value['[1]'/],value['[2]'/],[psock.hasOutgoi
ngData.Name/]);
[/if]
[/for]
static int seq_id;
printf("Client sending msg using PT: ");
PSOCK_SEND_STR([pt.hasPT_ArgumentType.Name/],
buf);
// while(1)
// {
PSOCK_READTO([pt.hasPT_ArgumentType.Name/],
"\n");
printf("Got: %s", buffer_[pt.Name/]);
// }

PSOCK_END([pt.hasPT_ArgumentType.Name/]);
}
[/for]
[/if]
[/for]
//----- PROTOTHEADS ENDS-----
//-----
[for (p : Process_Thread | n.processThread)]
[p.Unicast.toString()/]
PROCESS_THREAD([p.Name/],[p.EventArgument/],[p.
DataArgument/])
{
[if (p.powertrace)]

```

```

static int print_pwr = 0;
[/if]
PROCESS_BEGIN();
[for (hasClientConnection : Client_Connection |
p.hasClientConnection)]
[if (p.hasClientConnection.Name.toString() <> ' ')]
uip_ipaddr_t ipaddr;
uip_ip6addr(&ipaddr, [n.IP/]); //write server address
here.
printf("connection server address is \r\n");
PRINT6ADDR(&ipaddr);
printf("done \r\n");
[/if]
[/for]
// NOTE: udp requires client connection to generate
codes, add client connection to model.
[if(p.UDP)]
// NOTE: udp requires client connection to generate
codes, add client connection to model.
PRINTF("UDP client process started\n");
[for (hasClientConnection : Client_Connection |
p.hasClientConnection)]
[if (p.hasClientConnection.Name.toString() <> ' ')]
p.hasClientConnection.Name/=udp_new(&ipaddr, UIP_
HTONS([p.hasClientConnection.remotePort.No/]), NULL
);
udp_bind([p.hasClientConnection.Name/], UIP_HTONS([
p.hasClientConnection.LocalPort/]));
PRINTF("Created a connection with the server ");
PRINT6ADDR(&[p.hasClientConnection.Name/]-
>ripaddr);
PRINTF(" local/remote port %u%u\n",
UIP_HTONS([p.hasClientConnection.Name/]->lport),
UIP_HTONS([p.hasClientConnection.Name/]->rport));
[/if]
[/for]
[/if]
[comment][for (multithreading: Multithreading
|p.hasMT)]
static struct mt_thread [multithreading.Name/];
[/for][comment]
[if (p.hasEtimer.Name.toString() <> 'invalid')]
static struct etimer [p.hasEtimer.Name/];
[/if]
[if (p.hasStimer.Name.toString() <> 'invalid')]
static struct stimer [p.hasStimer.Name/];
[/if]
[if (p.hasCtimer.Name.toString() <> 'invalid')]
static struct ctimer [p.hasCtimer.Name/];
[/if]
[if (p.hasRtimer.Name.toString() <> 'invalid')]
static struct rtimer [p.hasRtimer.Name/];
[/if]
[if (p.hasTimer.Name.toString() <> 'invalid')]
static struct timer [p.hasTimer.Name/];
[/if]
[if (p.hasEtimer.Name.toString() <> 'invalid')]
etimer_set(&[p.hasEtimer.Name/], [p.hasEtimer.Period/]*
CLOCK_SECOND);
[/if]
[if (p.hasStimer.Name.toString() <> 'invalid')]
stimer_set(&[p.hasStimer.Name/], [p.hasStimer.Period/]*
CLOCK_SECOND);
[/if]
[if (p.hasCtimer.Name.toString() <> 'invalid')]
ctimer_set(&[p.hasCtimer.Name/], [p.hasCtimer.Period/]*
CLOCK_SECOND);
[/if]
[if (p.hasRtimer.Name.toString() <> 'invalid')]
rtimer_set(&[p.hasRtimer.Name/], [p.hasRtimer.Period/]*
CLOCK_SECOND);
[/if]
[if (p.hasTimer.Name.toString() <> 'invalid')]
timer_set(&[p.hasTimer.Name/], [p.hasTimer.Interval/]*
CLOCK_SECOND);

```

```

[/if]
[if (p.powertrace)]
powertrace_sniff(POWERTRACE_ON);
[/if]
[if (p.Continuous)]
while(1){
[/if]
[if(p.TCP)] [comment] this if also must includes psocks
[/comment]
PRINTF("TCP client process started\n\r");
uip_ip6addr(&ipaddr, [n.IP/]);
tcp_connect(&ipaddr,
UIP_HTONS([p.hasClientConnection.LocalPort/]),
NULL);
printf("Connecting...\n\r");
PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
if(uip_aborted() || uip_timedout() || uip_closed())
{
printf("Could not establish connection\n\r");
}
else if(uip_connected()) //if connection is established then
start listening.
{
printf("Connected\n\r");
[if (p.includePsocket)]
[for (pt : PT_Thread | n.ptThreadNode)]
PSOCK_INIT(&[pt.Name/], buffer_[pt.Name/],
sizeof(buffer_[pt.Name/]));
[/for]
[/if]
[comment]after this code generation for psock is made
since it has same samples with TCP [/comment]
[/if] [comment]End of cliet tcp and psock [/comment]

[if (p.TCP) or (p.UDP) or (p.includePsocket)]
PROCESS_YIELD(); //faster than process_pause
while(!(uip_closed() || uip_aborted() || uip_timedout()))
{
[/if]
[if (p.includePsocket)]
[for (pt : PT_Thread | n.ptThreadNode)]
[pt.Name/]_connection(&[pt.Name/]);
PROCESS_WAIT_EVENT_UNTIL(ev == tcpip_event);
// wait until any tcp or udp event occurs
printf("\nConnection closed.\n");
[/for]
}
[/if]
[for (s: Sensor| p.hasSensor)][comment]start of sensor
sampling code [/comment]
[if (s.Temperature)]
PROCESS_WAIT_EVENT();
SENSORS_ACTIVATE(sht11_sensor);
value['[0]'/]=(((sht11_sensor.value(SHT11_SENSOR_TEM
P)/10)-396)/10)-6;
printf("Temp: %d\n", value['[0]'/]);
SENSORS_DEACTIVATE(sht11_sensor);
[/if]
[if (s.Light)]
PROCESS_WAIT_EVENT();
SENSORS_ACTIVATE(light_sensor);
value['[2]'/]=(int)light_sensor.value(0)*0.7;
printf("Light: %d\n", value['[2]'/]);
SENSORS_DEACTIVATE(light_sensor);
[/if]
[if (s.Humidity)]
PROCESS_WAIT_EVENT();
SENSORS_ACTIVATE(sht11_sensor);
printf("Humid: %d\n",
sht11_sensor.value(SHT11_SENSOR_HUMIDITY));
value['[1]'/]=sht11_sensor.value(SHT11_SENSOR_HUMI
DITY);
SENSORS_DEACTIVATE(sht11_sensor);
[/if]

```

```

[/for] [comment]end of sensor sampling code [/comment]
[for (event:Event | p.handlesEvent)]
[if (event.eClass().name=="TCPIP_Event")]
//PROCESS_YIELD();
if([p.EventArgument/]==tcip_event)
{
[event.eClass().name.toLower()/]_handler();
}
[elseif (event.Name.contains("Timeout"))]
if(etimer_expired(&[p.hasEtimer.Name/]))
{
[event.Name/]_handler();
etimer_reset(&[p.hasEtimer.Name/]);
}
[/if]
[/for]
[if (p.TCP) OR (p.UDP) OR (p.includePsocket)]
} // end of uip control.
[/if]

[if (p.powertrace)]
if (print_pwr == 0) {
powertrace_print("#P");
}
if (++print_pwr == 3) {
print_pwr = 0;
}
[/if]
[if (p.Continuous)]
} // end of while loop
[/if]

PROCESS_END();
} // end of process.
[if (p.Unicast)]
static struct simple_udp_connection unicast_connection;
uip_ip6addr_t addr_unicast; // variable to keep ipv6
address
static void
receiver(struct simple_udp_connection *c,
const uip_ipaddr_t *sender_addr,
uint16_t sender_port,
const uip_ipaddr_t *receiver_addr,
uint16_t receiver_port,
const uint8_t *data,
uint16_t datalen)
{
printf("Data received on port %d from port %d with
length %d - %s\n",
receiver_port, sender_port, datalen, data);
printf("sender ip : ");
PRINT6ADDR(sender_addr); // printing the ip of the
message
printf("\n");
//process_start(&unicast_example_process, NULL); //
you may start unicast process manually if so, delete from
autostart.

} // end of receiver function.
/*-----*/
-----*/
static struct etimer send_timer; // to wait for not
interferencing
PROCESS_BEGIN();
[if (p.Continuous)]
while(1){
[/if]
uip_ip6addr(&addr_u,[n.IP/]); //write ip address of the
destination here! // (e.g 0xfe80, 0, 0, 0,
0x0212,0x7401,0x0001,0x0101)
simple_udp_register(&unicast_connection, 1235, // write
destination port instead of 1235
NULL, 1235, receiver);
etimer_set(&send_timer, 4);

```

```

PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&se
nd_timer));
char buf[20]; // the data which will be sent
sprintf(buf, "unicast %d", random_rand()%15); //
changing data
printf("sending unicast to ");
PRINT6ADDR(&addr_u);
printf("\n");
simple_udp_sendto(&unicast_connection, "write message
here", strlen("write message here") + 1, &addr_unicast);
// sending unicast
[if (p.Continuous)]
}
[/if]
PROCESS_END();
[/if]
[/for]
[/file][[/if]][/for]
[comment] PETRI NET [/comment]
[file (Name.concat('.xml'),false,'UTF-8')]
<?xml version="1.0" encoding="ISO-8859-1"?><pnml>
<net id="Net-One" type="P/T net">
[for (l:LogMan | aIoTSystem.logman)]
[for (e:ESP | aIoTSystem.esp)]
<place id="[e.Name/]">
</name>
<initialMarking>
<value>1</value> [comment]value declares how many
token does it contain.[/comment]
</place>
<transition id="T_[e.Name/].[I.URL/]">
<graphics>
<position x="435" y="[(i*90)/]">
</graphics>
<name>
<value>T_[e.Name/].[I.URL/]</value>
<graphics>
<offset x="0" y="0">
<arc id="[e.Name/] to T_[I.URL/]" source="[e.Name/]"
target="T_[e.Name/].[I.URL/]">
>
</tagged>
<arcpath id="000" x="1061" y="162"
curvePoint="false">
<arcpath id="001" x="1116" y="162"
curvePoint="false">
<type value="normal">
</arc>
<arc id="[e.Name/] to T_[I.URL/]"
source="T_[e.Name/].[I.URL/]" target="[I.URL/]">
<arcpath id="000" x="1061" y="162"
curvePoint="false">
<arcpath id="001" x="1116" y="162"
curvePoint="false">
<type value="normal">
</arc>
[/for]
[/for]

[for (l:LogMan | aIoTSystem.logman)]
[for (t : Tag | tag)]
<place id="[t.Name/]">
<graphics>
<position x="[760+(i*20)/]" y="[300+(i*50)/]">
</place>
[/for]
[/for]
[for (r:RaspberryPi | aIoTSystem.raspberrypi)]
<place id="[r.Name/]">
<graphics>
<position x="615" y="[315+(i*45)/]">
</graphics>
<name>
<value>[r.Name/]</value>
<graphics>

```

```

<offset x="0" y="0"/>
</graphics>
</name>
<initialMarking>
<value></value>
<graphics>
<offset x="0" y="0"/>
</graphics>
</initialMarking>
<capacity>
<value></value>
</capacity>
</place>
[/for]
[for (l:LogMan | aIoTSystem.logman)] [comment]first of
all create where all ends [/comment]

<place id="[l.URL]"/>
<graphics>
<position x="[210+i+5]" y="[48+(i*10)]"/>
</graphics>
<name>
<value>[l.URL]</value>
<graphics>
<offset x="0" y="0"/>
</capacity>
</place>
[for (c: Channel | channel)]
<transition id="T_[c.Name/]_[l.URL]"/>
<graphics>
<position x="435" y="[315+(i*50)]"/>
</graphics>
<name>
<value>T_[c.Name/]_[l.URL]</value>
<graphics>
<offset x="0" y="0"/>
</priority>
</transition>
<arc id="[c.Name/] to T_[l.URL]" source="[c.Name]"
target="T_[c.Name/]_[l.URL]"/>
<graphics>
<inscription>
<value/>
<graphics/>
</inscription>
<tagged>
<value></value>
</tagged>
<arcpath id="000" x="1061" y="162"
curvePoint="false"/>
<arcpath id="001" x="1116" y="162"
curvePoint="false"/>
<type value="normal"/>
</arc>
<arc id="[c.Name/] to T_[l.URL]"
source="T_[c.Name/]_[l.URL]" target="[l.URL]"/>
<graphics>
<inscription>
<value/>
<graphics/>
</inscription>
<tagged>
<value></value>
</tagged>
<arcpath id="000" x="1061" y="162"
curvePoint="false"/>
<arcpath id="001" x="1116" y="162"
curvePoint="false"/>
<type value="normal"/>
</arc>
[for (t: Tag | tagss)]
[for (r:RaspberryPi| aIoTSystem.raspberrypi)]
[c.Name/] [r.Name/]
[if (c.Name=r.Name)]
<transition id="T_[t.Name/]_[r.Name]"/>

```

```

<graphics>
<position x="735" y="[500+(i*50)]"/>
</graphics>
<name>
<value>T_[t.Name/]_[r.Name/]</value>
<graphics>
<priority>
<value></value>
</priority>
</transition>
[comment]define arcs[/comment]
<arc id="[t.Name/] to [r.Name]"
source="T_[t.Name/]_[r.Name]" target="[r.Name]"/>
<graphics>
</tagged>
<arcpath id="000" x="1061" y="162"
curvePoint="false"/>
<arcpath id="001" x="1116" y="162"
curvePoint="false"/>
<type value="normal"/>
</arc>
<arc id="[t.Name/] to [r.Name]" source="[t.Name]"
target="T_[t.Name/]_[r.Name]"/>
<graphics>
</tagged>
<arcpath id="000" x="1061" y="162"
curvePoint="false"/>
<arcpath id="001" x="1116" y="162"
curvePoint="false"/>
<type value="normal"/>
</arc>
[/if]
[/for]
[/for]
[/for]
[let abc : OrderedSet(String) =
aIoTSystem.logman.tag.getConnection(aIoTSystem.logman.tag,aIoTSystem.logman.tag->size()->asOrderedSet())
[for(t:String|abc)]
[if (i<=abc->size()/2)]
[t.substring(1, t.index('_')-1)]/[t.substring(t.index('_')+1,
t.size()/)]

<arc id="[t.substring(t.index('_')+1, t.size()/)] to
[t.substring(1, t.index('_')-1)]"
source="[t.substring(t.index('_')+1, t.size()/)]"
target="T_[t.substring(1, t.index('_')-1)]/[t.substring(t.index('_')+1, t.size()/)]"/>
<graphics>
<arcpath id="000" x="1061" y="162"
curvePoint="false"/>
<arcpath id="001" x="1116" y="162"
curvePoint="false"/>
<type value="normal"/>
</arc>
<arc id="[t.substring(1, t.index('_')-1)] to
[t.substring(t.index('_')+1, t.size()/)] "
source="T_[t.substring(1, t.index('_')-1)]/[t.substring(t.index('_')+1, t.size()/)]"
target="[t.substring(1, t.index('_')-1)]"/>
<graphics>
<inscription>
<value/>
<graphics/>
</inscription>
<tagged>
<value></value>
</tagged>
<arcpath id="000" x="1061" y="162"
curvePoint="false"/>
<arcpath id="001" x="1116" y="162"
curvePoint="false"/>
<type value="normal"/>

```

```

</arc>
<transition id="T_[t.substring(1, t.index('_')-
1)]_[t.substring(t.index('_')+1, t.size())]">
<graphics>
<position x="[735+(i*10)]" y="[500+(i*50)]"/>
</graphics>
<name>
<value>T_[t.substring(1, t.index('_')-
1)]_[t.substring(t.index('_')+1, t.size())]/</value>
</transition>
[/if]/[/for]/[/let]
<labels x="1250" y="427" width="250" height="68"
border="true">
<text>
Analysis Report
[let abc : OrderedSet(String) =
aIoTSystem.logman.tag.calculateDistance(aIoTSystem.lo
gman.tag,aIoTSystem.platform->at(1).Frequency)-
>asOrderedSet()]
[for(t:String|abc)]
[t/]
[/for]
[/let]
</text>
</labels>
<labels x="1051" y="427" width="404" height="150"
border="true">
<text>
Bill of Materials
ESP8266 x [aIoTSystem.logman.esp->size()/] =
[4*aIoTSystem.logman.esp->size()/]$
RaspberryPi x [aIoTSystem.raspberrypi->size()/] =
[30*aIoTSystem.raspberrypi->size()/]$
TmoteSky x [aIoTSystem.logman.tag->size()/] =
[100*aIoTSystem.logman.tag->size()/]$
IoT Log Manager x [aIoTSystem.logman->size()/] =
[1000*aIoTSystem.logman->size()/]$
</text>
</labels>
</net>
</pnml>
[/file]
[comment]define places[/comment]
[file (Name.concat('.lola'),false,'UTF-8')]
PLACE

[for (l:LogMan | aIoTSystem.logman)][l.URL/],[for
(e:ESP | aIoTSystem.esp)]
[e.Name/],
[/for]/[/for]

[for (l:LogMan | aIoTSystem.logman)][for (t : Tag |
tag)][t.Name/],
[/for]/[/for]

[for (r:RaspberryPi
aIoTSystem.raspberrypi)][r.Name/][if
(i<>aIoTSystem.raspberrypi->size())],[/if] [/for];
[comment]END[/comment]
[comment]define Marking[/comment]
MARKING
[for (l:LogMan | aIoTSystem.logman)][for (e:ESP |
aIoTSystem.esp)][e.Name/]:1,[/for]/[/for]
[for (l:LogMan | aIoTSystem.logman)][for (t : Tag |
tag)][t.Name/]:1[if (i<>aIoTSystem.logman.tag-
>size())],[/if]
[/for]/[/for];
[for (l:LogMan | aIoTSystem.logman)][comment] sensors
to raspberrypi[/comment]
[for (c: Channel | channel)]
[for (r:RaspberryPi | aIoTSystem.raspberrypi)]
[for(t:Tag|aIoTSystem.raspberrypi.tags )]
[if (c.Name=r.Name)]
[if(c.tagss->at(i).Name.toString()<>'invalid')]

```

```

TRANSITION [r.Name+'_'+c.tagss-
>at(i).Name.toString()/]
CONSUME [c.tagss->at(i).Name/]:1;
PRODUCE [r.Name/]:1;
[/if]/[/if]
[/for]/[/for]/[/for]/[/for]
[for (l:LogMan | aIoTSystem.logman)] [comment] sensors
to IoT Log Manager[/comment]
[for (r:RaspberryPi | aIoTSystem.raspberrypi)]

TRANSITION [r.Name+'_'+l.URL/]
CONSUME [r.Name/]:1;
PRODUCE [l.URL/]:1;
[/for]
[/for]
[for (l:LogMan | aIoTSystem.logman)]
[for (e:ESP | aIoTSystem.esp)]
TRANSITION [e.Name/]
CONSUME [e.Name/]:1;
PRODUCE [l.URL/]:1;
[/for]/[/for]
[comment]for meshed sinks[/comment]
[let abc : OrderedSet(String) =
aIoTSystem.logman.tag.asd(aIoTSystem.logman.tag,aIoT
System.logman.tag->size())->asOrderedSet()]
[for(t:String|abc)]
[if (i<=abc->size()/2)]

TRANSITION [t.substring(1, t.index('_')-
1)]_[t.substring(t.index('_')+1, t.size())/]

CONSUME [t.substring(t.index('_')+1, t.size())/]:1;
PRODUCE [t.substring(1, t.index('_')-1)/]:1;
[/if]/[/for]/[/let]/[/file]
[comment]***** RIOT RIOT RIOT
***** [/comment]
[for(n:NodeR | aIoTSystem.platformR.nodeR)]
[if(n.isServer)]
[file
(n.Name.concat('_SMain').concat(i.toString()+'.c'),false,'
UTF-8')]
[comment] Server Main Code Riot [/comment]
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include "od.h"
#include "shell.h"
#include "xtimer.h" // always include this.
#include "thread.h" //for threads
#include "msg.h" //for threads
[if (n.MultiHop)]
#include "net/gnrc.h"
#include "net/gnrc/ipv6.h"
#include "net/gnrc/netif.h"
#include "net/gnrc/netif/hdr.h"
[/if]
[if (n.gpioR->size()>0 or n.coaparamsR.gpioparams-
>size()>0)]
#include "periph/gpio.h"
[/if]
[if (n.socketR->at(1).isIPv4) ]
#include "net/ipv4/addr.h" //for udp and tcp
[/if]
[if (n.socketR->at(1).isIPv6) ]
#include "net/ipv6/addr.h" //for udp and tcp
[/if]
[if (n.CoAp)]
#include "net/nanocoap_sock.h" // for nano-coap server
[/if]
[if (n.UDP)]
#include "net/af.h" //for udp
#include "net/protnum.h" //for udp
#include "net/sock/udp.h" //for udp

```

```

[/if]
[if (n.TCP)]
#include "net/sock/tcp.h" // for tcp
[/if]
[if (n.saulR->size()>0)] //for saul
#include "saul_reg.h"
[/if]
[comment]Saul device definitions[/comment]
[for (saul:SAUL| n.saulR)]
static saul_reg_t* saul_[saul.Name/];
[/for]
[comment]Saul device definitions[/comment]
[for (gpio:GPIO| n.coaparamsR.gpioparams)]
gpio_t ['g'+gpio.Number/] = ['GPIO'+gpio.Number/];
[/for]
[comment]Saul device definitions[/comment]
[for (gpio:GPIO| n.gpioR)]
gpio_t ['g'+gpio.Number/] = ['GPIO'+gpio.Number/];
// gpio_toggle(g[gpio.Number/]);
[/for] [comment]define gpios which used in directly by
node or in coap server[/comment]
[comment]***Thread Definiations[/comment]
[if (n.UDP)]
char udp_stack['THREAD_STACKSIZE_MAIN']/];
[comment]***Define udp stack for udp
thread***[/comment]
[/if]
[if (n.TCP)]
char tcp_stack['THREAD_STACKSIZE_MAIN']/];
[comment]***Define tcp stack tcp udp
thread***[/comment]
[/if]
[if (n.threadR->size()>0)]
[for (thread:Thread|n.threadR)]
char
[thread.Name/]_stack['THREAD_STACKSIZE_MAIN']
/];
void *[thread.Name/](void *arg){
(void) arg;
while(1){
thread_yield();}
return NULL;}
[/for]
[/if]
[comment]**End of Thread Definiations for Server
[/comment]
[if (n.UDP)]
uint8_t buf['128']/];
char message['130']/];
/* This function is necessary to get ip address.
extern int _gnrc_netif_config(int argc, char **argv);
void *udp_thread(void)
{
[for (socket:Socket| n.socketR)]
[if (socket.isIPv6)]
sock_udp_ep_t local = SOCK_IPV6_EP_ANY;
[/if]
[if (socket.isIPv4)]
sock_udp_ep_t local = SOCK_IPV4_EP_ANY;
[/if]
sock_udp_t [socket.Name/];
local.port =[socket.LocalPortNumber/];
if (sock_udp_create(&[socket.Name/], &local, NULL, 0) <
0) {
puts("Problem while creating UDP socket");
return NULL;
}
puts("Configured network interfaces are shown
below:");
_gnrc_netif_config(0, NULL);
[ if(n.Multihop)]
gnrc_netif_t *netif;
int iface;
uint16_t port;
ipv6_addr_t addr;
/* get interface, if available */

```

```

iface = ipv6_addr_split_iface(addr_str);
if ((iface < 0) && (gnrc_netif_numof() == 1)) {
netif = gnrc_netif_iter(NULL);
}
else {
netif = gnrc_netif_get_by_pid(iface);
}
if (netif != NULL) {
gnrc_pktsnip_t *netif_hdr =
gnrc_netif_hdr_build(NULL, 0, NULL, 0);
gnrc_netif_hdr_set_netif(netif_hdr->data, netif);
LL_PREPEND(ip, netif_hdr);
}
[/if]
while (1) {
sock_udp_ep_t remote;
ssize_t res;
if ((res = sock_udp_rcv(&[socket.Name/], buf,
sizeof(buf), SOCK_NO_TIMEOUT,
&remote)) >= 0) {
sprintf(message, "%s", buf);
puts("A message is received by this server");
puts(message);

if (sock_udp_send(&[socket.Name/], buf, res, &remote) <
0) {
puts("Error during sending a reply to client");
}}}
return NULL;
}
[/for]
[/if] [comment] end of udp server[/comment]
[comment]TCP server start here[/comment]
static sock_tcp_t sock_queue;
uint8_t buf['128']/];

extern int _gnrc_netif_config(int argc, char **argv);
void *tcp_thread(void)
{
[for (socket:Socket| n.socketR)]
[if (socket.isIPv6)]
sock_udp_ep_t local = SOCK_IPV6_EP_ANY;
[/if]

[if (socket.isIPv4)]
sock_udp_ep_t local = SOCK_IPV4_EP_ANY;
[/if]
sock_tcp_queue_t queue;
local.port = [socket.LocalPortNumber/];
if (sock_tcp_listen(&queue, &local, &sock_queue, 1, 0) <
0) {
puts("Error creating listening queue");
return NULL;}
puts("Listening on port [socket.LocalPortNumber/]");
puts("Configured network interfaces2:");
_gnrc_netif_config(0, NULL);
while (1) {
puts("Configured network interfaces:");
_gnrc_netif_config(0, NULL);
sock_tcp_t *[socket.Name/];
if (sock_tcp_accept(&queue, &[socket.Name/],
SOCK_NO_TIMEOUT) < 0) {
puts("Error accepting new sock");
}
else {
int read_res = 0;
puts("Reading data");
while (read_res >= 0) {
read_res = sock_tcp_read([socket.Name/], &buf,
sizeof(buf),
SOCK_NO_TIMEOUT);
if (read_res < 0) {
puts("Disconnected");
break;}
else {

```

```

int write_res;
printf("Read: \n");
for (int i = 0; i < read_res; i++) {
printf("%c", buf[i]);
}
puts("\n");
if ((write_res = sock_tcp_write([socket.Name/], &buf,
read_res)) < 0) {
puts("Errored on write, finished server loop");
break;
} }
sock_tcp_disconnect([socket.Name/]);
}
sock_tcp_stop_listen(&queue);
return NULL;
}
[/for][comment] end of TCP server[/comment]
[/if (n.CoAp)]
#define COAP_INBUF_SIZE (256U)
#define MAIN_QUEUE_SIZE (8)
static msg_t _main_msg_queue
[["MAIN_QUEUE_SIZE"/]];
extern int _gnrc_netif_config(int argc, char **argv);
[/if]
[comment] Start of main function definition[/comment]
int main(void){
[comment] start of thread creation for server [/comment]
[/if (n.threadR->size())>0]]
[/for (thread:Thread)n.threadR)]
thread_create([thread.Name/]_stack,
sizeof([thread.Name/]_stack),
THREAD_PRIORITY_MAIN - [thread.Priority/],
THREAD_CREATE_WOUL_YIELD,
[thread.Name/], NULL, "[thread.Name/]");
[/for]
[/if]
[/if (n.UDP)]
thread_create(udp_stack, sizeof(udp_stack),
THREAD_PRIORITY_MAIN ,
THREAD_CREATE_WOUL_YIELD,
udp_thread, NULL, "udp_thread");
[/if]
[/if (n.TCP)]
thread_create(tcp_stack, sizeof(tcp_stack),
THREAD_PRIORITY_MAIN ,
THREAD_CREATE_WOUL_YIELD,
tcp_thread, NULL, "tcp_thread");
[/if]
[comment] end of thread creation for server [/comment]
[/if (n.coapparamsR.gpioparams->size())>0]]
if(
[/for (gpio:GPIO| n.coapparamsR.gpioparams)]
gpio_init(g[gpio.Number/],[if(gpio.isInput)]GPIO_IN[else
]GPIO_OUT[/if]) [if(n.coapparamsR.gpioparams-
>size())<>i)]||[/if]
[/for] < 0){
puts("Error while initializing GPIO's \n");
return 1;}
else
puts("leds init ok");
[/if]
[/if (n.gpioR->size())>0]]
if(
[/for (gpio:GPIO| n.gpioR)]
gpio_init(g[gpio.Number/],[if(gpio.isInput)]GPIO_IN[else
]GPIO_OUT[/if]) [if(n.gpioR->size())<>i)]||[/if]
[/for] < 0){
puts("Error while initializing GPIO's \n");
return 1;}
else
puts("leds init ok");
[/if]
[comment]*****COAP STARTS
HERE*****[/comment]
[/if (n.CoAp)]

```

```

puts("RIOT NanoCoap Server");
msg_init_queue(_main_msg_queue,
MAIN_QUEUE_SIZE);
puts("Address is autoconfiguring");
xtimer_sleep(3);
puts("Configured network interfaces:");
_gnrc_netif_config(0, NULL);
uint8_t buf[["COAP_INBUF_SIZE"/]];
sock_udp_ep_t local = { .port=COAP_PORT,
.family=AF_INET6 };
nanocoap_server(&local, buf, sizeof(buf));
[/if]
[comment]*****COAP ENDS
HERE*****[/comment]
while(1){
[comment]saul physical storage definitions[/comment]
[/for (saul:SAUL| n.saulR)]
[/if (saul.Temperature)]
phydat_t phy_[saul.Name/]_temp[i];
[/if]
[/if (saul.Humidity)]
phydat_t phy_[saul.Name/]_humid[i];
[/if]
[/if (saul.Temperature)]
saul_[saul.Name/] =
saul_reg_find_type(SAUL_SENSE_TEMP);
saul_reg_read(saul_[saul.Name/],
&phy_[saul.Name/]_temp[i]); // this contains sensor
result
[/if]

[/if (saul.Humidity)]
saul_[saul.Name/] =
saul_reg_find_type(SAUL_SENSE_HUM);
saul_reg_read(saul_[saul.Name/],
&phy_[saul.Name/]_humid[i]); // this contains sensor
result
[/if]
[/for]
[comment]saul physical storage definitions[/comment]
[/printf("main\n");
[/if (n.threadR->size())>0]]
thread_yield();
[/if]
}
return 0;
} [comment] end of main fuction definition [/comment]
[/file]
[file
(n.Name.concat('_Server').concat(i.toString()+'.c'),false,
UTF-8')]
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "od.h"
#include "shell.h"
[/if (n.gpioR->size())>0 or n.coapparamsR.gpioparams-
>size())>0]]
#include "periph/gpio.h"
[/if]
[/if (n.CoAp)]
#include "net/nanocoap.h"
[/for (coap:CoAPparams|n.coapparamsR)]
static ssize_t _[coap.functionName/]_handler(coap_pkt_t
*pkt, uint8_t *buf, size_t len, void *context){
(void)context;
[/for (gpio:GPIO| n.coapparamsR.gpioparams)]
extern gpio_t g[gpio.Number/];
gpio_toggle(g[gpio.Number/]);
[/for]
return coap_reply_simple(pkt, COAP_CODE_205, buf,
len,
COAP_FORMAT_TEXT, (uint8_t*)RIOT_BOARD,
strlen(RIOT_BOARD));
}

```

```

[/for]
const coap_resource_t coap_resources['/'] = {
COAP_WELL_KNOWN_CORE_DEFAULT_HANDLE
R,
[/for (coap:CoAPParams|n.coapparamsR)]
{ "[coap.functionName/]", [if
(coap.isPost)]COAP_GET[else]COAP_POST[/if], [coap.f
unctionName/]_handler, NULL},
{ "[coap.functionName/]", [if
(coap.isPost)]COAP_GET[else]COAP_POST[/if], [coap.f
unctionName/]_handler, NULL} [if (n.coapparamsR-
>size() <> i), /if]
[/for]
};
const unsigned coap_resources_numof =
sizeof(coap_resources) / sizeof(coap_resources['/0']/);
[/if] [comment]end of coap server[/comment]
[/file] [/if]
[/for] [comment] end of SERVER side [/comment]
[comment]***** Client Code Starts Here
*****[/comment]
[/for (n:NodeR | aIoTSystem.platformR.nodeR)]
[/if not (n.isServer)]
[/file
(n.Name.concat('_CMain').concat(i.toString()+'.c'), false, '
UTF-8')]
[comment] Client Main Code Riot [/comment]
#include <stdio.h>
#include <string.h>
#include "xtimer.h" //for udp and tcp and processes
#include "thread.h" //for threads
#include "msg.h" //for threads
[/if (n.socketR->at(1).isIPv4) ]
#include "net/ipv4/addr.h" //for udp and tcp
[/if]
[/if (n.socketR->at(1).isIPv6) ]
#include "net/ipv6/addr.h" //for udp and tcp
[/if]
[/if (n.TCP) or (n.UDP)]
#include "net/af.h" //for udp and tcp
[/if]
[/if (n.UDP)]
#include "net/protnum.h" //for udp
#include "net/sock/udp.h" //for udp
[/if]
[/if (n.TCP)]
#include "net/sock/tcp.h" //for tcp
[/if]
[/if (n.gpioR->size()>0)]
#include "periph/gpio.h"
[/if]
[/if (n.saulR->size()>0)] //for saul
#include "saul_reg.h"
[/if]
[/if (n.UDP)]
char udp_stack['[THREAD_STACKSIZE_MAIN]'];
[comment]***Define udp stack for udp
thread***[/comment]
[/if]

[/if (n.TCP)]
char tcp_stack['[THREAD_STACKSIZE_MAIN]'];
[comment]***Define tcp stack tcp udp
thread***[/comment]
[/if]
[comment]Saul device definitions[/comment]
[/for (saul:SAUL| n.saulR)]
static saul_reg_t* saul_[saul.Name/];
[/for]
[comment]Saul device definitions[/comment]
[/for (gpio:GPIO| n.gpioR)]
gpio_t ['g'+gpio.Number/] = ['GPIO'+gpio.Number/];
// gpio_toggle(g[gpio.Number/]);
[/for] [comment]define gpios for client node[/comment]
[/if (n.threadR->size()>0)]

```

```

[/for (thread:Thread|n.threadR)]
char
[thread.Name/]_stack['[THREAD_STACKSIZE_MAIN]
'];
void *(thread.Name/)(void *arg)
{
(void) arg;
while(1)
{
thread_yield();
}
return NULL;
}
[/for]
[/if]
[comment]**End of Thread Definations for Client
[/comment]
[/if (n.UDP)]
uint8_t buf['[30]'];
void *udp_thread(void){
[/for (socket:Socket| n.socketR)]
[/if (socket.isIPv6)]
sock_udp_ep_t local = SOCK_IPV6_EP_ANY;
[/if]
[/if (socket.isIPv4)]
sock_udp_ep_t local = SOCK_IPV4_EP_ANY;
[/if]
sock_udp_t [socket.Name/];
local.port = [socket.LocalPortNumber/];
if (sock_udp_create(&[socket.Name/], &local, NULL, 0) <
0) {
puts("Error creating UDP sock");
return NULL;}
while (1) {
[/if (socket.isIPv6)]
sock_udp_ep_t remote = { .family = AF_INET6 };
[/if]
[/if (socket.isIPv4)]
sock_udp_ep_t remote = { .family = AF_INET };
[/if]
ssize_t res;
remote.port = [socket.RemotePortNumber/];
//ipv6_addr_set_all_nodes_multicast((ipv6_addr_t
*)&remote.addr.ipv6,
//
IPV6_ADDR_MCAST_SCP_LINK_LOCAL);
[/if (socket.isIPv6)]
ipv6_addr_from_str((ipv6_addr_t
*)&remote.addr.ipv6, "[socket.IPAddress/]");
[/if]
[/if (socket.isIPv4)]
ipv4_addr_from_str((ipv4_addr_t
*)&remote.addr.ipv4, "[socket.IPAddress/]");
[/if]
if (sock_udp_send(&[socket.Name/], "[socket.Message/]",
sizeof("[socket.Message/]", &remote) < 0) {
puts("Error sending message to server");
sock_udp_close(&[socket.Name/]);
return NULL;}
if ((res = sock_udp_recv(&[socket.Name/], buf,
sizeof(buf), 1 * US_PER_SEC,
NULL)) < 0) {
if (res == -ETIMEDOUT) {
puts("Timed out");}
else {
puts("Error receiving message");
}}
else {
printf("Received message is prompting: \n");
for (int i = 0; i < res; i++) {
printf("%c", buf[i]);}
printf("\n\n");
xtimer_sleep(1);}
return NULL;}
[/for]

```

```

[/if] [comment] end of UDP Client code [/comment]
[comment] Start of TCP Client code [/comment]

[if (n.TCP)]
uint8_t buf[128];
void *tcp_thread(void){
xtimer_sleep(5);
while(1){
printf("****TCP CLIENT****");
int res;
[for (socket:Socket| n.socketR)]
[if (socket.isIPv6)]
sock_tcp_ep_t remote = SOCK_IPV6_EP_ANY;
[/if]
[if (socket.isIPv4)]
sock_tcp_ep_t remote = SOCK_IPV4_EP_ANY;
[/if]
remote.port = [socket.RemotePortNumber/];
[if (socket.isIPv6)]
ipv6_addr_from_str(ipv6_addr_t *)&remote.addr,
"[socket.IPAddress/]");
[/if]
[if (socket.isIPv4)]
ipv4_addr_from_str(ipv4_addr_t *)&remote.addr,
"[socket.IPAddress/]");
[/if]
static sock_tcp_t [socket.Name/];
if (sock_tcp_connect(&[socket.Name/], &remote, 0, 0) <
0) {
puts("Socket problem or Server is offline");
return NULL;}
puts("Sending \"[socket.Message/]!\");
if ((res = sock_tcp_write(&[socket.Name/],
"[socket.Message/]", sizeof("[socket.Message/])))) < 0) {
puts("Errored on write");}
else {
if ((res = sock_tcp_read(&[socket.Name/], &buf,
sizeof(buf),
SOCK_NO_TIMEOUT)) < 0) {
puts("Disconnected");
}
printf("Read: \"");
for (int i = 0; i < res; i++) {
printf("%c", buf[i]);}
}
puts("\");}
sock_tcp_disconnect(&[socket.Name/]);
// return res;
} // end of while loop
[/for][/if]
int main(void)
{
[comment] start of thread creation for client [/comment]
[if (n.threadR->size()>0)]
[for (thread:Thread|n.threadR)]
thread_create([thread.Name/]_stack,
sizeof([thread.Name/]),
THREAD_PRIORITY_MAIN - [thread.Priority/],
THREAD_CREATE_WOUT_YIELD,
[thread.Name/], NULL, "[thread.Name/]");
[/for][/if]
[if (n.UDP)]
thread_create(udp_stack, sizeof(udp_stack),
THREAD_PRIORITY_MAIN ,
THREAD_CREATE_WOUT_YIELD,
udp_thread, NULL, "udp_thread");
[/if]
[if (n.TCP)]
thread_create(tcp_stack, sizeof(tcp_stack),
THREAD_PRIORITY_MAIN ,
THREAD_CREATE_WOUT_YIELD,
tcp_thread, NULL, "tcp_thread");
[/if]
[ (n.gpioR->size()>0)]

```

```

[if (n.gpioR->size()>0)]
if(
[for (gpio:GPIO| n.gpioR)]
gpio_init(g[ gpio.Number/], [if(gpio.isInput)]GPIO_IN[else
]GPIO_OUT[/if]) [if(n.gpioR->size()<>i)]|[/if]
[/for] < 0){
puts("Error while initializing GPIO's \"n");
return 1;
}
else
puts("leds init ok");
[/if]
[comment] end of thread creation for client [/comment]
while(1){
[comment]saul physical storage definitions[/comment]
[for (saul:SAUL| n.saulR)]

[if (saul.Temperature)]
phydat_t phy_[saul.Name/]_temp[i];
[/if]
[if (saul.Humidity)]
phydat_t phy_[saul.Name/]_humid[i];
[/if]
[if (saul.Temperature)]
saul_[saul.Name/] =
saul_reg_find_type(SAUL_SENSE_TEMP);
saul_reg_read(saul_[saul.Name/],
&phy_[saul.Name/]_temp[i]); // this contains sensor
result
[/if]
[if (saul.Humidity)]
saul_[saul.Name/] =
saul_reg_find_type(SAUL_SENSE_HUM);
saul_reg_read(saul_[saul.Name/],
&phy_[saul.Name/]_humid[i]); // this contains sensor
result
[/if][/for]
[comment]saul physical storage definitions[/comment]
//printf("main\n");
[if (n.threadR->size()>0)]
thread_yield();
[/if]}
return 0;
}[/file][/if][/for][/template]

```

Listing B.13 Code generation rules.