



Verification and validation of wireless sensor network protocol properties through the system's emulation

Calypso Barnes

► To cite this version:

Calypso Barnes. Verification and validation of wireless sensor network protocol properties through the system's emulation. Networking and Internet Architecture [cs.NI]. Université Côte d'Azur, 2017. English. NNT : 2017AZUR4047 . tel-01618142

HAL Id: tel-01618142

<https://theses.hal.science/tel-01618142>

Submitted on 17 Oct 2017

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

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École doctorale STIC (Sciences et Technologies de l'Information et de la
Communication)

Thèse de doctorat

Présentée en vue de l'obtention du
grade de docteur en Sciences
Discipline : Electronique
de l'Université Côte d'Azur

présentée et soutenue par
Calypso Barnes

Vérification et Validation de Propriétés de Protocoles pour Réseaux de Capteurs sans Fil grâce au Couplage de la Simulation et de l'Emulation du Système

Dirigée par François Verdier

Soutenue le 28 juin 2017
Devant le jury composé de :

Olivier	Berder	Professeur, Université de Rennes I	Examineur
Jean-Marie	Cottin	Ingénieur chercheur, EDF R&D	Invité
Daniel	Gaffé	Maître de Conférences, Université Côte d'Azur	Invité
David	Navarro	Maître de Conférences HDR, Ecole Centrale de Lyon	Rapporteur
Alain	Pegatoquet	Maître de Conférences HDR, Université Côte d'Azur	Examineur
Frédéric	Pétrot	Professeur, Institut Polytechnique de Grenoble	Rapporteur
Davide	Quaglia	Professeur Assistant, Université de Vérone	Examineur
François	Verdier	Professeur, Université Côte d'Azur	Directeur de thèse

PhD Thesis

In Electronics Engineering

From the University Côte d'Azur

Presented on the 28th of June 2017

Author:

Calypso Barnes

**Supervised by François Verdier, Jean-Marie Cottin
and Alain Pegatoquet**

**Verification and Validation of Wireless
Sensor Network Protocol Properties
through the System's Emulation**



Remerciements

La thèse de doctorat a représenté une période significative de ma vie qui restera à jamais gravée dans ma mémoire. Ce fut un travail laborieux et enrichissant, et je n'aurais pas été aussi loin sans le soutien de certaines personnes pour lesquelles je suis infiniment reconnaissante.

Tout d'abord je souhaite remercier mon directeur de thèse, François Verdier, pour ses conseils et ses directives, et pour avoir toujours su se rendre disponible quand nécessaire. Je le remercie également pour ses enseignements avant et pendant la thèse. Je voudrais remercier mon encadrant Alain Pegatoquet, pour sa disponibilité, sa bonne humeur et sa positivité, ainsi que pour ses critiques constructives qui m'ont permis de faire beaucoup de progrès sur ma qualité rédactionnelle.

Ce travail de thèse n'aurait pas pu être mené sans mon tuteur industriel, Jean-Marie Cottin, qui m'a consacré beaucoup de son temps non seulement durant ma thèse, mais aussi durant mon stage de fin d'études. Je tiens à le remercier pour tout ce qu'il m'a appris, notamment en développement et débogage de code, ainsi que pour son soutien et la confiance qu'il m'a donnée.

Je tiens également à remercier des personnes qui ont beaucoup contribué à mon travail de thèse. Je pense notamment à Daniel Gaffé, qui a consacré de son temps pour m'aider sur Light Esterel, pour relire des papiers, et pour avoir été le professeur qui m'a initiée à la programmation. Je remercie aussi Tuan Dang, pour ses conseils, son éternelle bonne humeur et son aide sur le protocole OCARI.

Je n'aurais pas pu espérer avoir de meilleurs collègues, que ce soit à EDF R&D ou à mon laboratoire, le LEAT. Je les remercie tous de m'avoir soutenue pendant cette thèse, pendant les hauts et les bas, et je suis heureuse d'avoir pu partager avec eux de nombreux moments de convivialité, de débats enrichissants et d'activités diverses en dehors du travail. Ils ont permis de faire de ma période de thèse un excellent souvenir.

Je remercie EDF R&D et le LEAT de m'avoir accueillie dans leurs locaux pendant ces trois années, et d'avoir fait de sorte que ma thèse se déroule dans les meilleures conditions possibles. Enfin, je remercie ma famille et mes amis pour tout le soutien, l'encouragement et l'aide qu'ils m'ont apporté.

“Don’t stop because you’re tired. Keep going because you’re almost there.”

Ritu GHATOUREY

Résumé

Les réseaux de capteurs sans fil sont un domaine en plein essor qui montre un potentiel intéressant pour de nombreuses applications. Pour qu'ils soient plus facilement adoptés par les industriels, il est nécessaire de démontrer que le fonctionnement de ces réseaux est fiable, et par conséquent de valider les protocoles utilisés par ces nœuds pour communiquer. Différentes méthodes de validation peuvent être utilisées, mais nous montrons qu'à ce jour, aucune de ces méthodes ne s'est avérée être idéale. Nous avons donc développé un nouvel outil de validation pour ces protocoles, un environnement de simulation basé uniquement sur des codes en licence libre appelé SNOOPS (Sensor Network simulatiOn for the validatiOn of Protocol implementationS). Cet outil est capable d'exécuter le code binaire du protocole compilé sur un modèle de la plateforme hardware du nœud. Le nœud est modélisé en alliant l'émulateur de plateformes virtuelles QEMU au langage de description hardware SystemC, qui est utilisé dans ce cadre pour modéliser différents périphériques hardware du nœud ainsi que les communications en réseau entre les modèles de nœuds. Le principal attrait de SNOOPS est de posséder un module observateur qui a pour rôle d'arrêter la simulation si une propriété du protocole a été violée, afin de pouvoir trouver l'erreur qui est à l'origine de cette violation grâce à un debugger. Les propriétés du protocole à tester sont modélisées en Light Esterel, un langage réactif synchrone, à partir des spécifications du protocole. Elles sont ensuite compilées en C pour pouvoir être insérées plus simplement dans l'observateur. Un atout supplémentaire de SNOOPS est un module permettant d'interpréter et réinjecter en simulation des trames enregistrées dans un log en format pcap, provenant de tests avec des nœuds physiques pour lesquels l'origine de bugs n'a pas pu être déterminée. Nous avons testé sur SNOOPS le protocole OCARI développé par EDF R&D et ses partenaires industriels et académiques. Pour cela, nous avons développé un modèle de la plateforme ATMEL SAM3S et du module AT86RF233.

Abstract

Wireless sensor networks are a thriving area which shows good potential for many applications. So that they are more easily adopted by manufacturers, it is necessary to demonstrate that the operation of these networks is reliable, and therefore validate the protocols used by these nodes to communicate. Different validation methods can be used, but we show that to date, none of these methods has proved to be ideal. We have therefore developed a new validation tool for these protocols, a simulation environment based on free license codes called SNOOPS (Sensor Network simulatiOn for the valida-tiOn of Protocol implementationS). This tool is able to execute the binary code of the protocol compiled on a model of the node hardware platform. The node is modeled by combining virtual platforms QEMU emulator to the SystemC hardware description language, which is used in this context to model different devices hardware node as well as the communications network between the nodes. The main attraction of SNOOPS is its observer module who's role is to stop the simulation if a protocol property has been violated, in order to find the error that is at the origin of this violation through a debugger. The properties of the Protocol to test are modeled in Light Esterel, a synchronous reactive language, from the specifications of the Protocol. They are then compiled into C to be inserted more simply in the observer. An additional advantage of SNOOPS is a module to interpret and re-inject in simulation frames recorded in a log in pcap format, from tests with physical nodes for which the origin of bugs could not be determined. We tested on SNOOPS the OCARI protocol developed by EDF R&D and its industrial and academic partners. For this, we have developed a model of the ATMEL SAM3S platform and the AT86RF233 module.

Contents

Contents	ix
List of Figures	xi
Introduction	1
1 State of The Art in WSN protocol validation	7
1.1 WSN, a rapidly expanding technology	8
1.2 Communication protocols for WSN	13
1.3 State of the Art on WSN protocol validation methods	19
1.4 Conclusion	36
Bibliography	39
2 Developing the model of a node's hardware platform	49
2.1 Selected tools to model the hardware platform	51
2.2 Developing a platform model: The Atmel SAM3S	61
Bibliography	73
3 SNOOPS: Sensor Network simulatiOn for the validatiOn of Protocol implemen- tationS	77
3.1 What are the requirements for SNOOPS?	78
3.2 Developing a model of the network	80
3.3 Traffic generation	83
3.4 Modeling and Verifying protocol properties	90
3.5 Simulation output	99
3.6 Conclusion	102
Bibliography	102
4 Simulation results for some case studies	105
4.1 SNOOPS usage and benefits with OCARI	106
4.2 SNOOPS simulation time	119
4.3 Conclusion	123
Conclusion and Perspectives	125
A Publications	I
B The OCARI protocol	III
B.1 The MAC layer	IV
B.2 Format of the Macari frames	V
B.3 The NWK layer: OPERA	VII

C	AT86RF233 Radio state machines	XV
----------	---------------------------------------	-----------

List of Figures

1	Structure of the Thesis.	3
1.1	Block Diagram of a wireless sensor node.	8
1.2	Applications of WSN.	9
1.3	Applications of industrial WSN in a factory (source: [10]).	10
1.4	The evolution of the IoT market – infographic The Connectivist based on Cisco data [15].	12
1.5	WSN protocol stack model.	13
1.6	Development stages of a WSN system.	14
1.7	Layers of WSN protocols based on IEEE 802.15.4.	15
1.8	An example of MaCARI tree.	18
1.9	The global MaCARI cycle.	18
1.10	IoT-LAB (source: [55]).	24
1.11	Examples of SmartSantander nodes on a lamp post and on top of a bus in the city of Santander [9].	25
1.12	SmartSantander testbed physical network diagram [9].	25
1.13	Worldsens distributed simulation environment for Wireless Sensor Networks [84].	30
1.14	Interconnection of the virtual platform with SCNSL.	32
1.15	Concept of a hybrid simulator, associating physical and simulated nodes.	33
1.16	Different WSN system validation methods.	36
2.1	SystemC comparison with other hardware description languages [1].	51
2.2	SystemC Simulation Kernel [1].	52
2.3	TLM transaction	54
2.4	Tiny Code Generator	55
2.5	QEMU-SystemC block diagram [6].	57
2.6	QEMU and SystemC threads execution.	60
2.7	QEMU and SystemC synchronization through the TLMU interface.	61
2.8	Simplified block diagram of the SAM3S platform model.	61
2.9	Simplified block diagram of the SAM3S platform modeled with SystemC and QEMU through the TLMu interface.	62
2.10	Simplified description of an interrupt request transmission from a HW peripheral modeled in as a SystemC module to the CPU modeled by QEMU.	66
2.11	The two 32 bit set interrupt pending registers in QEMU.	67
2.12	Microcontroller to AT86RF233 Interface [20].	68
3.1	SNOOPS: A toolbox for WSN protocol verification and validation	78
3.2	Block diagram of different node models interconnected through the Radio Link module.	80

3.3	Top SystemC module in SNOOPS: instantiation of different nodes, the traffic generator, and connections to the radio module.	81
3.4	The radio link module.	82
3.5	Steps to reproduce a testbed scenario in SNOOPS.	85
3.6	Example of testbed scenario reproduced in SNOOPS.	86
3.7	Block diagram of the Frame Generator communicating with a simulated node.	87
3.8	Frame Generator's Finite State Machines.	88
3.9	The different stages used for property verification in SNOOPS.	91
3.10	Hello property modeled as a state machine.	93
3.11	Hello property modeled in LE.	94
3.12	Block diagram of the Clem toolbox [9].	96
3.13	Graphical user interface of Blif_simul testing the Hello property.	96
3.14	Block diagram representing the injection of the Hello property to check in the observer module of SNOOPS.	97
3.15	A section of a terminal output as SNOOPS is running.	99
3.16	Wireshark block diagram.	100
3.17	Wireshark used to analyze the information on a frame from the OCARI protocol.	101
4.1	Box representing Disassociation property with input and output signals.	106
4.2	Main loop of the program in Light Esterel representing the Disassociation property.	107
4.3	Topology of the nodes in the tested scenario.	108
4.4	Simulation traces with observer signaling that the property is verified.	108
4.5	Simulation traces with observer signaling that the property is violated.	109
4.6	Simulation output produced by SNOOPS and viewed with Wireshark showing a command frame (highlighted) sent with an unknown command identifier.	110
4.7	View of the terminal in which the remote debugger is launched - tracing the coordinator's stack's code to find out how the unknown command is interpreted.	111
4.8	Simulated topology: 7 nodes, all nodes are 1-hop neighbors except for the node with short address 6.	112
4.9	Simulation output viewed with Wireshark - 2 nodes pick the same color (color 1).	112
4.10	Simulation output viewed in Wireshark - highlighted Hello message from node 2 shows that node 5 is considered as a neighbor	113
4.11	View of the terminal in which the remote debugger is launched - tracing the stack's code to find the bug, due to node 5 being considered implicitly colored.	114
4.12	Output log viewed in Wireshark where we can see the Hello message (highlighted) with wrong priority list for 1-hop neighbors.	114
4.13	Simulation output viewed with Wireshark - Frame 837 is sent 99 microseconds after frame 836.	115
4.14	Testbed Setup	115
4.15	Topology of the network.	115
4.16	Comparing Association Request frames	116
4.17	Comparing Data frames	117
4.18	Simulation output frame traces from 2 regular OCARI protocol cycles viewed with Wireshark.	117

4.19 Network topology.	118
4.20 Simulation traces from the two last OCARI protocol cycles before the property violation viewed with Wireshark.	118
4.21 Normalized simulation execution time in relation to the global quantum for simulations with 1 and 2 emulated nodes.	120
4.22 Impact of the number of emulated nodes and the global quantum on the simulation's execution time.	121
4.23 Impact of enabling the Direct Memory Interface on the simulation's execution time.	122
4.24 Impact of the frequency of the SystemC clock fed to the platform models on the simulation's execution time.	123
 B.1 The OCARI protocol stack.	 IV
B.2 The OCARI global cycle.	IV
B.3 General format of MACARI frames.	VI
B.4 The OPERA module.	VIII
 C.1 Basic Operating Mode State Diagram of the AT86RF233 RF transceiver. . . .	 XV
C.2 Extended Operating Mode State Diagram of the AT86RF233 RF transceiver. .	XVI

Introduction

Context and research motivations

Wireless Sensor Networks (WSN) are a promising technology, which is employed in a wide and growing variety of applications. They are key components of the emerging Internet-of-Things (IoT) paradigm [1]. This expanding technology offers many advantages, such as helping the industry save operating costs and gain efficiency. Nevertheless, developing and deploying wireless sensor network protocols is a complex task. This is even more true when these protocols are aimed at applications that have strong constraints regarding fault tolerance, robustness and the reliability of the network, such as military or industrial applications. A lost packet, a missed alarm or a sensor blocked due to interrupts can potentially lead to devastating consequences. That is why all WSN protocols, especially the ones aimed at such constraining applications, have to be extensively tested and validated before their deployment in the field. However, the validation and verification of the functional properties and performance of WSN is nontrivial.

Protocol stack implementation is about concurrent (layers work in parallel) and real-time programming: timers, interrupt handling and task scheduling among layers. In a protocol implementation, the MAC layer is usually the piece of code that carries the most stringent timing requirements, as it is the layer responsible for reception (avoiding overloading and losing packets), and scheduling of time slots (time to transmit, time to listen, as well as sleep and wakeup time on the precision of which depend most part of energy-savings). State machines and their timeouts, the scheduling of sleep and wakeups, as well as time slots among a highly distributed and asynchronous systems, make the implementation and validation even more challenging tasks.

Many different validation approaches exist ranging from mathematical analysis, to simulation, to testbed analysis, each approach having its own advantages and drawbacks.

Validation heavily relies on human driven testing of the real platform and can take a very long time in the project. WSN simulation environments exist (Worldsens, NS-2, OMNet++...) and it is more and more necessary to test protocols using simulators first. However, none of these environments is really ideal to efficiently validate a protocol stack implementation.

It is in this context that the company EDF R&D, which has been the project leader in the development of a WSN protocol aimed at industrial applications (OCARI) [2], and the LEAT laboratory of the University Côte d'Azur have decided to start a collaboration, which is embodied by this thesis. OCARI is the protocol used as the experimental subject in this work, which is aimed at improving WSN protocol validation.

Research contributions

A new open source emulation framework

In this thesis, we propose a new simulation environment named SNOOPS (Sensor Network simulation for the validation of Protocol implementationS). It is aimed at validating WSN protocol implementations to assist in the final validation phase of a WSN system's development before its industrialization. It can also be used to validate a WSN system before adopting and deploying it. For example, a company could use SNOOPS to verify that a WSN system provided by a client complies with its specifications without needing the source codes of the protocol.

SNOOPS allows modeling high level node models as well as detailed sensor node models that can execute WSN protocols compiled code. The code that is tested on these detailed node models is the same as the one which will be deployed on the physical sensor nodes. However, the simulation environment offers the advantage of having a better visibility on the state of the the sensor node's different peripherals, as well as the possibility to trace the execution of the protocol without altering it (which may happen when using probes on physical nodes).

Although WSN simulators based on sensor node emulation exist, they only support a limited number of hardware platforms, and cannot be used to test protocols ported on other platforms. SNOOPS offers the advantage of flexibility on the type of hardware platform that can be modeled. To model the sensor node's hardware platform, we associated QEMU [3], an open source hardware virtualizer, and SystemC-TLM [4], a popular open source hardware description language. As QEMU provides a wide range of platform and processor models, and given the flexibility and modularity of SystemC models, the association of both simulators allows a good flexibility to create component libraries and model a wide range of hardware platforms. Therefore, SNOOPS can be adapted to a large number of hardware platform types and protocols. TLMu [5] is used to interface QEMU and SystemC, with SystemC being the simulation master. Using TLMu as a QEMU wrapper makes it possible to instantiate several node models based on QEMU.

We developed a model of the Atmel SAM3S microcontroller and the Atmel AT86RF233 radio transceiver using QEMU and SystemC co-simulation. This model is able to emulate the execution of a binary protocol stack. Several instances of the node model can communicate through a radio link module we developed in SystemC to handle a network topology. This module also creates a simulation log file in a format that is compatible with network analyzers, thus allowing the user to analyze more easily the frames generated by the simulated nodes.

Because it is based only on open source codes, SNOOPS can also be redistributed as open source.

Protocol Property validation during simulation

SNOOPS is capable of validating protocol properties during simulation to verify that the WSN protocol under test complies with its specifications. The protocol properties are modeled based on the protocol specifications in a synchronous language, Light Esterel [6]. It is possible to compile Light Esterel property models into C, which facilitates the insertion of properties to check into the simulation framework. These property models are inserted in a SystemC module called the observer, which analyzes the frame exchanges between simulated nodes and halts the simulation if a property is detected as violated.

The user can then locate the source of the bug that caused the property to be violated by exploiting the debugging capabilities provided by the simulation framework.

Repeating testbed scenarios in the simulation framework

SNOOPS offers the possibility to replay scenarios from log files recorded during experiments on physical testbeds. This feature can be used to repeat scenarios that led to the apparition of a transient bug, the source of which could not be determined. Reproducing those scenarios in simulation offers the user better debugging capabilities than on the testbed to locate the source of the bug more easily.

Thesis structure

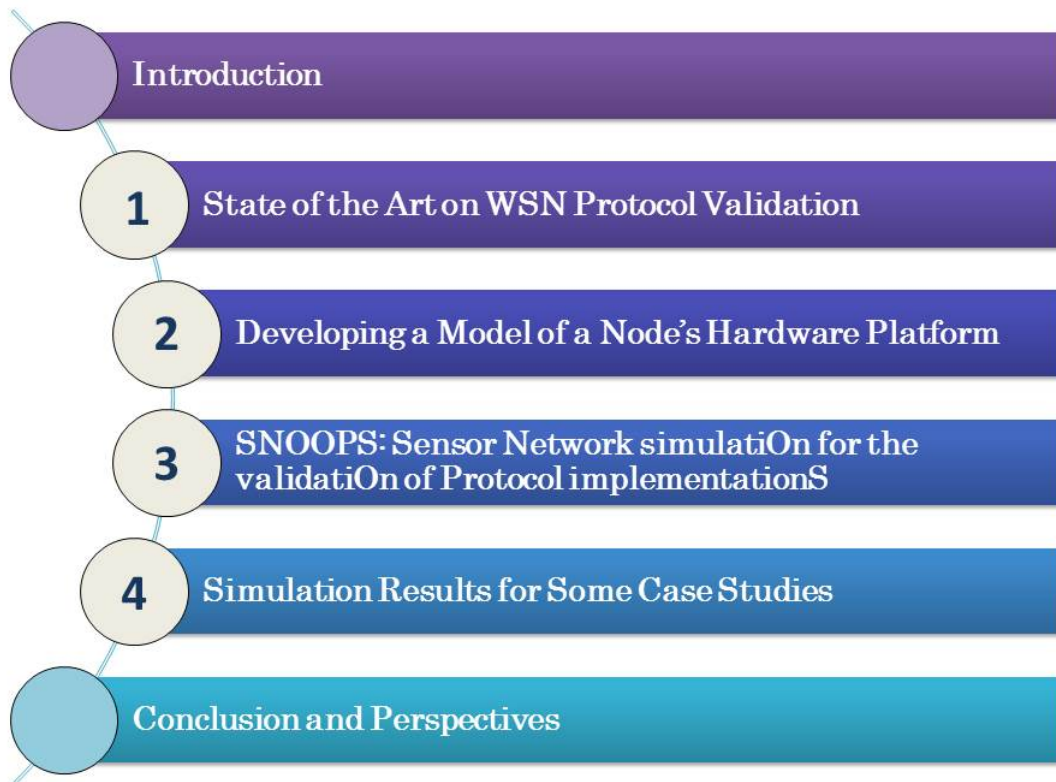


Figure 1: Structure of the Thesis.

The structure of the thesis is shown in Figure 1. After this introduction, in Chapter 1 we introduce wireless sensor networks, the constraints related to this technology and their predicted market growth. We also introduce the concept of wireless sensor network protocols and their development. Then, we expose the state of the art on WSN protocol validation methods.

Then, in Chapter 2, we describe the development of a sensor node's hardware model. This model is used to emulate the execution of the protocol stack's compiled binary code. We describe the simulation tools we used to develop the platform model, and the mod-

eling of the CPU and different hardware peripherals of the Atmel SAM3S and AT86RF233 RF transceiver.

In Chapter 3 we present the various features of SNOOPS, the WSN simulation environment we developed. We explain the network connection between the node models, the production of the simulation log file of frames exchanged in the network, as well as the mechanism to replay scenarios recorded from sensor node exchanges in a physical testbed. Finally we explain how protocol properties are modeled and inserted into the observer module of the simulation framework, and the observer's role of halting the simulation if a protocol property is violated.

Chapter 4 shows the different benefits of SNOOPS through several case studies. We also study the effect of various simulation parameters on its execution time.

We end this thesis with our conclusions and perspectives.

Bibliographies can be found at the end of each chapter with the references cited in that chapter. Appendices are found at the end of the manuscript. They include the list of publications done during the thesis, a part of the OCARI protocol's specifications, as well as the operating state diagrams of the AT86RF233 RF transceiver.

Bibliography

- [1] I. Khan, F. Belqasmi, R. Glitho, N. Crespi, M. Morrow, and P. Polakos, “Wireless sensor network virtualization: A survey,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 553–576, 2015. 1, 24
- [2] K. Al Agha, M.-H. Bertin, T. Dang, A. Guitton, P. Minet, T. Val, and J.-B. Viollet, “Which wireless technology for industrial wireless sensor networks? the development of ocari technology,” *IEEE Transactions on Industrial Electronics*, vol. 56, no. 10, pp. 4266–4278, 2009. 1, 17
- [3] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, 2005. 2, 55
- [4] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the ground up*, vol. 71. Springer Science & Business Media, 2011. xi, 2, 31, 51, 52
- [5] “Transaction level eMulator (TLMu),” 2011. edgarigl.github.io/tlmui/. Accessed on Sep. 27, 2016. 2, 58
- [6] D. Gaffé and A. Ressouche, “Algebraic framework for synchronous language semantics,” in *Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on*, (Birmingham, UK), pp. 51–58, IEEE, July 1-3 2013. 2, 92, 95

Chapter 1

State of The Art in WSN protocol validation

Contents

1.1 WSN, a rapidly expanding technology	8
1.1.1 What is a WSN?	8
1.1.2 Applications	9
1.1.3 Industrial needs and constraints	10
1.1.4 Predicted Growth and evolution of WSN	12
1.2 Communication protocols for WSN	13
1.2.1 What is a protocol?	13
1.2.2 Development stages of a protocol	14
1.2.3 An overview of protocols for Industrial WSN	14
1.2.4 OCARI	17
1.3 State of the Art on WSN protocol validation methods	19
1.3.1 What is a verification method?	20
1.3.2 The use of formal methods to validate protocols	20
1.3.3 Testbeds	22
1.3.4 WSN Simulation and Emulation	26
1.3.5 Issues with existing solutions to reliably validate a protocol imple- mentation	34
1.4 Conclusion	36
Bibliography	39

1.1 WSN, a rapidly expanding technology

1.1.1 What is a WSN?

Wireless Sensor Networks are distributed embedded systems that collect data on their environment (temperature, pressure, humidity, etc.) and relay it autonomously through their network to an end user. The network is generally composed of a sink node or gateway and a certain number of other sensors that are capable of collecting data and relaying it through a multi-hop architecture to the sink node. The sink node transmits the collected data to a task manager that will analyze the data and take decisions accordingly.

As seen in Figure 1.1, the main elements composing a wireless sensor node are:

- The microcontroller which executes the communication protocol's program
- An RF transceiver controlled by the microcontroller to communicate with other nodes
- At least one sensor to collect environmental data
- A battery which may be recharged by some ambient energy (solar, mechanical, etc.) collected from the environment of certain nodes.

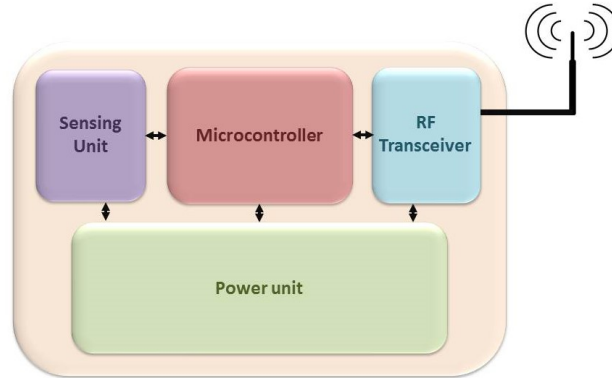


Figure 1.1: Block Diagram of a wireless sensor node.

The development of WSN systems is a complex task for several reasons. The sensor nodes are small devices that are constrained in terms of energy resources, memory, data rate and processing capabilities [1]. Additionally, WSN are geared to serve a number of needs and constraints depending on the intended applications, sometimes contradictory.

The network's reliability [2] is the ability of the network to carry out its functional requirement of relaying information over the network over a broad range of operational conditions. A reliable network is able to address communication disruptions, unanticipated variations in traffic and variations in the operational environment using a predictable "fail-safe" mechanism. Moreover, WSN are often energy-constrained, and a reliable WSN should maintain operational integrity during tough energy saving situations. These issues that must be addressed complicate even further the development of a WSN product.

1.1.2 Applications

The large emergence of WSN was originally motivated by military applications. The first wireless network resembling what we today call WSN was developed by the United States military in the 1950s to monitor Soviet submarines using acoustic sensors [3]. In 1980, the United States Defense Advanced Research Projects Agency developed a program called the Distributed sensor networks [4]. Eventually governments and universities began taking an interest in WSN for applications such as air quality monitoring, forest fire detection and weather stations. At the same time, WSN made their way into industrial applications such as waste water treatment or factory automation.



Figure 1.2: Applications of WSN.

The decrease in size and cost of micro-sensors, the widening range of sensor variety as well as the development of wireless communications have widened the number and type of applications based on WSN technology.

Nowadays, WSN can be found in applications such as:

- **Environmental monitoring:** WSN can be useful to gain better knowledge on our environment, and to signal problems as when they are used to monitor forest fires, air quality, seismic activity and radioactivity levels. WSN can also be used to observe wildlife whilst limiting human intrusion. For example, ZebraNet is an application to monitor zebra in their natural environment for biologists to understand their mobility patterns [5].
- **Agriculture:** WSN can be used to know when irrigation or the use of fertilizer and pesticides is needed for a more efficient crop management. WSN can also be used in agriculture to detect the risk of frost and plant disease. The collected data can serve to determine which are the best conditions for each crops by analyzing data obtained during the best harvests [6].
- **Healthcare:** WSN can be used for long term surveillance of chronically ill patients or the elderly. An example of use of WSN for healthcare is the application for a capsule endoscope, monitoring internal organs by transmitting images and video from inside the body to other wireless devices outside the body [7].

- **Industry:** Using WSN in industrial environments has many advantages. For example, they allow getting rid of the constraint and weight of cables in submarines or planes, they can also be used for machine and equipment surveillance for predictive maintenance. In Figure 1.3, we can see the different applications for which WSN can be used in a factory (mobile field inspection, inventory movement, etc.). It is worth noting that WSN can also include actuators, which are nodes that can act on their environment [8]. In the industry, this technology can be used for example to control valves regulating a water flow in a tank, according to sensed data on the tank's water level.
- **Smart cities:** WSN can be used to monitor traffic, to understand the flow and congestion of traffic for more efficient road systems, or to adjust street lights for passing cars during the night (e.g. Santander, Spain [9]).



Figure 1.3: Applications of industrial WSN in a factory (source: [10]).

1.1.3 Industrial needs and constraints

In this section, We focus on the constraints of industrial applications, given that the industry is a sector where WSN are developing rapidly and given that industrial protocols will be the focus of our experimentations.

The industrial sector is one of the domains for which reliability and Quality of Service (QoS) of WSN are primordial. The QoS provided by WSN refers to the correctness of received data by the sink node of the network. Sensor information is usually data for which the reception delay is of sensitive nature, as for example alarm notification on industrial sites. Data received with a long latency due to processing time or traffic congestion may be obsolete and lead to wrong decisions in a management system. In industrial environments, the lack of reliability of control systems can have serious consequences, such as injuries, explosions, material losses or environmental pollution. An uninterrupted production is of capital importance for some industry players, often operating with narrow profit margins and for whom an interruption would have disastrous consequences on profits [11].

For this reason, before investing in new wireless systems and equipment, industrialists require an operational reliability demonstration in realistic industrial conditions.

Indeed, industrial environments can be particularly hostile for WSN communications. These harsh operating conditions can be of various nature [11]:

- Very high or very low temperatures (some companies would like to deploy WSN in environments where temperatures would rise to 1420°C).
- High humidity levels without condensation (95%)
- Potentially explosive situations
- Mobile or fixed metallic equipment affecting transmissions (noise, signal overload, distortions, inter-modulations)
- Vibrations
- Dust
- Highly caustic or corrosive environments
- Interference from other instrumentation also operating on the same frequencies, such as those operating in the ISM band using WiFi for example [12]

Let us cite some of the functional needs and constraints that are expected for industrial WSN and the implementation complexity resulting thereof:

- **Fault tolerant links:** a protocol must detect and recover from the loss of packets (ensured by the MAC layer),
- **Tolerance of RF medium transients,** using mechanisms such as adaptive routing of packets in a mesh network (NETWORK layer),
- **Deterministic behaviors:** ensuring at least a predictable access and bound end-to-end delays (these are regulatory requirements like in the EN5425 standard for wireless fire alarms; all layers concerned),
- A **good "link budget":** ratio of emitting power to sensitivity of receiver (determined by PHY layer).
- **Power saving techniques:** a battery lifetime of up to 10 years can be expected for some applications (all layers concerned),
- Addressing a potentially **large number of nodes** (determined by the addressing scheme at MAC and NETWORK layers).

For many industrial applications, the requirement on data rate (few bits/s to 30kb/s typically) is not stringent. These are commonly referred to as Low Throughput Networks (LTN).

All these constraints make the QoS more difficult to guarantee. A validation of WSN systems taking into account these environmental conditions is essential for industrialists to deploy such networks with confidence.

1.1.4 Predicted Growth and evolution of WSN

The global revenue of the WSN market has been showing a strong growth in the last years, and this growth will only be stronger in the future years as WSN integrate into IoT. Indeed, through the development of sensor, semiconductor and wireless networking technologies and communication protocols, WSN have the potential of becoming the backbone of the IoT. Associating WSN with cloud computing and big data processing allows a limitless potential for applications. It is this great range of potential applications that hint that in the near future WSN will surely become an integral part of our everyday lives [13].

In a statistical report from Dr. Harrop dating 2012 [13], it is showed that the WSN market annual revenue was increasing steadily at a rate around 30%, the rate of 2012 even up 40% from the previous year reaching nearly 600 million US dollars. Dr. Harrop predicted that this rapid increase will continue and will reach 2 billion US dollars by 2020.

In another report from 2014 [14], IDTechEx research has found that the WSN market will grow to 1.8 billion US dollars by 2024.

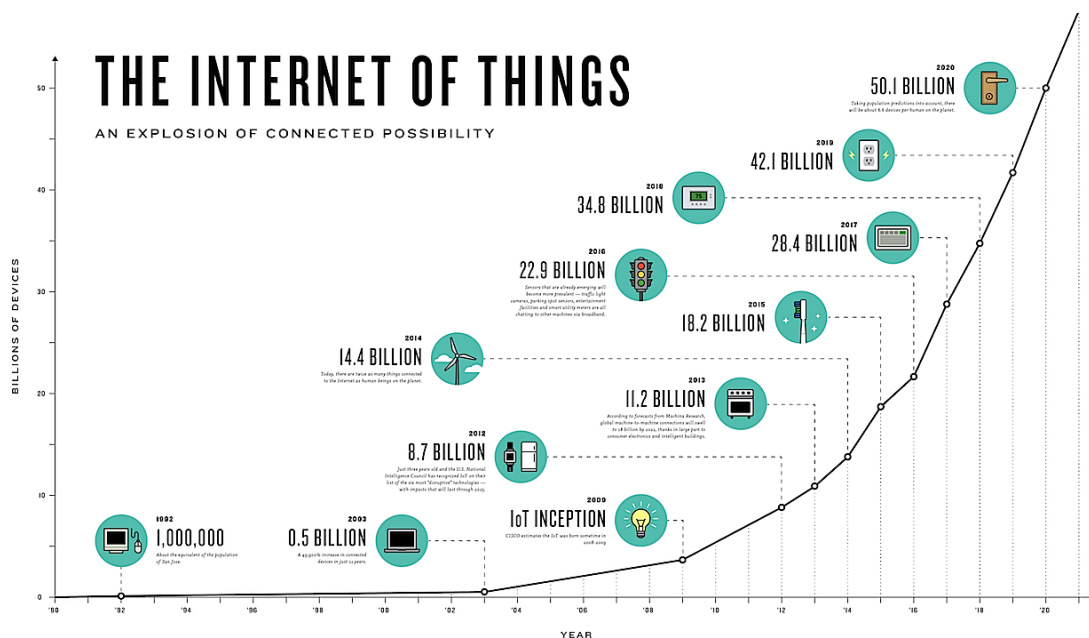


Figure 1.4: The evolution of the IoT market – infographic The Connectivist based on Cisco data [15].

The IoT market, in which WSN should become essential, is also predicted to grow rapidly as illustrated in Figure 1.4. According to research by IndustryARC released in 2016, the Industrial Internet of Things market is predicted to grow to \$123.89 Billion by 2021. It will be the Industrial Internet of Things market segment that will be generating the highest revenue in the overall IoT market in the forecast period. However, the highest growth rate is that of another industry: the IIoT (Industrial IoT) segment for healthcare or medical devices is expected to grow at a CAGR (Compound Annual Growth Rate) of 59.8 percent [16].

These increasing numbers show that WSN and IoT are promising areas where there will be a lot of research and development needs in the future, from both the industry and academia, in order to develop new applications with better standards and technologies. Indeed, the WSN business is set to become a multibillion dollar activity, but this will only happen if there is major progress with standards and technology. Currently, it is the USA

industry that is leading the development of WSN partly due to the heavier funding available, and the interest of the military as well as large industrial companies such as IBM or Microsoft [14].

Some potential synergies of WSN with other areas than IoT have been researched [17]. They would also contribute to the development of the WSN market. For example a synergy between mobile robots equipped with GPS and WSN could be useful for search and rescue operations [18]. In this case, the WSN would help robots react to events outside their perception range. Another example is the possible synergy between WSN and RFID, for applications that would benefit from combining the sensing capabilities of WSN with the identifying capabilities of RFID. Such an application could be food monitoring, where RFID is used to identify, categorize and manage the flow of goods, and can give information on temperature and raise alarms [19]. However, the tools to read RFID tags (RFID loggers) have to be handled manually and have a short reading range. For this application, WSN could provide a longer reading range and a flexible topology. This synergy could improve good monitoring.

1.2 Communication protocols for WSN

To conceive a WSN capable of automatically handling a number of diverse tasks, the conception of an efficient communication protocol plays an important role.

1.2.1 What is a protocol?

A communication protocol is a set of rules that govern the communication of distributed entities through the exchange of messages. A protocol must define the precise format of valid messages, the procedure rules for the exchange of data, as well as a vocabulary for valid messages (data units of the protocol) and their meaning.

A model of the protocol stack for WSN can be seen in Figure 1.5. This stack is based on a simplification of the OSI model that has 7 protocol layers.

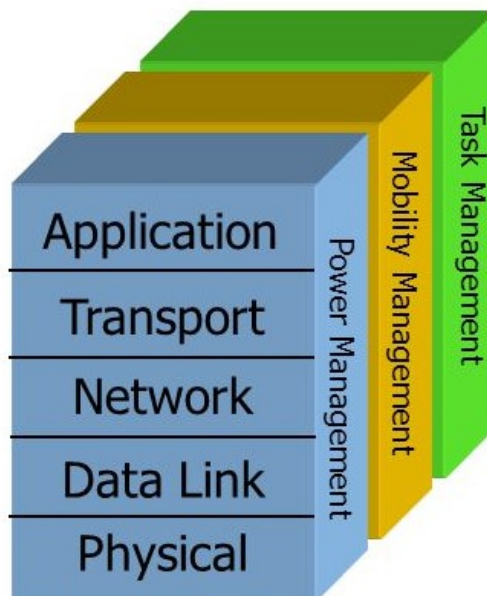


Figure 1.5: WSN protocol stack model.

The WSN protocol stack possesses 5 layers, and 3 management planes.

The application layer (APP) provides an interface to receive data from the user and to execute different type of applications. The transport layer can be used to handle the data flow. The Network layer (NET) implements the routing mechanisms. The Data Link layer (MAC) contains the MAC protocols to handle the access to the radio medium. These protocols are used to avoid collisions, to handle packet corruption, to minimize transmission delays and increase transmission reliability. The Physical layer (PHY) offers services such as modulation, transmission, encryption and decryption of data.

The Power Management plane handles how the sensor uses its power, for example by setting the sensor to sleep mode when it is not needed. The Mobility Management plane is used to detect and register the movements of the different sensor nodes in the network. The Task Management plane balances and schedules the sensing tasks.

1.2.2 Development stages of a protocol

Figure 1.6, presents the typical life cycle of a WSN product development [20]. This product's development begins with its inception (**theory**), where the different product specifications and the product's theoretical behavior are developed. Then comes a **simulation** stage, where a behavioral model of the system is simulated to check the correct functionality of the product and figure out if some behaviors have not yet been defined. When the behavioral model of the WSN product is validated through simulation, it is possible to implement it, meaning to develop the software that will be deployed on the hardware node (**Local Prototyping**). The next stage is generally testing the software on WSN **testbeds** in a lab to validate the behavior of the protocol once implemented and flashed on a node under diverse environmental conditions. The **early stage product** is obtained once the product's behavior has been mostly validated.



Figure 1.6: Development stages of a WSN system.

1.2.3 An overview of protocols for Industrial WSN

WSN protocols have been designed for different applications, such as industrial surveillance, health or the smart home. Each application has different needs and constraints for the communication system. Some require very precise data to be communicated between nodes while others require a long autonomy.

Many solutions have been developed for communication protocols adapted to WSN needs. Indeed, in [21], the authors identified over 70 MAC protocols for WSN, which were classified into four families:

- **Scheduled Protocols** for periodical high density traffic, generally based on TDMA (Time Division Multiple Access) combined with FDMA (Frequency Division Multiple Access) (e.g. TSMP [22], IEEE 802.15.4 [23])
- **Protocols with Common Active Period** for which nodes define common periods of activity and sleep, needing a precise synchronization (e.g. TMAC [24])

- **Preamble Sampling Protocols** for which each node can choose its activity period independently from the others. Nodes sleep most of the time and wake up shortly to check if a frame is being transmitted. Preambles are long to ensure that all potential receivers detect and receive the frame (e.g. Preamble sampling CSMA [25])
- **Hybrid protocols** which are combinations of previously mentioned protocols to adapt to different needs of traffic flow (e.g. ZMAC [26])

Despite the many existing MAC protocols in literature, few have been implemented and standardized. In the following sections, we review some of the main WSN protocols aimed at industrial applications.

1.2.3.1 IEEE 802.15.4

The IEEE 802.15.4 standard [23] is meant for applications that only necessitate an infrequent exchange of small packets, and for which low energy consumption is essential. This standard defines the PHY and MAC layers. Many popular protocols for WSN are based on this IEEE standard. The specifications of the PHY layer require that the radio operates in one of the three ISM (Industrial, Scientific, Medical) frequency bands. To regulate the channel access, the MAC layer uses the CSMA-CA technique (Carrier Sense Multiple Access with Collision Avoidance) which lacks determinism.

The main issue with the IEEE 802.15.4 protocol is that it lacks reliability in its MAC layer in terms of packet loss. Indeed, in [27], the authors show that when the power manager is used to reduce energy consumption, the packet loss rate reaches up to 80% in a 50 node network.

Alternative WSN standards are built on top of the PHY and MAC layers defined by IEEE 802.15.4, some of them also bring modifications to those layers, as seen in Figure 1.7.

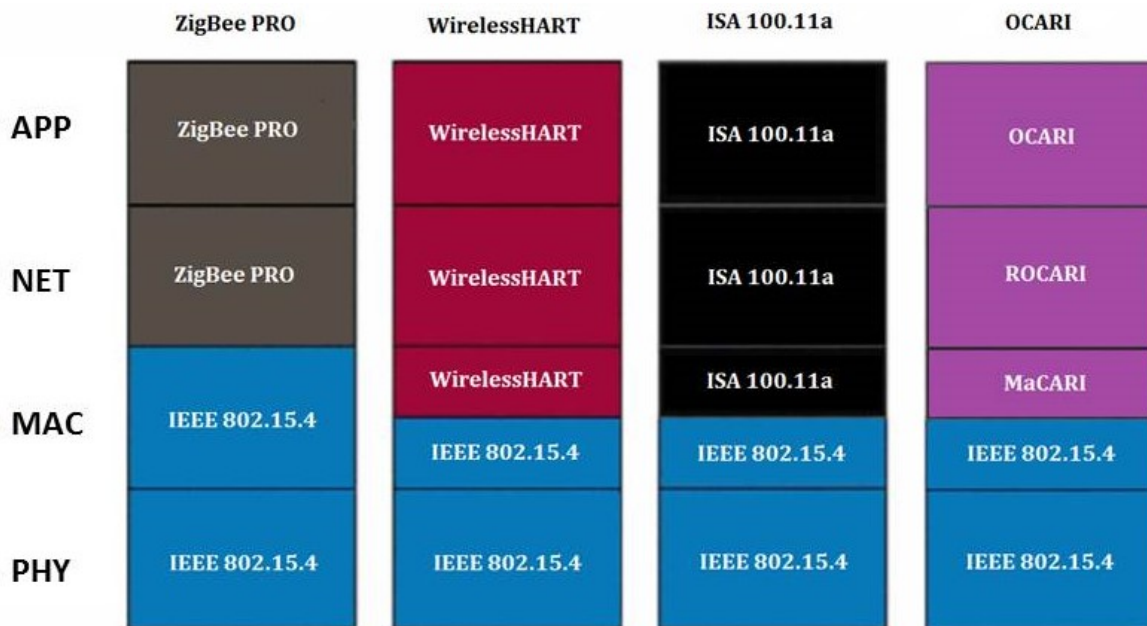


Figure 1.7: Layers of WSN protocols based on IEEE 802.15.4.

1.2.3.2 ZigBee

Zigbee [28] is a low cost and low power standardized protocol based on the IEEE 802.15.4 MAC and PHY layer specifications. It provides specifications on the NET and APP layers to provide a complete protocol stack. The 2008 Zigbee specifications [29] defines two protocol stacks: Zigbee for light applications such as the smart home and Zigbee PRO, a more robust protocol aimed at industrial control applications. The performances of Zigbee PRO are more reliable than those of Zigbee but require a more complex protocol stack. Despite Zigbee having found its place in smart home applications, this standard is seldom used for industrial applications, even in its more robust form, Zigbee PRO. This might be to the unreliable and nondeterministic nature of the CSMA-CA medium access technique which is unsuitable to more critical industrial applications.

This is why more reliable WSN standards for hostile RF environments have emerged.

1.2.3.3 WirelessHart

WirelessHART [30] is specified in the HART Field Communication Protocol, a control protocol established since the mid 80s in the area of industrial automation. The 7th revision of this protocol extends it to wireless communications by basing itself on the IEEE 802.15.4 PHY and MAC layers.

WirelessHART was conceived to allow communication between wireless sensor nodes in hostile industrial environments, where wireless communications must have a reliability close to that of wired communications [31]. The main difference with the IEEE 802.15.4 MAC layer is that communications between nodes are coordinated through the TDMA technology (Time Division Multiple Access) instead of CSMA-CA, which eliminates the non deterministic aspect of the channel access attempt and the random back-off used by CSMA-CA. WirelessHART uses the FHSS method (Frequency Hopping Spread Spectrum) to emit through 16 alternative channels defined in the IEEE 802.15.4 standard. It is possible with this protocol to forbid the use of certain channels which are then "blacklisted", to avoid WirelessHART interfering with other wireless systems that may have real time constraints.

1.2.3.4 ISA100

WirelessHART was the first wireless standard on the industrial automation market. Following its introduction, ISA100 [32] was created by the International Society of Automation (ISA) to develop a standard for a larger variety of industrial control networks than those of covered by WirelessHART [33]. The objective of ISA100 is to provide a unique infrastructure of integrated wireless systems for the industrial sector thanks to a family of standards defining wireless systems for industrial automation and control applications.

The first standard of the ISA100 family was ISA100.11a. It shares a lot of aspects of WirelessHART, including the TDMA technique for medium access. However, the ISA protocol provides a greater number of options for industrial WSN. It reintroduces the MAC protocol based on contention as in 802.15.4 or ZigBee in addition to the MAC protocol based on TDMA to get a higher data flow when needed. Although the CSMA-CA option suffers from the same shortcomings as ZigBee in hostile RF environments, but allows a higher data rate in more familiar environments.

ISA100 protocols have reserved some headers for future versions to add revisions or new functionalities to the standards. ISA100 also allows channel hopping to avoid interference with other RF systems, and can use adaptive channel hopping to detect busy

channels or channels with low performance. The network layer of this standard uses header formats from the 6LoWPAN (IPv6 over Low power Wireless Personal Area Networks) standard [34] allowing using 6LoWPAN networks as a base.

1.2.4 OCARI

The OCARI protocol (Optimization of Communication for Ad hoc Reliable Industrial networks) [35] is a protocol based on the PHY and the MAC layer of the IEEE 802.15.4 standard. However, the MAC layer is modified in order for OCARI to have a more deterministic behavior. It satisfies the following criteria in harsh environments:

- A deterministic MAC layer for time constraint applications
- An optimized energy consumption routing strategy to maximize the network's lifetime
- Some Support for walking speed mobility for certain nodes

This project was born in 2006 and was funded by the French National Research Agency. It was led by the French company EDF and included several partners, both industrial and academic.

The industrial applications targeted by OCARI are:

- Real time centralized supervision of personal dose in nuclear power plants
- Condition-based maintenance of mechanical and electrical components in power plants
- Environmental monitoring in and around power plant
- Structure monitoring of hydroelectric dam
- Machine and equipment surveillance for predictive maintenance
- Mobile instrumentation for test and measurement during outage periods
- Fire detection

OCARI was developed based on the TELIT ZigBee Pro stack [36]. While the PHY layer was preserved, the MAC layer was modified to make it more deterministic and less power consuming. OCARI's MAC layer is called MaCARI. It guarantees through the network tree a default route for constrained traffic and offers a deterministic access to the medium by scheduling activities which is why synchronization is a key aspect of OCARI. MaCARI also reduces power consumption by allowing all entities to fall asleep and wake up in time.

1.2.4.1 The MaCARI cycle

There are two types of nodes using OCARI: coordinator nodes (full function devices), and end device nodes that can be either fixed or mobile nodes (reduced function devices which cannot establish a network). The MaCARI tree is created by association (father-son). If a node no longer receives frames from its father, it tries to re-associate with a new father. This automatic repair supports micro-mobility. An example of MaCARI tree can be found in Figure 1.8

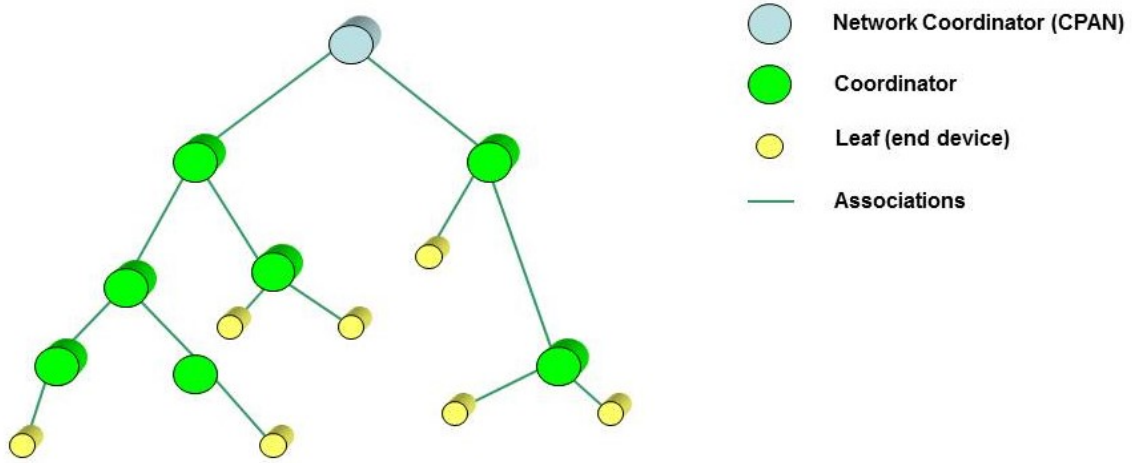


Figure 1.8: An example of MaCARI tree.

The MaCARI cycle is segmented into three periods as seen in Figure 1.9. The first period is a synchronization period so that all nodes share the same notion on time. The second period is the activity period where data is exchanged between nodes (inner-star harvesting and routing between coordinators) and the last period is the inactivity period where nodes sleep to save energy.

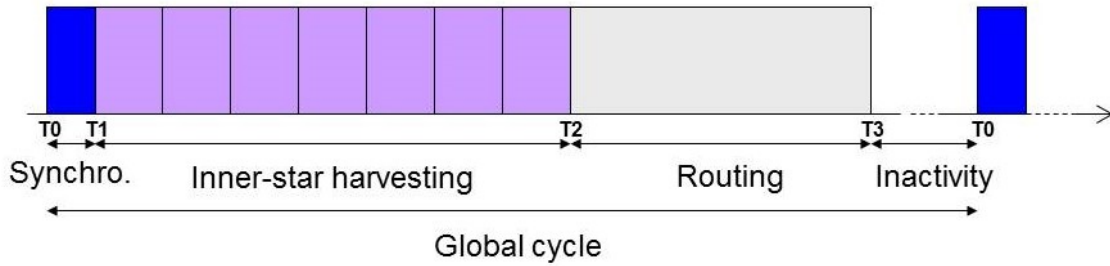


Figure 1.9: The global MaCARI cycle.

During the synchronization period, the network coordinator (CPAN) schedules activities and imposes the broadcast of a sequence of beacons: it broadcasts the beacon to his sons and the same beacon is repeated as a waterfall effect down the tree. The beacon is a frame containing all the synchronization information so that all the nodes synchronize on T1 and know when the next T0 is.

During the inner-star harvesting period, each star has its own time slot to exchange data. A star is composed of a group of end devices and their local coordinator node (father). The end devices send their sensed data to their coordinator, then a time slot is dedicated to sending back this data up the MaCARI tree from the local coordinator to its father.

The routing period is used to send data up the tree from coordinator to coordinator all the way up to the CPAN. During this period the leaf nodes can sleep. The medium access method is different between the deployment phase and once the network is stabilized.

During the deployment phase, the access method is CSMA-CA. Exchanges between coordinators are managed by OPERA via MaCARI. OPERA, which stands for Optimized Energy efficient Routing and node Activity scheduling for wireless sensor networks, includes EOLSR (Energy-efficient Optimized Link State Routing), an energy efficient routing protocol and OSERENA (SchEdule RoutEr Nodes Activity), a coloring algorithm op-

timized for dense wireless networks. Once the network is stabilized, each node picks a color so that they may be no interference between two nodes of the same color. To do so, two nodes of the same color must be separated by at least 3 hops. The $[T_2; T_3]$ period is used to distribute a succession of colour slots, and the access method becomes TDMA. Nodes can sleep during colour slots that are neither theirs or one of their neighbours'.

During the inactivity period, all nodes can sleep. The length of this period depends on the application and the energy constraints.

OCARI is currently in its industrial deployment phase. It runs on the Dresden Elektronik deRFsam3-23T09-3/23M09-3 platform (Atmel SAM3S and Atmel AT86RF233), as well as on the Adwave Adwrf24-LRS platform (Atmel SAM3S and Atmel AT86RF233 coupled with a LNA chip). We have focused more particularly on the description of this protocol as it is the protocol under test in our experimentation. The interested reader can find more information on the OCARI protocol specifications in Appendix B.

1.3 State of the Art on WSN protocol validation methods

Data transmission networks are not always reliable. At different levels of the network, different type of bugs may occur:

- At the **hardware** level due to for example:
 - Interrupts not taken in time
 - Hardware failures
 - De-synchronization due to clock drifts
- At the **software** level due to problems such as:
 - Mistake in specifications and/or their interpretation
 - Programming mistakes (e.g. deadlocks)
 - Buffer overflows
 - Compiler errors
- At the **transmission** level due to for example:
 - Packet transmission errors that would cause data to be lost
 - A too important traffic flow
 - Interference

However, there are some types of networks that are more reliable in which packets can be duplicated, packet routing can be dynamic and sequences can be altered. The need to guarantee a reliable network providing a minimal quality of service (QoS), robustness and security is the reason why it is important to validate protocol implementations.

There are various methods to validate a WSN system. Validation of the WSN product should be done at different points during its development, and the method that is best adapted depends on the development stage of the product. Moreover, communication protocol validation has been done so far by using different methods in a complementary way. In this section, we review different validation and verification methods and consider their various advantages and disadvantages.

1.3.1 What is a verification method?

The verification of a software system consists in determining the design and implementation errors that may have been introduced during different software development phases [37]. To obtain a good reliability of a software product, its development should begin with a formal specification. The system's verification is then based on this specification. However, to our knowledge, to this date no methods has proven to be ideal for the elimination of all the errors in a WSN system.

Verification is used to make sure that a system respects a set of properties.

In the case of a communication protocol verification, properties that must be verified are those related to the protocol itself and its QoS. Regarding properties related to the protocol, it must be verified that there are no deadlocks (process or thread entering a never ending waiting state because a requested system resource is held by another waiting process), non specified receptions or dead code. Regarding properties related to the QoS, it must be verified that the service provided by the protocol is the expected service, for example that the delay to transmit a packet across the network does not exceed a certain time (latency).

The verification techniques that can be applied to WSN are static analysis using formal methods, as well as tests using simulators, emulators and testbeds, which we will review in the following sections.

1.3.2 The use of formal methods to validate protocols

Experimentations on prototypes suffer from material limitations as well as limitations in terms of the range of scenarios that can be experimented. On the other hand, simulation does not allow an exhaustive analysis of all the system's possible executions. The certainty that a property will be verified in all the execution cases is consequently impossible. The use of formal methods seems essential for systems needing a high level of reliability.

A major issue for the validation of wireless communication protocols is that it is difficult to represent concepts related to WSN (concurrency, communications, timing, probabilities...) in a formal language. Moreover, formal validation methods are not always adapted to WSN systems [38].

1.3.2.1 Formal Verification

To formally verify a system, one must model the system and its interactions in order to prove a set of properties on the model by applying formal methods. Formal methods are computer techniques based on mathematical logic that allow proving in a rigorous way that a system complies with a set of properties [39].

The system and its properties are modeled in a mathematical language that allows establishing if the properties are verified. To formally verify a system, one needs [40]:

- A **verification method** that allows establishing the proof of the system's correct functionality.
- A **formal modeling language** for the system.
- A **tool** that allows modeling the system under study in the chosen formal language in order to implement the verification method, and if necessary a test generation method.

Formal Verification Methods

Numerous formal verification techniques have been proposed. The two main methods are Model checking and Network Calculus.

Model Checking [41] [38] is an automatic formal verification method for dynamic systems. This method aims at verifying through algorithms if a system satisfies a specification, that is generally expressed as temporal logic. The system is usually modeled as a finite State Machine (FSM) representing its specifications. From this model of states and transitions, a semantic interpretation is generated representing all the possible behaviors of the system [41]. Algorithms allow an exhaustive exploration of the possible states of the system. The types of properties verified during this exploration are:

- Accessibility (There is an execution path to each given state)
- Safety (An undesirable event never happens)
- Deadlock (A state has no exit transition)

For example, for a real time protocol it can be checked that no state for which a packet's delay is superior to a given deadline is accessible.

The main limitation of model checking is combinatorial explosion when the number of possible states is too big. It is then impossible to explore all of them in a reasonable time.

Network calculus [42] [40] is a theoretical environment that allows analyzing the performance of a communication network. It gives strict bounds on the system's performance, for example on hard real time constraints or QoS. Communications are represented as flows going through the network. The constraints of these communications are defined by mathematical functions called arrival curves or service curves. The calculus of performance bounds is done for a given topology, and it is not possible to represent the dynamism of the topology (node or radio links disappearing). Moreover, modeling communication protocols as service curves is based on unverified hypothesis, for example that nodes provide a minimum service.

According to the PhD research done in [40], Model Checking is a better verification method in the case of WSN. The reason is that Network Calculus was specifically developed to study worst case end to end delays in networks, but while it can easily be used to represent a TDMA protocol, it might not be the case for large scale distributed protocols. Model Checking has the advantage of being an automatic process and provides counterexamples that are very useful in the development process of a protocol. However, the combinatory explosion of the number of states remains a limiting factor to the use of Model Checking.

Formal modeling languages

Specification languages or formal description techniques are developed to insure the clarity, completeness, consistency and traceability of specifications.

When using the model checking verification method [41], a system's model must describe the behavioral rules of the system. These rules must be described using a formal modeling language that must be capable, for a given application, to model the various aspects of a WSN system. The model's interpretation is represented as a state chart in which each state is a recording of values of the system variables. The transition between states depends on the behavioral rules of the protocol.

When modeling a WSN system, the main aspects that must be represented are:

- Concurrency (nodes working simultaneously)
- Communication protocols
- Exchanged data
- Probabilities (unreliable radio medium, Bit Error Rate, etc.)
- Time

The following languages are some of those that can be used to model WSN.

Petri nets [43] [44] is a graphical formalism that models concurrency, so it is adapted to distributed systems. The system's behavior is described by the changing positions of tokens in the Petri network, depending on the initial positions and the crossing of transitions. A precise representation of time is made possible by a temporal extension of Petri Nets [45], where time constraints are added to cross transitions. There is also a stochastic temporal version of Petri networks that can represent probabilities [46]. Unfortunately, to our knowledge and to date, there isn't any tool that is able to check Petri networks that model both time and probabilities [40].

Process Algebra [47] [40] is a framework used to model the behavior of distributed systems and concurrent processes through mathematical expressions. The system is represented in the form of a process term, using the basic operators, the communication operators, and recursion. The process terms represent the states in this labeled transition system model, and transitions correspond to actions. The resulting process term is manipulated by means of equational logic, to prove that its graph conforms with the desired external behaviour [47]. This framework can be used to detect undesirable properties and to formally derive desirable properties of a system specification. There are no model checking tools for Process Algebra.

Automata [40] are machines with inputs, outputs and internal states. Outputs depend on the inputs and the internal states. There are some Timed Automata (TA) [48] that allow representing real time systems' behaviors using an explicit representation of time. However, it is more difficult to represent communications and concurrency with timed automata because the synchronization of several elements of the system would have to be encoded. Because the number of possible combination between the actions of the different elements increases in a quadratic way with the number of elements, this encoding becomes quickly complicated and long as the system's size increases. Consequently, to represent concurrency and communications, Networks of Timed Automata (NTA) have been introduced [49]. A NTA is composed on several TA working in parallel and synchronizing over common actions. NTA enable modeling time, concurrency, as well as communications through synchronization. It is also possible to add probabilistic transitions to timed automata which then become Probabilistic Timed Automata. There are model checking tools that allow verifying TA, such as UPPAAL [50], PRISM [51] or Fortuna [52].

1.3.3 Testbeds

Various aspects of distributed applications such as the radio communication over a shared medium can be complex to simulate. Consequently, the results of the evaluation of a WSN system through simulations or theory can only be considered approximate. Even excellent channel models have to be confirmed by real world measurements. Due to this

fact, real world experiments have increased in popularity. Experimentation on real sensor nodes is usually done in so called testbeds. [53]

A WSN testbed is a platform for the experimentation of development projects [20]. It allows testing the software implementation of the full protocol stack on a set of hardware nodes in a controlled environment [1]. The packets exchanged between nodes can be traced using a sniffer.

WSN testbeds enable more realistic and reliable experimentations than on WSN simulators when it comes to capturing the subtleties of the underlying hardware, software and dynamics of the WSN. The deployment of those WSN testbeds is increasing rapidly, a development that is also due to the increasing collaboration between industry and academia.

WSN testbeds allow experimenting on different aspects of WSN systems conception, such as the protocol stack, resource management and network optimization.

However, the development and testing of WSN systems on real platforms can quickly become tedious if the number of nodes exceeds a few dozens [54], for the following reasons:

- Sensor nodes are generally small systems with very limited memories and debugging capacities (limited memory space to add code dedicated to debugging)
- Software deployment and debugging require a connection to each device, thus individual manipulations of each node
- Sensor nodes are generally powered by batteries with a limited lifetime, so they might have to be changed often.

Large scale testbeds and testbed federations have been developed to make it possible to run scenarios more easily on a larger number of nodes.

1.3.3.1 Federated WSN testbeds (WSN-FT)

WSN-FT are created by interconnecting through the internet several independent testbeds that are locally managed. This allows carrying out larger scale and more complex experiments as well as access to a larger variety of hardware nodes.

IoT-LAB

IoT-Lab was originally called Senslab. It gives access to more than 2700 sensor nodes spread over 6 locations in France [54]. The nodes are set on a fixed location, however, some of them can be mounted on robots as seen in Figure 1.10 to experiment with mobile nodes. The available processor architectures are MSP430, STM32, Cortex-M3 and Cortex-A8. 96 nodes spread over 2 locations are said to be open host. The tests using the IoT-LAB can be programmed and followed via a web interface. IoT-LAB provide a secure connection (ssh) that enables the user to start, stop, reset, update the nodes, and read or write on the serial links. One master server manages several clients (each testbed site) and to handle concurrent reservations, the method is first-come-first-served.

WISEBED

WISEBED [56] was created as a common effort between 9 research institutes across Europe. The goal was to federate different networks in Europe in order to establish a heterogeneous European WSN with several thousand nodes. The federation of these testbeds is based on the virtualization of testbeds and virtual links between them. WISEBED is



Figure 1.10: IoT-LAB (source: [55]).

composed of 750 motes, mainly iSense, MicaZ, Pacemate, SunSPOT, and TelosB motes. The motes are equipped with a wide range of sensors ranging from most commonly used temperature sensors to more sophisticated Anisotropic Magnetoresistance (AMR) sensors. The deployed resources are accessible to users via a web interface. WISEBED uses its own generic XML-based language called WiseML, for experiment and testbed description, configuration, and results storage while some other federated testbeds use databases. To handle the concurrent reservations, WISEBED uses a first-come-first-serve approach, meaning that the first user submitting an experiment on available resources gets the access.

SmartSantander

The largest federation in terms of number of nodes is SmartSantander [9], providing a city wide IoT experimentation platform with around 20000 nodes that are spread over 4 European cities [57]: Belgrade in Serbia, Guildford in the UK, Lubeck in Germany and Santander in Spain. Some of these testbeds are also accessible through WISEBED and are part of its federation.

The SmartSantander framework builds upon WISEBED and its underlying capabilities, but extends them for the use in a larger scale outdoor environment by adding support for wireless reprogramming of nodes, facilitating the selection of adequate experimentation resources, as well as lower configuration overhead for the management of testbed nodes.

The main advantage of SmartSantander is that it provides access to nodes that are embedded in real urban infrastructure as shown in Figure 1.11 for more realistic mobility experiments. Some nodes are embedded into public transportation vehicles and are remotely accessible for experimentation. They can be used to determine the location of the vehicles, estimate arrival time to bus stops, or make atmospheric measurements [58].

The connectivity in the SmartSantander WSN testbed is achieved through different technologies as shown in Figure 1.12. Fixed nodes are organized into different clusters and form a mesh network. All the data collected by the sensors as well as the testbed and



Figure 1.11: Examples of SmartSantander nodes on a lamp post and on top of a bus in the city of Santander [9].

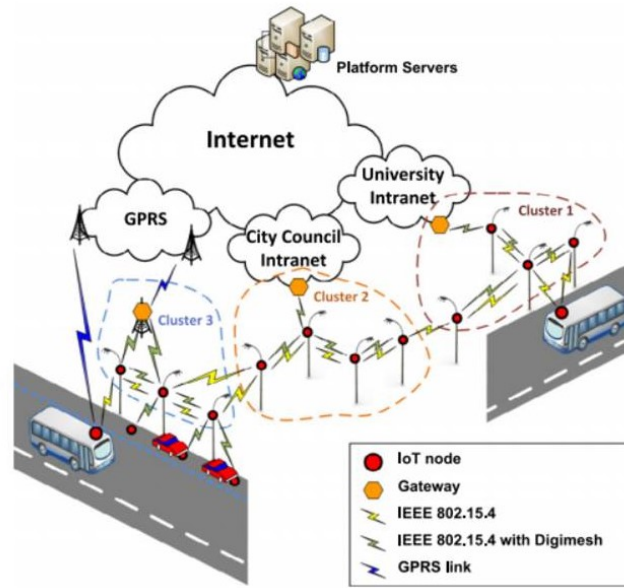


Figure 1.12: SmartSantander testbed physical network diagram [9].

experiment management traffic are forwarded through the cluster head (gateway node).

1.3.3.2 Limitations of WSN testbeds

While it is necessary to validate protocols on testbeds before their deployment, this validation method is not sufficient to guarantee a reliable WSN system.

The reproducibility of an experiment refers to the ability of the testbed to reproduce similar results when using identical inputs and the same environmental conditions. An experiment's reproducibility has the advantage of offering the possibility to evaluate the performance of WSN applications, tune their parameters and compare protocols [20]. The issue is that experimental testing of event driven WSN applications is highly unrepeatable due to the fact that the state of such a system depends on the sequence of events and their timing. Moreover, channel conditions such as fading, obstacles or interference may vary from one experiment to another. Debugging probes that are used on testbeds can be intrusive and cause Heisenbugs [59], which are intermittent bugs that may appear or disappear when one attempts to probe or isolate them. This can happen because the investigation of a failure can affect the process scheduling, causing a scheduling related failure not to occur again.

Generally, WSN testbeds have a tendency to be limited in size (number of nodes, test-

ing area), their maintenance is expensive and they take a long time to set up [60]. Some also lack flexibility given that most nodes are fixed and the type of available platforms can be limited. This is why testbed federations is important, but some progress still needs to be made. Users may lack control on a number of factors influencing the test environment such as local interference due to infrastructure or other experiments being carried out at the same time.

1.3.4 WSN Simulation and Emulation

Another approach to WSN protocol study is simulation. WSN simulators can be used for testing, evaluation and initial validation of a protocol stack [20]. There are two main types of simulators used for evaluation of WSNs [53]:

- **General purpose simulators:** The most popular simulators are platform independent as the sensor nodes' hardware is abstracted. In these types of simulator, the user must implement the application to be tested through an API provided by the simulator. However, this implementation is not suited for deployment on a WSN hardware platform.
- **WSN specific simulators:** Some simulation environments support WSN specific hardware by cross compiling the WSN code for a platform where the simulation is supposed to run. Some simulators also support native platform code or native binary images by emulating the hardware of WSN platforms, and thus are even able to interpret the binary code compiled for a supported WSN platform.

While simulators are a less costly solution than WSN testbeds, their fidelity to reproduce channel conditions as well as the conditions of the protocol's execution can be a concern. The fidelity of a simulator depends on the accuracy of its underlying model of the system. However, the more accurate the model will be, the more complex and slow to execute it will be.

In order to represent a system, a model serving as approximation or representation of certain aspects of the architecture, the behavior, some functions or characteristics of the system must be created [61]. This model can be used to represent :

- The system under development
- The environment in which this system operates

For this purpose, a large number of WSN simulators have been developed, and it is often complex to choose the most appropriate simulator for a given application.

The key requirements for these simulators are:

- The **accuracy** of the system and the network's behavior, as an imprecise simulation can lead to erroneous results
- The **scale** of the network that can contain hundreds or even thousands of nodes. The simulation's execution time and the necessary memory must not increase too much with the number of simulated nodes
- **Modeling the energy consumption** can be an important feature to evaluate networks where nodes have a limited autonomy

- The **flexibility** of the simulator is important to be able to easily modify existing modules or add new ones
- Compatibility with a possible **heterogeneity** of the nodes of the network
- An available **graphical user interface** can ease configuration and the visualization of the simulation results

An important issue to bring to light is also that existing simulators are unable to model many essential characteristics of real systems [1]. Additionally, due to the degradation in the conduct of simulation studies, some simulation results may be questionable and lack credibility [62].

If the final protocol validation is generally done using testbeds, software validation on a virtual platform, which is a software based system mirroring the functionality of a System-on-Chip, allows detecting and analyzing a great number of problems that would be difficult to reproduce or diagnose directly on a physical hardware platform. Moreover, simulation offers a better visibility of the system than hardware testbeds and enables collecting a lot of information related to code performance, to the hardware platform when it is modeled and to the protocol itself.

1.3.4.1 Using general purpose WSN Simulators to validate protocols at a behavioral level

General purpose simulators are useful to quickly try out new ideas and to investigate the behavior of new protocols and mechanisms in varied topologies at large scale and in a repeatable manner. There is a broad variety of this type of simulation tools, which have been designed to explore and validate WSN systems before actual implementation and real-world deployment [63].

However, these simulators are rather dedicated to model the network and are not an adequate solution to model the node architecture since they are usually incapable of modeling the concurrency within the node and provide a direct path to HW/SW synthesis [64]. Another drawback of general purpose WSN simulators is a lack of fidelity of the environmental model with reality. Simulation alone is therefore of limited use to plan for real-world WSN systems and deployments [60].

NS-2

NS-2 (Network Simulator 2) is the second version of a network simulation tool developed by the Virtual InterNetwork Testbed (VINT) project [65]. It is one of the most popular general-purpose open-source network simulators [63]. The core of the simulator is in C++, with Object-Tcl(OTcl) based scripting. It is based on a discrete event engine and has an extensible object oriented architecture that allows users to add custom components and libraries. Moreover, many communication protocols and simulation models are provided by NS-2. To support WSN simulations, NS-2 includes ad-hoc and WSN specific protocols such as directed diffusion [66] or SMAC [67].

However, NS-2's installation is complicated and time consuming and the user must learn a scripting language to use it, as well as queuing theory and modeling techniques [68]. Moreover, NS-2 does not scale well in terms of memory usage and simulation time [64].

NS-3

NS-3 (Network Simulator 3) is the most recent version of the Network Simulator (NS) family that is intended to replace NS-2. While NS-3 is still in the active development process [63], it is nevertheless a powerful open source tool for network modeling and optimization. NS-3 is meant to improve the architecture, software integration, models and educational components of NS-2 [69]. NS-3 has a modular object oriented architecture and a discrete event simulation engine just like NS-2. However, NS-3 does not run NS-2 scripts as they are written in OTcl, instead NS-3 uses C++ or Python scripts. While many communication protocols are included in NS-3 (TCP/IP, IPv6, IEEE 802.11, etc.), the only one of interest for WSN applications (at this time) is the LR-WPAN IEEE 802.15.4 protocol, although other protocols may be included in future releases of the simulator. NS-3 offers extensive online support and documentation, which makes it relatively easy to use for inexperienced users.

OMNet++

OMNet++ [70] is a discrete event network simulator based on C++ that has an extensible and modular component based architecture. It is distributed under the Academic Public License, meaning that it is free for non profit use. Like NS-2, OMNet++ is extensible and actively maintained by its research community [71]. This simulation library provides a deep analysis of network activities at the packet layer [63]. The OMNet++ package includes an Eclipse based Integrated Development Environment which contains a graphical user interface that facilitates configuring, debugging and analyzing the simulation results. NED, which stands for Network Description, is a high level declarative language. It is used in OMNet++ to define topologies and assemble simulation models into larger components to represent a greater system. OMNet++ offers extensive documentation.

There are several OMNet++-based WSNs simulation frameworks such as:

- **Castalia** is a Wireless Sensor Network simulator for early-phase algorithm/protocol testing. It supports realistic channel and radio models, enhanced modeling of the sensing devices and nodes' clock drift and implements MAC and routing protocols.
- The **MiXiM** framework (mixed simulation) supports node mobility, models for walls or other obstacles, various network layers and protocols including IEEE 802.15.4 and T-MAC.
- **NesCT** allows to run TinyOS applications in the OMNet++ environment by translating NesC code (an event-driven, component-based programming language implemented on TinyOS) into C++ classes.
- The **PAWiS** (Power Aware Wireless Sensors) framework can be used to simulate concurrently the power consumption of several nodes.

It is also possible to integrate SystemC models into OMNet++, but only with its commercial version OMNEST.

OPNET

OPNET (Optimized Network Engineering Tools) [72] is also a discrete event object oriented simulator used to model and simulate data networks. Its system is defined by a hierarchical model. The top level of this hierarchy is the network model, the second level

is the node level where data flow models are defined. The next level is the process editor, and the last level is the parameter editor. The network is simulated at packet level. OPNET was originally conceived to analyze fixed networks, but has been extended to mobile and satellite networks. For this purpose, OPNET includes libraries containing models for the equipment and protocols for many of the best known communication technologies [73]. In the area of WSN, OPNET facilitates the simulation of ZigBee based networks by providing several ZigBee components, such as the ZigBee coordinator, router and end device.

However, OPNET is only distributed under a commercial license.

1.3.4.2 Using WSN Emulators or simulators compatible with emulation to validate protocols at a binary level

While simulation uses abstract models of the target system, emulation replicates the systems' functionality. Emulation is therefore capable of much greater fidelity than simulation while potentially offering greater flexibility than a physical testbed. However, emulation is a much less exploited approach despite much potential [60].

Atemu

Atemu (ATmel EMUlator) [74] is a software emulator written in C for systems based on Atmel's AVR microcontroller family (e.g. MICA2 [75]), with an ISS (Instruction Set Simulator) at its core. ATEMU provides a GUI, called Xatdb, which provides users a complete system for debugging and monitoring the execution of their code [64]. It was the first WSN simulator to provide instruction level precision of each individual sensor node. This is achieved by using a cycle-by-cycle implementation strategy where each node and each device are advanced by one clock cycle every round to ensure that nodes, their internal devices, and the radio communication are correctly synchronized.

This cycle-by-cycle implementation strategy achieves good synchronization but scales poorly, as each device adds work to be done each clock cycle [76]. Moreover, this emulator is limited to AVR microcontroller based platforms.

Tossim

Tossim [77] is a discrete event simulator for sensor networks based on TinyOS [78] nodes included in the TinyOs framework. It provides a high degree of accuracy by modeling a few low-level components, while running the software source code, as well as a high level of scalability and execution speed for networks with a large number of sensor nodes [63].

However, the Tossim approach is not able to capture the fine-grain timing and interrupt properties of the code [79]. Another limitation of TOSSIM is that it is only able to run the same program on each node of the network. Moreover, TOSSIM is not extensible and only the MicaZ hardware platform is supported [63].

Avrora

Avrora [76] is a set of simulation and analysis tools for programs written for the AVR microcontroller produced by Atmel and the Mica2 sensor nodes. This instruction level simulator written in java scales better than Atemu and approaches the performance of Tossim, while preserving cycle accuracy.

However, AVRORA solely supports AVR MCU cores and does not provide any extensions for other CPU architectures. The only radio transceiver that is modeled is the CC1100 from Texas Instruments. This severely hinders AVRORA's flexibility [63]. Moreover, AVRORA does not implement a clock manager, making it impossible for the CPU to modify its frequency and to model the clock drift phenomenon [80].

COOJA/MSPSIM

COOJA [81] is a flexible Java-based simulator for simulating networks of sensors running the Contiki operating system [82]. COOJA's three main features are a GUI based on Java's standard Swing toolkit to parameter and analyze simulation results, the simulation of the radio medium and an extensible framework, which allows integration of external tools to provide additional features to the Cooja application. Current versions of Cooja work with two different emulators: Avrora for the emulation of Atmel AVR-based devices, and MSPSim for emulation of TI MSP430-based devices [83].

MSPSim is a Java-based instruction level emulator of the Texas Instruments MSP430 microcontroller series. It combines cycle accurate interpretation of CPU instructions with discrete event simulation of all other components. Together, COOJA/MSPSim can simulate sensor nodes at various levels of details, which can co-exist and interact in the same simulation.

While COOJA/MSPSIM is the best choice for the development of Contiki-based applications running on the MSP430 hardware architecture and applications running on AVR hardware platforms, there is no support provided for other hardware platforms and target OS [63].

Worldsens

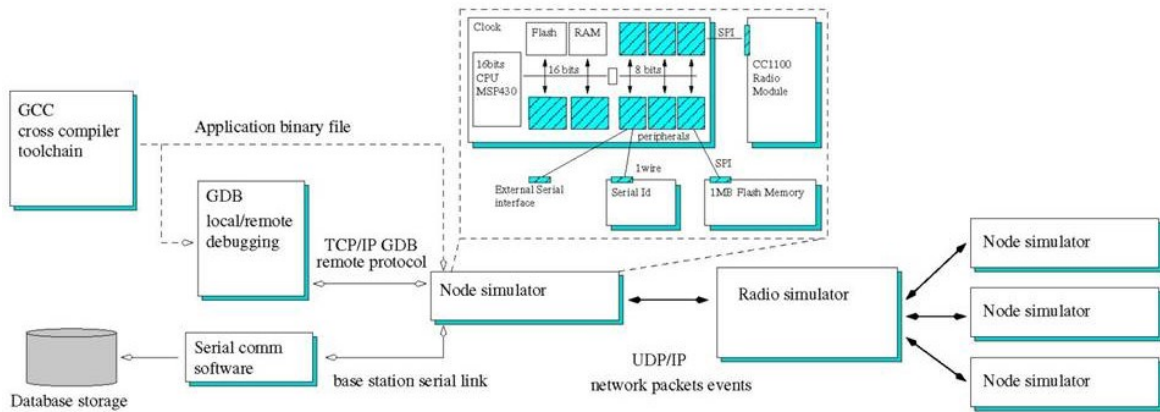


Figure 1.13: Worldsens distributed simulation environment for Wireless Sensor Networks [84].

The Worldsens framework [79], which offers an integrated platform for the design, development, optimization and deployment of WSN applications, is represented in Figure 1.13. As for the Cooja/MSPSim simulator, the Worldsens development framework consists in two simulators that can be used either independently or together during different stages of a WSN product development: WSim and WSNNet.

WSim is a cycle accurate hardware platform emulator that can model nodes based on the MSP430 and ATmega128 processors, and uses instruction cycles as a timing reference. It allows users to precisely analyze the application's execution.

WSNet is an event-driven wireless network simulator written in C++ that allows evaluating and validating high level conception choices. WSNet is known as having an accurate channel model. It offers numerous radio medium models, some exploiting the Friis propagation formula, Rayleigh fading, multiple frequencies, complex antenna radiation pattern, etc. Moreover, WSNet can simulate some physical phenomena such as fires [63].

When both WSim and WSNet are used together, they can simulate the behavior of an entire WSN with great precision. Both simulators communicate through sockets and synchronize on rendez-vous points with the help of a save and backtrack mechanism.

WSNet does not offer a GUI environment to interpret simulation results, which might be considered a disadvantage of the WSNet toolkit [63]. The hardware platforms that are supported by WSim are limited to those based on MSP430 and ATmega128 processors. Unfortunately, the Worldsens simulation framework is no longer updated. WSim's last commit on github is dated January 4th 2012. There has been no kind of activity reported on WSNet's project website since September 2013. The link to www.worldsens.net, which is provided in [84] had its domain name registration expire. However, the codes are still available for download.

SCNSL

The SystemC Network Simulation Library (SCNSL) [85] is an extension of SystemC [86], which is a popular class library within C++, providing a common language to model both hardware and software, so it can be used to design both the architecture and the behavior of a hardware component. SCNSL can model packet-based networks such as wireless networks, ethernet, fieldbus, etc. As done by basic SystemC for signals on the bus, SCNSL provides primitives to model packet transmission, reception, contention on the channel and wireless path loss. The use of SCNSL allows relying on a single simulation environment (i.e., SystemC) to model hardware and communication aspects of networked embedded systems.

The availability of a SystemC extension for network modeling allows simulating not only the HW/SW of a network node but also the communication environment in which the node operates. For instance, in [87] and [88], a complete WSN scenario has been simulated by connecting the μ CSim ISS with SCNSL.

Figure 1.14 shows how to interconnect a virtual platform with SCNSL. A WSN scenario consists of several nodes and, for one of them, it is possible to analyze the detailed simulation of interactions among SW, HW and networks, while the other nodes are modeled at behavioral level. The dashed box contains the detailed node model. CPU, bus, and HW peripherals are simulated by the traditional virtual platform (implemented in SystemC, QEMU, OVP, or even a commercial one) [89]. Software is compiled with the CPU-specific toolchain and the corresponding binary is loaded into the model of the memory. The core of the interaction is the thick edge box in Figure 1.14. It can represent either a radio-frequency interface directly attached to the bus or an external RF modem connected to the system-on-chip through a serial interface (e.g., SPI, I2C, UART). The component denoted as “virtual device” is a special peripheral acting as a bridge between the VP and SCNSL. From one side it is connected to the bus and is seen by the CPU as a set of memory-mapped registers; from the other side it interacts with an instance of the SCNSL task which is able to send/receive packets on the network. From the SCNSL perspective, each task instance is connected to the hosting node by using standard TLM sockets. All the nodes are bound to the model of the radio channel which simulates the mutual effect of all packet transmissions.

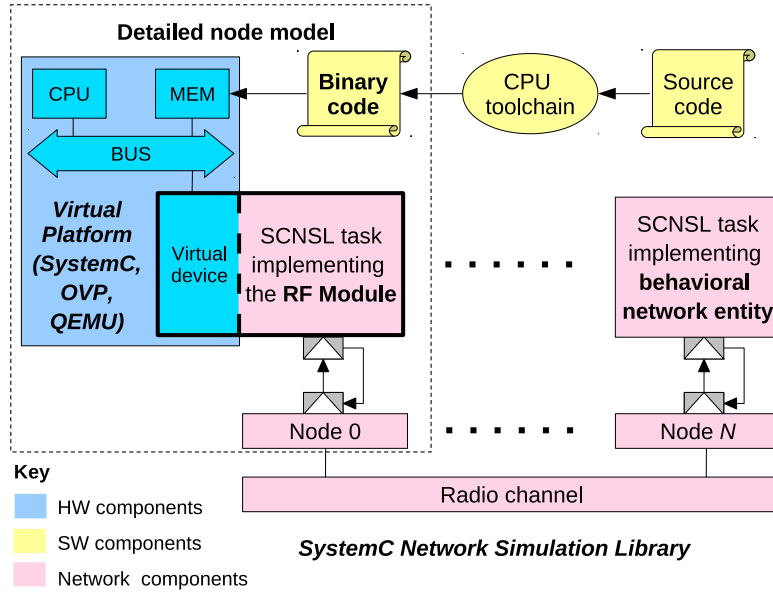


Figure 1.14: Interconnection of the virtual platform with SCNSL.

However, the only protocol modeled in SCNSL is the IEEE 802.15.4 beaconless mode.

IDEA1 [90] is a WSN simulator that is based on SCNSL. Like SCNSL it models propagation delay, interferences, collisions and path loss, but IDEA1 also models the energy consumption of the node hardware platforms. The energy consumption of each component is calculated by recording state transitions of the microcontroller and transceiver and knowing the given current consumption of each of these states. Several processors and transceivers have been modeled in IDEA1 as finite state machines [91], including the AT-MEL ATmega128, Microchip PIC16LF88, TI CC2420, TI CC1000 and Microchip MRF24J40, which are basic components of some motes such as MICA2 and MICAz. A graphical user interface allows the user to configure the network and set the sensor nodes characteristics.

An improved version of IDEA1 is called **IDEA1TLM** [92]. This simulation framework can model the software aspect of the nodes using a custom-built instruction set simulator called BLISS. The nodes are modeled as finite state machines. This simulation framework allows users to evaluate which circuits and which configurations are the most appropriate for a given application.

1.3.4.3 Hybrid Simulators

Hybrid WSN Simulators, as depicted in Figure 1.15, are WSN simulators that communicate with at least one real sensor node, that may be used to produce data packets or to benefit from the wireless communication links [93]. The coexistence of real and simulated nodes implicates a thorough coordination, the simulation's execution time must mirror the real execution time in order to preserve causality [94].

Emstar

EmStar [95] is a framework targeted at more powerful nodes called Microservers such as Crossbow's Stargate platform, but can be used for less powerful motes as well [93]. This framework enables studying the system's fault tolerance and its reaction to fault isolation

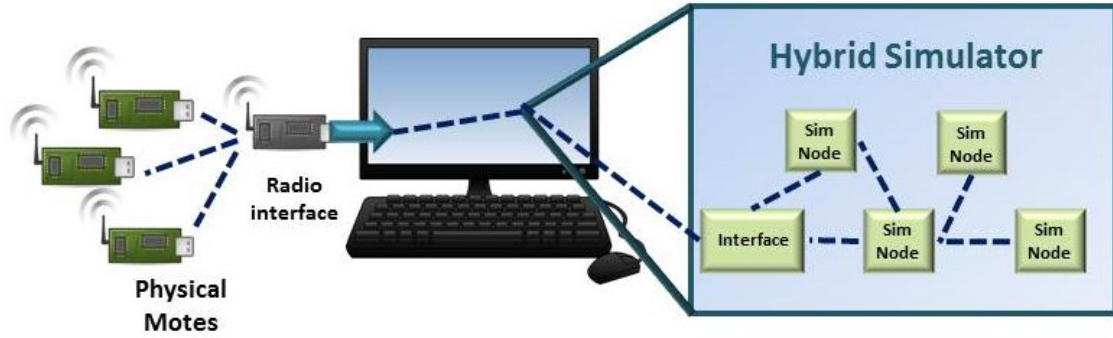


Figure 1.15: Concept of a hybrid simulator, associating physical and simulated nodes.

(failure in one software component such as memory corruption). It also enables system visibility, in-field debugging and resource sharing across multiple applications.

Emstar is a multi-process system that consists of libraries made to implement message passing ICP (Network interprocess communication) primitives, tools supporting simulation, emulation and visualization of live systems, as well as services supporting networking, sensing and time synchronization [95].

To run hybrid simulations, Emstar provides an interface to a real radio for communication between simulated and real nodes. However, a physical wire is required to connect each physical node which limits the size and geographical area of the network to be debugged.

Emstar operates exclusively at the application layer, which means that it cannot directly control hardware or process interrupts with low latency.

ATOSSIM

ATOSSIM [96] is a hybrid simulation framework based on TOSSIM. While in TOSSIM all nodes have to run the same code, different codes can run on nodes in ATOSSIM. Such nodes are called shadow nodes, and their main role is to act as bridge between simulated nodes and real nodes. They collect sensed data from real nodes and reroute packets from virtual nodes to real ones. An application gateway handles the coordination between real, shadow and virtual nodes, so that simulation timing be constrained to satisfy at least soft real-time guarantees. These guarantees may be met if the event handler processing time is low on average, and if simultaneous execution of simulated events does not frequently occur. However, simulation execution time may even be double than real time [97].

According to [93], simulation results show a timing error that increases with the number of nodes, and becomes noticeable when more than 50 nodes are simulated. So ATOSSIM's results may be unreliable for large networks.

H-TOSSIM

H-TOSSIM [98] is a hybrid testbed that extends TOSSIM with physical nodes. H-TOSSIM needs at least three physical nodes, one of which shares the simulated environment with virtual nodes, and the other two bridge the real world to the virtual one. Using two physical nodes as bridge allow sending concurrent messages from the virtual to the real world and to match the rate of the radio (up to 250 Kbps) and the computer's serial link (at most 115 Kbps) [93]. As for the synchronization of the virtual and real time, H-TOSSIM checks

if the virtual time is faster than the real time. If it is the case, H-TOSSIM goes to sleep until the real time catches up with the simulation time. In the other case, H-TOSSIM executes immediately [93].

However, like TOSSIM, H-TOSSIM is limited to TinyOS based nodes.

SensorSim

SensorSim [99] is a hybrid simulator that is an extension of the NS-2 simulator. SensorSim provides a new GUI and can interact with physical nodes. To enable communication between real and simulated sensor nodes, SensorSim provides a simulated protocol stack for each real node. These protocol stacks are connected to their corresponding real node through a proxy and a gateway mechanism. These stacks represent the real nodes in the simulated network topology by providing the PHY and MAC layers, and sometimes the routing layer as well. The gateway server handles the packet format conversion between NS-2 and real nodes, and routes packets towards their destination. When a packet event arrives from a real node to the simulator, the real time delay is adjusted and the event must be placed in the correct order in the event queue.

A synchronization approach is said to be considered for SensorSim in [99] between real and simulated nodes consisting in pausing the simulation and reorganizing events inside the simulator scheduler to ensure the correct order of execution of events. However, the authors of [93] declare that the simulation scheduler and real nodes are not synchronized, meaning that real nodes cannot interact with virtual nodes at the correct time.

1.3.5 Issues with existing solutions to reliably validate a protocol implementation

As we have shown, there are many methods and tools that have been developed to validate wireless sensor networks. Yet, none of these tools and methods can be used on its own for a reliable validation and verification of a WSN system. Here are the main reasons for that.

General purpose simulators have limitations mainly due to the fact that they do not model the hardware platform of the node. Consequently, while they are very useful to test a protocol at a behavioral level, a protocol's implementation can't be validated on those type of simulators. Moreover, they have trouble capturing real world dynamics.

While **formal methods** are known for reliable validations, they become very complex to implement on WSN systems when concurrency, time, statistics and a large network with a high number of state combinations have to be modeled. Moreover, they are better suited to model the behavior of a protocol than its actual implementation and the interaction of software and node hardware, which is a source of bugs.

Emulators allow testing the protocol implementation's binary code on a model of the hardware platform of the node. There are less existing emulation frameworks for WSN than general purpose simulators. The main limitation of existing emulation frameworks is the limited type of hardware platforms that are supported. For example, Atemu is limited to MICA-2 motes. If one wanted to model ARM based nodes, none of the existing emulators could support it. Moreover, emulators are not as scalable as general purpose simulators because they model the systems with more precision.

Testbeds are used to test a protocol's implementation on physical nodes in real environmental conditions. However, it can be difficult to control the environmental conditions of the test (transient obstacles causing reflections or link losses, interference, etc.),

which may render some tests non reproducible in the same exact conditions. There is less visibility in testbeds than on simulators where the code's execution can be traced more easily. Moreover, testbeds can be limited in the number of scenarios that can be tested (number of nodes, topology, mobility, geographical testing area, environmental conditions, interference, etc.).

1.4 Conclusion

Wireless Sensor Networks is an up and coming technology that is being deployed for a myriad of applications at a very fast rate [14]. The WSN market growth and growth predictions show that it is a promising field of research for the industry and academia.

However, if WSN are to be widely adopted in fields where reliability and robustness are an issue, some progress must be done on the validation and verification methods of WSN systems.

So far, validation has been done by combining different approaches during the different development stages of a WSN system (Figure 1.16).

There has been a lot of research on the development of WSN protocols, and many WSN protocols have been developed in theory, but in comparison, few have been actually deployed due to the complexity and cost of the development and validation of a WSN system.



Figure 1.16: Different WSN system validation methods.

According to Code Complete [100], the average number of bugs in released industrial software is about 15 to 50 per 1000 delivered lines of code. Consequently, it is likely that released WSN communication standards still have bugs.

If a company wishes to use a WSN solution, it should be able to test that the WSN product that it has bought is reliable and complies to the corresponding specifications. In that case, the protocol's source code is often inaccessible, which makes validation methods such as general purpose simulators impossible to use, and makes testbeds and emulators the best solutions to test the executable binary file of the protocol. However, emulators can only be used if the information on the platform type is available, and if that type of platform is supported.

EDF R&D has been developing a WSN protocol named OCARI, and needed a new solution to help with the protocol's validation. The requirements were the following:

- Validating the compiled protocol as deployed on real nodes
- Visibility on the state of the protocol's execution and the states of memories and registers of the hardware platform running the protocol, meaning using an hardware emulator
- Validating protocol properties over a wide range of scenarios
- Reproducing lab experiments for which the source of bugs could not be determined
- An Open Source solution

Given the need for visibility of the protocol's execution, we looked into using a emulator or a simulator that offers the possibility to model hardware using for example the SystemC hardware description language.

The following table recapitulates the different particularities of the different simulators and emulators reviewed in this State of the Art chapter. The different criteria that are studied are the language in which the simulator is developed, whether its source codes are open source, whether it supports cross-compiling (ability to execute the protocol's binary code), a possible support for a SystemC extension to potentially model a hardware platform, the presence of a GUI and finally, the simulator's main drawbacks for the needs that we have identified to validate a binary protocol file.

Simulator	Language	Open source	Cross-compiling support	GUI	SystemC Extension support	Main drawbacks
NS-2	C++, OTcl	Yes	No	Yes	Described in literature [101]	- Doesn't scale well - Complicated installation and use
NS-3	C++	Yes	No	Yes	No	- Complicated to use
OMNet++	C++, NED	Yes	No	Yes	Yes With commercial licence	- No environmental model - Lack of scalability
OPNET	C, C++	No	No	Yes	No	- Hardware specific - Limited documentation
Atemu	C, XML	Yes	Yes	No	No	- Limited scalability - HW specific
Avrora	Java	Yes	Yes AVR μ C	No	No	- HW specific - No clk manager model
Tossim	NesC	Yes	Yes TinyOS nodes	Yes	No	- All nodes run the same code - Can't capture properties linked to interruptions and time
Cooja/MSPsim	Java	Yes	Yes MSP430	Yes	No	- Limited to Contiki OS
Worldsens	C	Yes	Yes MSP430 ATmega128	Yes	No	- No longer supported - Lack of diversity of modeled architectures
SCNSL	C++ SystemC	Yes	No	No	Yes	- Lack of HW and protocol models
IDEA1TLM	C++ SystemC	Yes	Yes AVR μ C MSP430	Yes	Yes	- Only small microcontroller units
Emstar	C	No	Yes	Yes	No	- Limited to iPAQ sensors and MICA2 motes - Limited scalability
ATOSSIM	NesC	Yes	Yes	Yes	No	- Timing errors for large scale scenarios
H-TOSSIM	NesC	Yes	Yes	Yes	No	- Limited to TinyOs based nodes
SensorSim	C++, OTcl	Yes	No	Yes	No	- No synchronization between real and virtual nodes

After reviewing all these simulators, we were not satisfied with any of them for the needs identified to validate a binary protocol implementation. The simulators that are open source with cross compiling support do not model a large enough variety of hardware platforms (limited to AVR, TinyOS, MSP430 and ATmega128 microcontrollers), so they are not compatible with the validation of many WSN protocols. For example, the OCARI protocol, which runs on an ARM Cortex-M3 based microcontroller, could not be emulated using these simulators. The only open source simulators that openly supports a SystemC extension that would enable modeling hardware are SCNSL and IDEA1 (based on SCNSL), which are actually entirely based on SystemC. We considered using the beta version of SCNSL and associating it with a virtual platform [89]. The main limitation of this solution was that only one instance of a virtual platform modeling the node's hardware could be co-simulated with SCNSL.

This is the reason why we decided to develop a new WSN simulation framework based on the emulation of several hardware platforms of WSN nodes, that could be easily adapted to a large variety of hardware platforms.

Bibliography

- [1] I. F. Akyildiz, T. Melodia, and K. R. Chowdhury, "Wireless multimedia sensor networks: Applications and testbeds," *Proceedings of the IEEE*, vol. 96, no. 10, pp. 1588–1605, 2008. 8, 23, 27
- [2] J. E. Hardy, W. W. Manges, J. A. Gutierrez, and P. T. V. Kuruganti, *Wireless Sensors and Networks for Advanced Energy Management*. United States. Department of Energy, 2005. 8
- [3] Silicon Labs, "The evolution of wireless sensor networks," 2013. 9
- [4] C.-Y. Chong and S. P. Kumar, "Sensor networks: evolution, opportunities, and challenges," *Proceedings of the IEEE*, vol. 91, no. 8, pp. 1247–1256, 2003. 9
- [5] M. Martonosi, "The princeton zebranet project: Sensor networks for wildlife tracking," *Princeton University*, pp. 2–7, 2004. 9
- [6] A. Bagula, "Applications of wireless sensor networks," 2012. University of Cape Town, wireless.ictp.it/wp-content/uploads/2012/02/WSN-Applications.pdf. 9
- [7] "Project: Ieee p802.15 working group for wireless personal area networks(wpans)," 2008. mentor.ieee.org/802.15/file/08/15-08-0154-00-0006-capsule-endoscope-using-an-implant-wban.pdf. 9
- [8] A. Nayak and I. Stojmenovic, *Wireless sensor and actuator networks: algorithms and protocols for scalable coordination and data communication*. John Wiley & Sons, 2010. 10
- [9] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, E. Theodoridis, *et al.*, "Smartsantander: Iot experimentation over a smart city testbed," *Computer Networks*, vol. 61, pp. 217–238, 2014. xi, 10, 24, 25
- [10] Industrial Controls, "Industrial wireless cost savings: A tank farm example," 2017. www.industrialcontrolsonline.com/training/online/industrial-wireless-cost-savings-tank-farm-example. xi, 10
- [11] US Department of Energy, "Industrial wireless technology for the 21st century," 2002. Office of Energy and Renewable Energy Report, www1.eere.energy.gov/manufacturing/industries_technologies/sensors_automation/pdfs/wireless_technology.pdf. 10, 11
- [12] A. Sikora and V. F. Groza, "Coexistence of ieee802. 15.4 with other systems in the 2.4 ghz-ism-band," in *2005 IEEE Instrumentation and Measurement Technology Conference Proceedings*, vol. 3, pp. 1786–1791, IEEE, 2005. 11

- [13] P. Harrop and R. Das, "Wireless sensor networks (wsn) 2012–2022: forecasts, technologies, players—the new market for ubiquitous sensor networks (usn)," *IDTechEx, Tech. Rep.*, 2012. 12
- [14] PR Newswire, "Wireless sensor networks (wsn) 2014-2024: Forecasts, technologies, players," 2015. www.prnewswire.com/news-releases/wireless-sensor-networks-wsn-2014-2024-forecasts-technologies-players-300147357.html. 12, 13, 36
- [15] i-scoop, "Internet of things – iot guide with definitions, examples, trends and use cases," 2016. www.i-scoop.eu/internet-of-things-guide/. xi, 12
- [16] i-scoop, "Internet of things – iot guide with definitions, examples, trends and use cases," 2016. www.i-scoop.eu/internet-of-things/industrial-internet-things-market/. 12
- [17] P. Rawat, K. D. Singh, H. Chaouchi, and J. M. Bonnin, "Wireless sensor networks: a survey on recent developments and potential synergies," *The Journal of supercomputing*, vol. 68, no. 1, pp. 1–48, 2014. 13
- [18] K. Kotay, R. Peterson, and D. Rus, "Experiments with robots and sensor networks for mapping and navigation," in *Field and Service Robotics*, pp. 243–254, Springer, 2006. 13
- [19] E. Ngai and F. Riggins, "Rfid: Technology, applications, and impact on business operations," *International Journal of Production Economics*. 13
- [20] N. Zaman, *Wireless Sensor Networks and Energy Efficiency: Protocols, Routing and Management: Protocols, Routing and Management*. IGI Global, 2012. 14, 23, 25, 26
- [21] A. Bachir, M. Dohler, T. Watteyne, and K. K. Leung, "Mac essentials for wireless sensor networks," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 2, pp. 222–248, 2010. 14
- [22] K. Pister and L. Doherty, "Tsmf: Time synchronized mesh protocol," *IASTED Distributed Sensor Networks*, pp. 391–398, 2008. 14
- [23] IEEE standards association, "Ieee standard for low-rate wireless networks," 2015. 14, 15
- [24] T. Van Dam and K. Langendoen, "An adaptive energy-efficient mac protocol for wireless sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems*, pp. 171–180, ACM, 2003. 14
- [25] S. Mahlke and M. Bock, "Csma-mps: a minimum preamble sampling mac protocol for low power wireless sensor networks," in *Factory Communication Systems, 2004. Proceedings. 2004 IEEE International Workshop on*, pp. 73–80, IEEE, 2004. 15
- [26] I. Rhee, A. Warrier, M. Aia, J. Min, and M. L. Sichitiu, "Z-mac: a hybrid mac for wireless sensor networks," *IEEE/ACM Transactions on Networking (TON)*, vol. 16, no. 3, pp. 511–524, 2008. 15
- [27] G. Anastasi, M. Conti, and M. Di Francesco, "The mac unreliability problem in ieee 802.15. 4 wireless sensor networks," in *Proceedings of the 12th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems*, pp. 196–203, ACM, 2009. 15

- [28] Z. Alliance, "Ieee 802.15. 4, zigbee standard," 2009. 16
- [29] ZigBee Standards Organization, "Zigbee specification," 2008. people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2011/kjb79_-ajm232/pmeter/ZigBee 16
- [30] J. Song, S. Han, A. Mok, D. Chen, M. Lucas, M. Nixon, and W. Pratt, "Wirelesshart: Applying wireless technology in real-time industrial process control," in *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*, pp. 377–386, IEEE, 2008. 16
- [31] P. Radmand, A. Talevski, S. Petersen, and S. Carlsen, "Comparison of industrial wsn standards," in *2010 4th IEEE International Conference on Digital Ecosystems and Technologies (DEST), IEEE, Dubai, United Arab Emirates*, pp. 632–637, 2010. 16
- [32] G. Wang, "Comparison and evaluation of industrial wireless sensor network standards isa100. 11a and wirelesshart," 2011. 16
- [33] R. S. Wagner, "Standards-based wireless sensor networking protocols for space-flight applications," in *Aerospace Conference, 2010 IEEE*, pp. 1–7, IEEE, 2010. 16
- [34] Z. Shelby and C. Bormann, *6LoWPAN: The wireless embedded Internet*, vol. 43. John Wiley & Sons, 2011. 17
- [35] K. Al Agha, M.-H. Bertin, T. Dang, A. Guitton, P. Minet, T. Val, and J.-B. Viollet, "Which wireless technology for industrial wireless sensor networks? the development of ocar technology," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 10, pp. 4266–4278, 2009. 1, 17
- [36] Telit, "Z-one pro protocol stack user guide," 2010. www.telit.com/fileadmin/user_upload/media/products/short_range/ZE51-2.4__ZE61-2.4/Telit_Z_ONE_PRO_Protocol_Stack_User_Guide_r0.pdf. 17
- [37] F. Saad Khorchef, *Un cadre formel pour le test de robustesse des protocoles de communication*. PhD thesis, Bordeaux 1, 2006. 20
- [38] J. Francomme, K. Godary-Dejean, and T. Val, "Validation formelle d'un mécanisme de synchronisation pour réseaux sans fil," in *CFIP'2009*, 2009. 20, 21
- [39] J. Qadir and O. Hasan, "Applying formal methods to networking: Theory, techniques, and applications," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 256–291, 2015. 20
- [40] A. Mouradian, *Proposition et vérification formelle de protocoles de communications temps-réel pour les réseaux de capteurs sans fil*. PhD thesis, Lyon, INSA, 2013. 20, 21, 22
- [41] C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of model checking*. MIT press, 2008. 21
- [42] J.-Y. Le Boudec and P. Thiran, *Network calculus: a theory of deterministic queuing systems for the internet*, vol. 2050. Springer Science & Business Media, 2001. 21
- [43] J. C. Baeten, "A brief history of process algebra," *Theoretical Computer Science*, vol. 335, no. 2-3, pp. 131–146, 2005. 22

- [44] I. Romero-Hernandez, *Modélisation, spécification formelle et vérification de protocoles d'interaction: une approche basée sur les rôles*. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 2004. 22
- [45] P. Merlin and D. Farber, "Recoverability of communication protocols—implications of a theoretical study," *IEEE transactions on Communications*, vol. 24, no. 9, pp. 1036–1043, 1976. 22
- [46] Y. Atamna, *Réseaux de Petri temporisés stochastiques classiques et bien formés: définition, analyse et application aux systèmes distribués temps réel*. PhD thesis, 1994. 22
- [47] W. Fokkink, *Introduction to process algebra*. Springer Science & Business Media, 2013. 22
- [48] F. Zhang, L. Bu, L. Wang, J. Zhao, X. Chen, T. Zhang, and X. Li, "Modeling and evaluation of wireless sensor network protocols by stochastic timed automata," *Electronic Notes in Theoretical Computer Science*, vol. 296, pp. 261–277, 2013. 22
- [49] E.-R. Olderog and H. Dierks, *Real-time systems: formal specification and automatic verification*. Cambridge University Press, 2008. 22
- [50] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *Formal methods for the design of real-time systems*, pp. 200–236, Springer, 2004. 22
- [51] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *International Conference on Computer Aided Verification*, pp. 585–591, Springer, 2011. 22
- [52] J. Berendsen, D. N. Jansen, and F. Vaandrager, "Fortuna: Model checking priced probabilistic timed automata," in *Quantitative Evaluation of Systems (QEST), 2010 Seventh International Conference on the*, pp. 273–281, IEEE, 2010. 22
- [53] J. Horneber and A. Hergenröder, "A survey on testbeds and experimentation environments for wireless sensor networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 1820–1838, 2014. 23, 26
- [54] C. B. des Roziers, G. Chelius, T. Ducrocq, E. Fleury, A. Fraboulet, A. Gallais, N. Mitton, T. Noël, and J. Vandaele, "Using senslab as a first class scientific tool for large scale wireless sensor network experiments," in *International Conference on Research in Networking*, pp. 147–159, Springer, 2011. 23
- [55] G. Papadopoulos, J. Beaudaux, A. Gallais, T. Noel, and G. Schreiner, "Adding value to WSN simulation using the IoT-LAB experimental platform," in *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*, pp. 485–490, Oct 2013. xi, 24
- [56] I. Chatzigiannakis, S. Fischer, C. Koninis, G. Mylonas, and D. Pfisterer, "Wisebed: an open large-scale wireless sensor network testbed," in *International Conference on Sensor Applications, Experimentation and Logistics*, pp. 68–87, Springer, 2009. 23

- [57] I. Khan, F. Belqasmi, R. Glitho, N. Crespi, M. Morrow, and P. Polakos, "Wireless sensor network virtualization: A survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 553–576, 2015. 1, 24
- [58] A.-S. Tonneau, N. Mitton, and J. Vandaele, "A survey on (mobile) wireless sensor network experimentation testbeds," in *2014 IEEE International Conference on Distributed Computing in Sensor Systems*, pp. 263–268, IEEE, 2014. 24
- [59] M. Grottke and K. S. Trivedi, "A classification of software faults," *Journal of Reliability Engineering Association of Japan*, vol. 27, no. 7, pp. 425–438, 2005. 25, 79
- [60] G. Coulson, B. Porter, I. Chatzigiannakis, C. Koninis, S. Fischer, D. Pfisterer, D. Bimschas, T. Braun, P. Hurni, M. Anwander, *et al.*, "Flexible experimentation in wireless sensor networks," *Communications of the ACM*, vol. 55, no. 1, pp. 82–90, 2012. 26, 27, 29
- [61] N. Nasreddine, *Conception et modélisation d'un émulateur de réseaux de capteurs sans fils*. PhD thesis, INSA de Toulouse, 2012. 26
- [62] S. Kurkowski, T. Camp, and M. Colagrosso, "Manet simulation studies: the incredible," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 9, no. 4, pp. 50–61, 2005. 27
- [63] I. Minakov, R. Passerone, A. Rizzardi, and S. Sicari, "A comparative study of recent wireless sensor network simulators," *ACM Transactions on Sensor Networks (TOSN)*, vol. 12, no. 3, p. 20, 2016. 27, 28, 29, 30, 31
- [64] W. Du, *Modélisation et simulation de réseaux de capteurs sans fil*. PhD thesis, Ecully, Ecole centrale de Lyon, 2011. 27, 29
- [65] K. Fall, "Network emulation in the vint/ns simulator," in *Computers and Communications, 1999. Proceedings. IEEE International Symposium on*, pp. 244–250, IEEE, 1999. 27
- [66] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, "Directed diffusion for wireless sensor networking," *IEEE/ACM Transactions on Networking (ToN)*, vol. 11, no. 1, pp. 2–16, 2003. 27
- [67] W. Ye, J. Heidemann, and D. Estrin, "Medium access control with coordinated adaptive sleeping for wireless sensor networks," *IEEE/ACM Transactions on Networking (ToN)*, vol. 12, no. 3, pp. 493–506, 2004. 27
- [68] A. P. Das and S. M. Thampi, "Simulation tools for underwater sensor networks: A survey," *Network Protocols and Algorithms*, vol. 8, no. 4, pp. 41–55, 2017. 27
- [69] D. McLoughlin, E. O'Connell, W. Elgenaidi, J. Coleman, and T. Newe, "Review and evaluation of wsn simulation tools in a cloud based environment," in *Sensing Technology (ICST), 2016 10th International Conference on*, pp. 1–6, IEEE, 2016. 28
- [70] A. Varga, "Discrete event simulation system," in *Proc. of the European Simulation Multiconference (ESM'2001)*, 2001. 28
- [71] N. Zhu, *Simulation and optimization of energy consumption in wireless sensor networks*. PhD thesis, Ecole Centrale de Lyon, 2013. 28

- [72] Riverbed, “Opnet,” 2017. www.riverbed.com/fr/products/steelcentral/opnet.html. 28
- [73] C. Marghescu, M. Pantazica, A. Brodeala, and P. Svasta, “Simulation of a wireless sensor network using opnet,” in *Design and Technology in Electronic Packaging (SI-ITME), 2011 IEEE 17th International Symposium for*, pp. 249–252, IEEE, 2011. 29
- [74] D. Blazakis, J. McGee, D. Rusk, and J. S. Baras, “Atemu: a fine-grained sensor network simulator,” in *Sensor and Ad Hoc Communications and Networks (IEEE SECON). First Annual IEEE Communications Society Conference on*, pp. 145–152, IEEE, 2004. 29
- [75] crossbow technology, inc, “Mica2 datasheet,” 2004. www.eol.ucar.edu/isf/facilities/isa/internal/CrossBow/DataSheets/mica2.pdf. 29
- [76] B. L. Titzer, D. K. Lee, and J. Palsberg, “Avrora: Scalable sensor network simulation with precise timing,” in *Proceedings of the 4th international symposium on Information processing in sensor networks*, p. 67, IEEE Press, 2005. 29
- [77] P. Levis, N. Lee, M. Welsh, and D. Culler, “Tossim: Accurate and scalable simulation of entire tinyos applications,” in *Proceedings of the 1st international conference on Embedded networked sensor systems*, pp. 126–137, ACM, 2003. 29
- [78] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, *et al.*, “Tinyos: An operating system for sensor networks,” in *Ambient intelligence*, pp. 115–148, Springer, 2005. 29
- [79] A. Fraboulet, G. Chelius, and E. Fleury, “Worldsens: development and prototyping tools for application specific wireless sensors networks,” in *Information Processing in Sensor Networks. 6th International Symposium on*, pp. 176–185, IEEE, 2007. 29, 30
- [80] G. Chelius, A. Fraboulet, and E. Fleury, “Worldsens: a fast and accurate development framework for sensor network applications,” in *Proceedings of the 2007 ACM symposium on Applied computing*, pp. 222–226, ACM, 2007. 30
- [81] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón, “Cooja/mspsim: interoperability testing for wireless sensor networks,” in *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, p. 27, ICST, 2009. 30
- [82] Contiki, “Contiki: The open source os for the internet of things,” 2017. www.contiki-os.org/. 30
- [83] K. Roussel, Y.-Q. Song, and O. Zendra, “Using cooja for wsn simulations: Some new uses and limits,” in *EWSN 2016—NextMote workshop*, pp. 319–324, Junction Publishing, 2016. 30
- [84] N. Fournel, A. Fraboulet, G. Chelius, E. Fleury, B. Allard, and O. Brevet, “Worldsens: from lab to sensor network application development and deployment,” in *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pp. 551–552, IEEE, 2007. xi, 30, 31

- [85] F. Fummi, D. Quaglia, and F. Stefanni, "A systemc-based framework for modeling and simulation of networked embedded systems," in *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pp. 49–54, IEEE, 2008. 31, 127
- [86] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the ground up*, vol. 71. Springer Science & Business Media, 2011. xi, 2, 31, 51, 52
- [87] F. Fummi, G. Perbellini, D. Quaglia, and A. Acquaviva, "Flexible energy-aware simulation of heterogenous wireless sensor networks," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pp. 1638–1643, April 2009. 31
- [88] P. Sayyah, M. T. Lazarescu, S. Bocchio, E. Ebeid, G. Palermo, D. Quaglia, A. Rosti, and L. Lavagno, "Virtual platform-based design space exploration of power-efficient distributed embedded applications," *ACM Trans. Embed. Comput. Syst.*, vol. 14, Apr. 2015. 31
- [89] C. Barnes, J.-M. Cottin, D. Quaglia, E. Fraccaroli, A. Pegatoquet, F. Verdier, and S. Angeleri, "Network-aware virtual platform for the verification of embedded software for communications," in *Digital System Design (DSD), 2015 Euromicro Conference on*, pp. 518–525, IEEE, 2015. 31, 39
- [90] W. Du, F. Mieyeville, D. Navarro, and I. O. Connor, "Ideal: A validated systemc-based system-level design and simulation environment for wireless sensor networks," *EURASIP Journal on Wireless Communications and Networking*, vol. 2011, no. 1, p. 143, 2011. 32
- [91] H. V. Bitencourt, A. B. Da Cunha, and D. C. Da Silva, "Simulation domains for networked embedded systems," in *Student Forum on Microelectronics, SForum*, 2012. 32
- [92] M. Galos, F. Mieyeville, D. Navarro, and I. O'Connor, "Systemc fine-grained hw-sw fully heterogeneous wsn simulation and uml metamodel behavioural extraction," *Analog Integrated Circuits and Signal Processing*, vol. 77, no. 2, pp. 123–133, 2013. 32
- [93] S. Saginbekov and C. Shakenov, "Testing wireless sensor networks with hybrid simulators," *arXiv preprint arXiv:1602.01567*, 2016. 32, 33, 34
- [94] A. De Paola, G. L. Re, F. Milazzo, and M. Ortolani, "Adaptable data models for scalable ambient intelligence scenarios," in *Information Networking (ICOIN), 2011 International Conference on*, pp. 80–85, IEEE, 2011. 32
- [95] L. Girod, N. Ramanathan, J. Elson, T. Stathopoulos, M. Lukac, and D. Estrin, "Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks," *ACM Transactions on Sensor Networks (TOSN)*, vol. 3, no. 3, p. 13, 2007. 32, 33
- [96] A. Lalomia, G. L. Re, and M. Ortolani, "A hybrid framework for soft real-time wsn simulation," in *Proceedings of the 2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pp. 201–207, IEEE Computer Society, 2009. 33
- [97] C. S. Metcalf, *TOSSIM live: Towards a testbed in a thread*. 2007. 33

- [98] L. Wenjun, X. Zhang, T. Weihua, Z. Xiaocong, *et al.*, “H-tossim: Extending tossim with physical nodes,” *Wireless Sensor Network*, vol. 1, no. 04, p. 324, 2009. 33
- [99] S. Park, A. Savvides, and M. B. Srivastava, “Sensorsim: A simulation framework for sensor networks,” in *Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pp. 104–111, ACM, 2000. 34
- [100] S. McConnell, *Code complete*. Pearson Education, 2004. 36
- [101] N. Drago, F. Fummi, and M. Poncino, “Modeling network embedded systems with ns-2 and systemc,” in *Circuits and Systems for Communications, 2002. Proceedings. ICCSC’02. 1st IEEE International Conference on*, pp. 240–245, IEEE, 2002. 38

Chapter 2

Developing the model of a node's hardware platform

Contents

2.1	Selected tools to model the hardware platform	51
2.1.1	SystemC	51
2.1.2	TLM 2.0	53
2.1.3	QEMU	55
2.1.4	TLMu	56
2.2	Developing a platform model: The Atmel SAM3S	61
2.2.1	Cortex-M3 CPU emulated by QEMU	62
2.2.2	The hardware peripherals of the Atmel SAM3S	63
2.2.3	The mechanism to relay HW Interrupts to the NVIC	66
2.2.4	The AT86RF233 Radio transceiver model	67
2.2.5	Synchronization issues and solutions	69
2.2.6	Conclusion	72
	Bibliography	73

A WSN protocol is a complex reactive system. Its development goes through several stages, as described in Section 2.b.2. In the final stages corresponding to a pre-industrialization phase of the protocol, it is not only the functionalities of the protocol that must be tested, but its entire implementation, as well as the correct interaction of the protocol with the hardware platform it is executed on. For this purpose, we have decided to develop a new simulation environment called SNOOPS (Sensor Network simulatiOn for the validatiOn of Protocol implementationS). This environment would serve the purpose of facilitating the final validation phases of a protocol before its deployment, but also to test that deployed protocols comply with their specifications, without need for the source codes to be available. The features necessary to achieve these goals for SNOOPS are:

- The capability of executing the binary code of a compiled protocol
- A model of the hardware platform the protocol was designed to work on, and flexibility on the types of hardware platforms that it is possible to model
- The validation of protocol properties during simulation based on the protocol's specifications
- Random scenario generation to test all cases of the protocol's execution, even unlikely ones
- The possibility to reproduce real life scenarios that have gone wrong to find the source of the observed bugs

An additional constraint to the development of this simulation environment is that it be based solely on open source codes, so that it may be redistributed as open source. This should insure the durability of this project as anyone will be able to use and improve it.

Given the constraint of basing the simulation environment on open source codes, we have chosen to use:

- **QEMU** (Quick EMUlator) to model the CPU of the sensor node, as it is an open source platform virtualizer that offers a wide range of processor, platform and peripheral models
- **SystemC-TLM** to model the different hardware peripherals and the network connections, as it is a popular open source hardware description language based on C++ (compatible with QEMU based on C), and that it can model both hardware and software at various level of abstraction. This is compatible with modeling nodes with a high precision, as well as higher level, more simple nodes.
- **TLMu** is used as an interface between QEMU and SystemC. This interface is needed as QEMU and SystemC have different simulation cores, their notion of time is different and their execution depend on each other's production of events. Therefore they need a solution to communicate and synchronize their notions of time. TLMu is an open source wrapper for QEMU that solves these issues, and allows instantiating several QEMU cores in the same simulation process where SystemC is master, therefore making it possible to simulate more than one node.

OCARI, the protocol under test in this work, runs on the Dresden Elektronik deRFsam3-23T09-3/23M09-3 platform (Atmel SAM3S and Atmel AT86RF233), as well as on the Ad-wave Adwrf24-LRS platform (Atmel SAM3S and Atmel AT86RF233 coupled with a LNA (Low Noise Amplifier) chip). In this chapter, we will describe the development of a sensor node model associating a model of the Atmel SAM3S microcontroller and the Atmel AT86RF233 transceiver using QEMU, SystemC and TLMu.

2.1 Selected tools to model the hardware platform

2.1.1 SystemC

2.1.1.1 Overview

SystemC [1] is a popular open source class library within C++. It has the advantage of providing a common language to model both hardware and software, so it can be used to design both the architecture and the behavior of a hardware component. SystemC also offers the advantage of integrating both hardware and software modules at different abstraction levels, such as the transaction level or register level, to perform HW/SW co-simulation.

A comparison of SystemC's modeling capacities with other hardware description languages is shown in Figure 2.1. As it can be seen, SystemC can be used for models with a wide range of abstraction levels, from simple System Requirements down to the system's RTL description level.

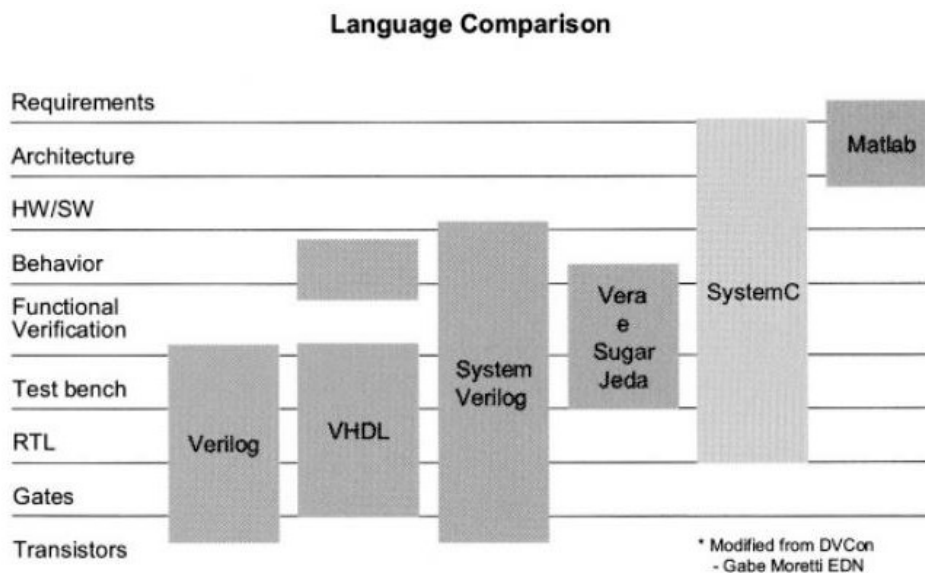


Figure 2.1: SystemC comparison with other hardware description languages [1].

SystemC's main hardware oriented features are:

- Its time model: the time is tracked using a class called *sc_time* with a 64 bit resolution. This class allows obtaining the current time and to generate time delays. SystemC also provides a class named *sc_clock* for modules that require clocks.
- The hardware data types which support explicit bit widths
- Hierarchy within modules: SystemC entities are called modules, they are interconnected through channels, and modules can be instantiated inside other modules to model hardware hierarchy
- Communications management: communications are modeled by channels, which are connected to modules using port classes. Channel implementations can vary greatly, from very simple (e.g. FIFO), to quite complex (e.g. AMBA bus).

- **Concurrency:** a simulation kernel is provided by SystemC to switch the simulation between each concurrent units, called processes, according to simulation events. This is referred to as a cooperative multitasking environment.

2.1.1.2 The simulation kernel

SystemC's simulation kernel is represented in Figure 2.2. The two principal phases of operation in SystemC are the elaboration and execution phase. The elaboration phase corresponds to all calls done before the call of the `sc_start()` function, which includes declaration and interconnection of different SystemC modules. The execution phase is handled by the simulation kernel, which coordinates the execution of the different processes. Processes are run when they are triggered by an event to which they are sensitive.

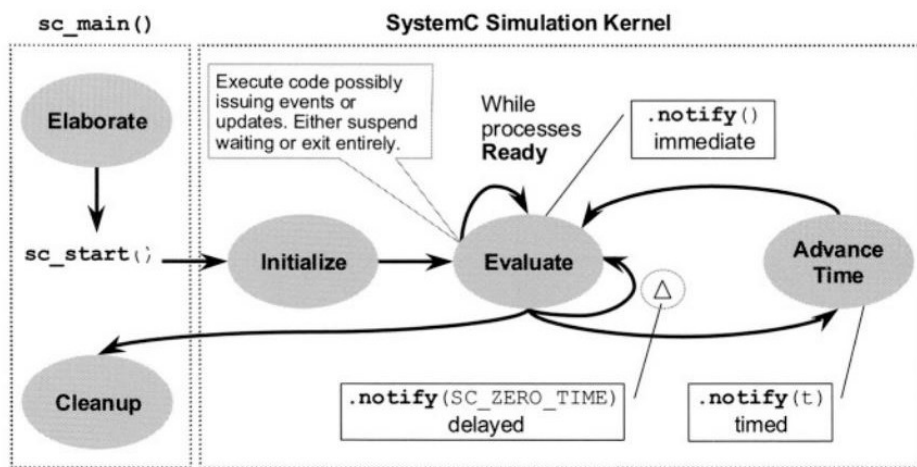


Figure 2.2: SystemC Simulation Kernel [1].

2.1.1.3 Modules and ports

SystemC needs to deal with very large designs, that is why it uses modules to implement hierarchy. Modules are the basic building blocks of a SystemC design. They are classes that allow a piece of design to be worked on separately. They may contain processes and instances of other modules.

Ports are used by modules to receive or send data of various type, they can be thought of as a pin. SystemC ports are object instances of predefined classes that can be accessed through the member functions of the module. Ports connect to other ports via signals, which can also be called channels. Modules generally use ports and signals to communicate with other modules, they can also be used to connect to a clock signal.

Another way for different modules to communicate is by using TLM transactions (structures containing among other a data pointer, an address, a command...) that are exchanged through TLM sockets. This mechanism is explained in Section 2.1.2.1.

2.1.1.4 Time

There are three different notions of time measurement in SystemC [1]. The wall clock time is the time for the simulation to run, including the time the system spent on other activities. The processor time does not include the time spent by the system on other activities. The simulated time is the time modeled in simulation. The simulation time is tracked

using a 64 bit unsigned integer that is set to 0 at the beginning of simulation. An *sc_time* class object contains an integer, a time unit, and a time resolution. The method *sc_time_stamp()* is used to obtain the current simulated time value, by returning an *sc_time* object. The method *wait()* allows introducing the notion of delay in SystemC processes, by blocking them for a time specified in the *wait()* call, to model realistic behaviors of components such as a signal transmission delay.

2.1.1.5 The concurrency of processes and events

In real systems, several activities may happen in the same instant. For example, a SoC model could have several timers incremented at the same instant. In SystemC, processes are used to model concurrency. However, concurrency is not truly executed concurrently, as a SystemC program is generally executed on only one processor. A simulation process runs and then returns the control to the SystemC kernel by calling a *return()* or *wait()* function. However the kernel cannot force a process to return the control. The processes in SystemC are simply pieces of C++ code designated as process by the user.

Static processes are registered before the simulation starts during the elaboration phase, which is the phase where modules are instantiated and connected. Dynamic processes can be registered after the start of simulation by calling *sc_spawn* within another process. After the elaboration phase, the simulation starts, and the next phase is the initialization. During this phase, the kernel identifies the processes and classifies them as either runnable or waiting process. There are two internal phases in the simulation stage, evaluate and run time. During the evaluation stage, runnable processes are run one at a time until they return the control to the kernel. There is no specified order in which the processes should be run. When all runnable processes have been executed, the kernel advances the time until the time of the next scheduled event in the event queue. Processes waiting for that time will become runnable.

A SystemC event happens at a single moment in time, it has no value or duration. They are caused using the *notify()* function of the *sc_event* class. If no processes are waiting for an event that occurs, it goes unnoticed.

2.1.2 TLM 2.0

TLM 2.0 is a Transaction Level Modeling standard [2]. In SystemC, TLM is used for inter-process communications using function calls. The main focus of TLM 2.0 is the modeling of memory mapped buses, but it can serve for other purposes. It consists in a set of core interfaces, mainly the global quantum (a time interval used for synchronization), initiator and target sockets and the generic payload. TLM 2.0 classes are layered on top of the SystemC class library.

2.1.2.1 TLM transactions – The generic payload

A TLM transaction or generic payload is a data structure passed from an initiator module to a target module through function calls, as shown in Figure 2.3. In order to forward those transactions, TLM uses a structure named TLM socket. There are initiator sockets and target sockets, and every initiator socket must be connected to a target socket. The module that needs to forward a transaction uses one of its initiator sockets to send the transaction by calling its *b_transport* method of the TLM 2.0 blocking transport interface. The transaction is received by the target socket that is connected to the initiator socket. At the reception of the transaction, a function from the module containing the target socket

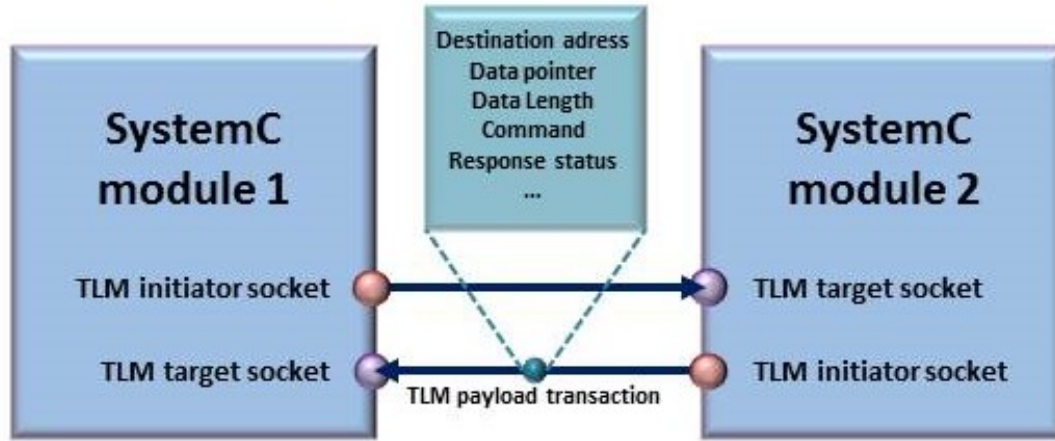


Figure 2.3: TLM transaction

is called. This function defines what must be done with the incoming transaction. A payload transaction object has a standard set of bus attributes, mainly an address, a data pointer, a data length, a TLM command, and a response status. The two commands that are supported are read and write, the address attribute indicates the lowest address at which the data must be read or written. The data pointer points to a buffer in the initiator socket. If the TLM command is read, the read data will be written in this buffer. If the command is write, the data to be written will be read from this buffer.

2.1.2.2 The global quantum

Temporal decoupling is the concept of several threads in a parallel system each having their own local time, and synchronizing only when they need to communicate. This is what allows SystemC processes to run ahead of simulation time for an amount of time known as the time quantum. The simulation speed is improved by temporal decoupling because it limits the number of context switches and events.

A quantum boundary is the local time offset of a SystemC process relative to the current simulation time as returned by the method *sc_time_stamp*. The global quantum is the default time interval between successive quantum boundaries. The value of the global quantum is handled by the *tlm_global_quantum* class. In this class, a local quantum time can be calculated based on the global quantum which is constant. It is calculated by subtracting the value of *sc_time_stamp* to the next integer multiple of the global quantum.

2.1.2.3 The TLM quantum keeper

The *tlm_quantumkeeper* is a utility class of TLM 2.0 that provides a set of methods for managing and interacting with the time quantum. It contains a method that allows setting the global quantum, which is the default time between synchronizations of the process with the simulation time. The *inc* method increases the local time offset. The *need_sync* method returns true if the local time offset is greater than the local quantum calculated by the *tlm_global_quantum* class. Finally, the *sync* method calls *wait(local_time_offset)* to suspend the process until the simulation time becomes equal to the effective local time, which corresponds to the local time offset added to the simulation time.

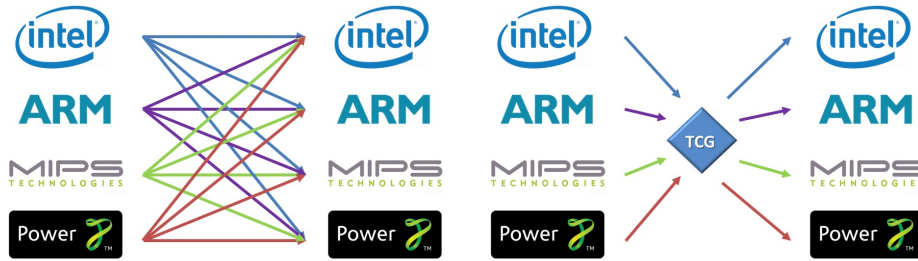
2.1.3 QEMU

2.1.3.1 Overview

QEMU (Quick EMUlator) [3] was created by Fabrice Bellard. It is an open source hardware emulator and virtualizer based on C that uses the dynamic binary translation of instructions to execute a code meant for one target architecture on another host architecture. The Instruction Set Architectures (ISA) supported by QEMU include, but are not limited to, x86, MIPS, SPARC, ARM, Microblaze and PowerPC. QEMU has two modes of operation. In the full system emulation mode, a full system is emulated including one or more processors and hardware peripherals. The user mode emulation is used to launch processes compiled for one CPU onto another. The first GIT code commit dates back to 2003 and the community around QEMU has been quite active since then, the freely available source codes are often updated [4].

2.1.3.2 Tiny Code Generator

QEMU uses the dynamic recompilation of the target code to emulate the target architecture on a host architecture. This recompilation is executed by what is called the Tiny Code Generator (TCG). Given that QEMU provides many host and many target ISA, an intermediate translation into bytecode of the target instructions is necessary. This way, instead of needing a specific translator from each target to each host ISA, one is needed to translate each target ISA into intermediate code, and to translate the intermediate code into each host ISA. This greatly diminishes the number of translators that have to be implemented as Figure 2.4 illustrates.



(a) Number of translators necessary without an intermediary translation (b) Number of translators necessary with the TCG

Figure 2.4: How the TCG optimizes the number of translators necessary between different architectures.

The TCG translates groups of several instructions at a time that are called translation blocks. Translation blocks are read, disassembled, and translated into blocs of TCG bytecode. An optimization is conducted at this stage to suppress TCG bytecode instructions that do not modify the result of the operation (e.g. `and_i32 t1, t1, t1`), or instructions that handle dead variables. After this optimization, the TGC bytecode blocks are translated into blocks of native opcodes of the host processor. This native bloc can then be executed on the host processor. Recently translated blocks are stored in a translation cache, so that they may be reused without going through the different translation stages.

The size of a translation block can vary. It is composed of a list of subsequent instructions terminated by a branch instruction (i.e. a basic block). There cannot be a branch

instruction in the middle of a translation block. No instruction but the first of a translation block can be the target of a branch.

2.1.3.3 The clocks

It is unlikely that the execution time of the guest code on its native machine be the same as its execution time on the host machine. To get a notion of time in QEMU, three different clocks are present [5]. The *host_clock* provides information on the time of the host system using the function *gettimeofday()* of the standard C library. It provides the number of microseconds that have passed since January 1st, 1970. It is used when a real time clock is necessary. The *rt_clock* is based on the host system's *CLOCK_MONOTONIC*, which provides the number of nanoseconds that have passed since the system's boot. The *rt_clock* does not stop increasing its time when QEMU is executed. Finally, the *vm_clock* is the virtual machine clock. It is the reference clock for the guest system, and the virtual processor. It is also based on the host system's *CLOCK_MONOTONIC*, but is reset to 0 when the virtual machine starts. It increases its time at the same speed as the *rt_clock*, but the difference is that the *vm_clock* stops when the guest system stops. When the *-icount* option is specified in QEMU, the *vm_clock* increases according to the number of instructions executed by the virtual processor instead of being based on *CLOCK_MONOTONIC*. Using this option means that the *vm_time* will be proportional to the number of instructions executed, independently from the speed of the host system. Using the parameter *-icount x* will make the *vm_clock* increase its time by $2 \times x$ ns every time a guest instruction is executed. So if QEMU is launched with the option *-icount 2*, the *vm_clock* time will be increased by $2 \times 2 = 4$ ns every time a host instruction is executed, giving the virtual processor a frequency of 250 MHz.

2.1.3.4 GDB stub

QEMU provides a GDB stub to enable debugging on the guest code that is being executed on the virtual platform. By starting QEMU with the *-S* option, QEMU will support remote debugging with GDB and wait for a GDB remote connection on port 1234. The *-gdb* option allows the user to define a different port to connect the gdb stub. The guest program can then be debugged as if it were running on the host machine.

2.1.4 TLMu

TLMu is a wrapper concept for QEMU that integrates into SystemC-TLM 2.0. It handles communications and time synchronization between both simulation cores. TLMu is based on the QEMU-SystemC interface, which was originally proposed in 2007 by Marius Monton and GreenSocs [6].

2.1.4.1 History: QEMU-SystemC by Marius Monton and derived works

The original project of the open source emulation framework named QEMU-SystemC aimed at attaching devices modeled in SystemC to QEMU [7]. In QEMU-SystemC, QEMU is the simulation master and SystemC is the slave. To connect both simulation frameworks, a module is created in QEMU called *sc_link* with the same characteristics as the SystemC device. This *sc_link* module mimics the configuration space (for PCI devices) or memory map of the SystemC modules and receives read and write operations from the virtual CPU. These accesses are transmitted to the SystemC device and the response is

returned to QEMU. As shown in Figure 2.5, this communication happens through what is called the SystemC bridge, which converts the read and write accesses into TLM transactions that are sent to the SystemC device.

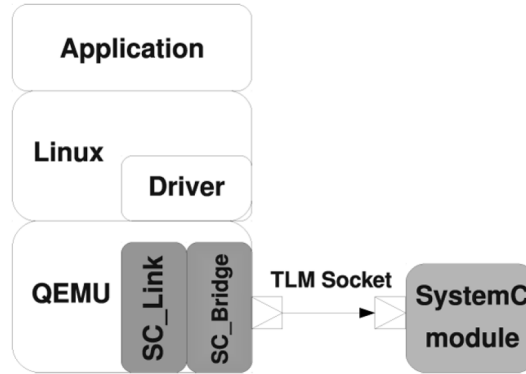


Figure 2.5: QEMU-SystemC block diagram [6].

Since its creation, the QEMU-SystemC framework has been reused and the association of QEMU and SystemC has been developed for different applications.

For instance, in [8], the authors modeled with the QEMU-SystemC framework a multi-processor platform, by replacing existing Instruction Set Simulators (ISS) in the platform model by processor models provided by QEMU. To obtain an accurate timing of the processor models, the QEMU models were modified at the price of a slower simulation speed. For the accurate timing of the time required for instruction execution, as the binary code is translated, micro-operations were added before the micro-operations of each translated target instruction, which increment the number of cycles of the instructions that have been simulated since the last synchronization. The number of cycles added are based on the number of cycles of the instructions given by the datasheet of the simulated processor. The time accuracy was also improved by checking if the cache line was either present or loaded into the instruction cache. To model the platform, each QEMU processor was wrapped in a SystemC wrapper and concurrently executed in a different SystemC process (SC_THREAD). Processors that share the same translation cache are encapsulated in the same SystemC module. The wrappers are connected to interconnect through which they can communicate with the different hardware components modeled in SystemC (memory, timers,...). This open source simulation framework was named Rabbits [9] and has led to the creation of the company Antfield [10] in 2016, which provides a catalog of hardware components in SystemC for Rabbits in order to facilitate the modeling a wide variety of hardware platforms.

As another example, the authors of [11] and [12] developed the QEMU-SC framework to be used for SoC development. In [11], they describe how they turned QEMU into an Instruction Accurate Instruction Set Simulator (IA-ISS), for more precise simulation. In [12], the authors explain the enhancements brought to the interface between QEMU and SystemC, mainly making interrupts signals bi-directional, so that interrupts can also be sent from QEMU to SystemC, for example in cases where the interrupt controller is modeled in SystemC.

A similar concept to the QEMU-SystemC framework is QBox [13], which was developed in part by the same company (GreenSoCs) from which originated QEMU-SC. The main difference between QEMU-SC and QBox is that in Qbox, QEMU is no longer the simulation master, SystemC is. QEMU is treated as a SystemC module within a larger

SystemC simulation context. In order for QEMU to be treated as a SystemC module, it is wrapped in a set of SystemC TLM-2.0 interfaces, this wrapper is called TLM2C. QEMU is compiled into QBox shared libraries, each QBox library containing a specific CPU core. SystemC and each QBox instance run in different threads, which can run at the same time, with the constraint that the time difference between the SystemC thread and each QBox thread does not exceed one quantum (time interval defined by the use).

In [14], a TriCore architecture was modeled with QEMU and SystemC using TLMU [15] as an interface, which is a wrapper concept for QEMU that integrates in SystemC similar to QBox in the sense that SystemC is the simulation master. The main difference with QBox is that all QEMU instances execute in the same process, while in QBox, there are separate processes for SystemC and each QEMU instance. Their synchronization method is therefore also different.

We also decided to use TLMu for our SoC model as we needed SystemC to be the simulation master. However, QBox could have also been used, but the QBox source codes were not yet available for download as we started developing the platform model, therefore the platform model was developed with TLMu. However, it is part of our perspectives to also test QBox to evaluate the advantages it could bring to our simulation environment.

TLMu as well as its synchronization mechanism are described in more detail in the following section.

2.1.4.2 TLMu: A version of QEMU-SystemC

TLMu [15] is an open source wrapper for QEMU that integrates into SystemC-TLM-2.0 models. In this Simulation framework, SystemC is the master and QEMU the slave. The advantage with TLMu is that several instances of QEMU can be called, enabling the simultaneous emulation of different virtual platforms.

In this project, the CPU emulators are built as shared libraries, one library per supported architecture (e.g. libtlu-arm.so). Each emulated CPU is executed as a different thread in the same process as SystemC. The TLMu wrapper loads QEMU as a shared object and accesses QEMU's main function by a loaded function pointer from the shared object. The wrapper defines callback functions. In QEMU, the SystemC environment is registered as a memory region, which when accessed uses a callback to access the TLM memory bus model to read or write memory spaces in SystemC. This is the mechanism that allows the control switch between QEMU and SystemC.

2.1.4.3 Time and synchronization in TLMu

Term definitions

To begin this section, in order to simplify its comprehension, we recall all the terms specific to the QEMU SystemC synchronization and their definition:

- **icount:** Using this QEMU option means that the virtual machine time will be proportional to the number of instructions executed, independently from the speed of the host system. Using the parameter *-icount x* will make the *vm_clock* increase its time by $2 \times x$ ns every time a guest instruction is executed
- **vm_clock:** When the *-icount* option is specified in QEMU, the *vm_clock* increases according to the number of instructions executed by the virtual processor. The *vm_time* will be proportional to the number of instructions executed, independently from the speed of the host system.

- **sc_time_stamp**: A SystemC method used to obtain the current simulated time value.
- **global_quantum**: In the TLM quantum keeper's class, it is the default time interval between successive quantum boundaries (theoretical time between synchronizations).
- **local_time**: The time that has passed in a SystemC thread since its last synchronization
- **local_quantum**: The local quantum can be calculated based on the global quantum which is constant, by subtracting the value of *sc_time_stamp* to the next integer multiple of the global quantum. When the *local_time* surpasses the *local_quantum*, a synchronization occurs.
- **speed_factor**: In TLMu, The *Speed_factor* is set according to the platform's frequency. It corresponds to the duration of a CPU cycle.
- **Delta_ns**: variable used in TLMu's *sync* method. It contains the time that has passed in QEMU since the last *sync* method call. That time is deduced from the number of instructions executed by the CPU and the *speed_factor*.

The synchronization mechanism

The time in SystemC and QEMU progress differently, and these time lines need to be synchronized [16]. In SystemC, the time progresses according to the information provided by the hardware platform. In the QEMU launched by TLMu, the time advances according to the number of guest instructions that have been executed. The time in only one simulation framework advances at a time as shown in Figure 2.6. The time in QEMU advances first, then QEMU is frozen until the time in SystemC catches up with the time QEMU was frozen at.

When TLMu instantiates a CPU from QEMU, it passes the parameter `-icount 1` to QEMU. As seen in Section 2.1.3.3, this means that QEMU's *vm_clock* will increase by $2^1 = 2ns$ every time a guest instruction is executed. This is equivalent to a 500 MHz CPU frequency, assuming that one instruction takes approximately one CPU cycle to be executed. A frequency adjustment is performed by TLMu, as it always uses the same *icount* parameter, and the user enters the desired CPU frequency as a parameter in the TLMu constructor.

Synchronization may be triggered by TLMu in several cases:

- When the CPU make I/O accesses into the main emulator
- When the CPU gets interrupted
- After the execution of a translation block

In those cases, a TLMu callback function will be called from QEMU called *sync()*, and QEMU passes as a function parameter its *vm_clock* time. As the *vm_clock* time increases by 2 ns every time a guest instruction is executed, TLMu deduces the number of instructions that have been executed since the last time *sync()* was called. A CPU frequency parameter is set in TLMu by the user. From this frequency and the number of instructions that have been executed since the last *sync()* function call, TLMu calculates the virtual time that has passed since the last time the function was called. Once that time is calculated, it is added to the local time value of the TLM quantum keeper that is used by TLMu

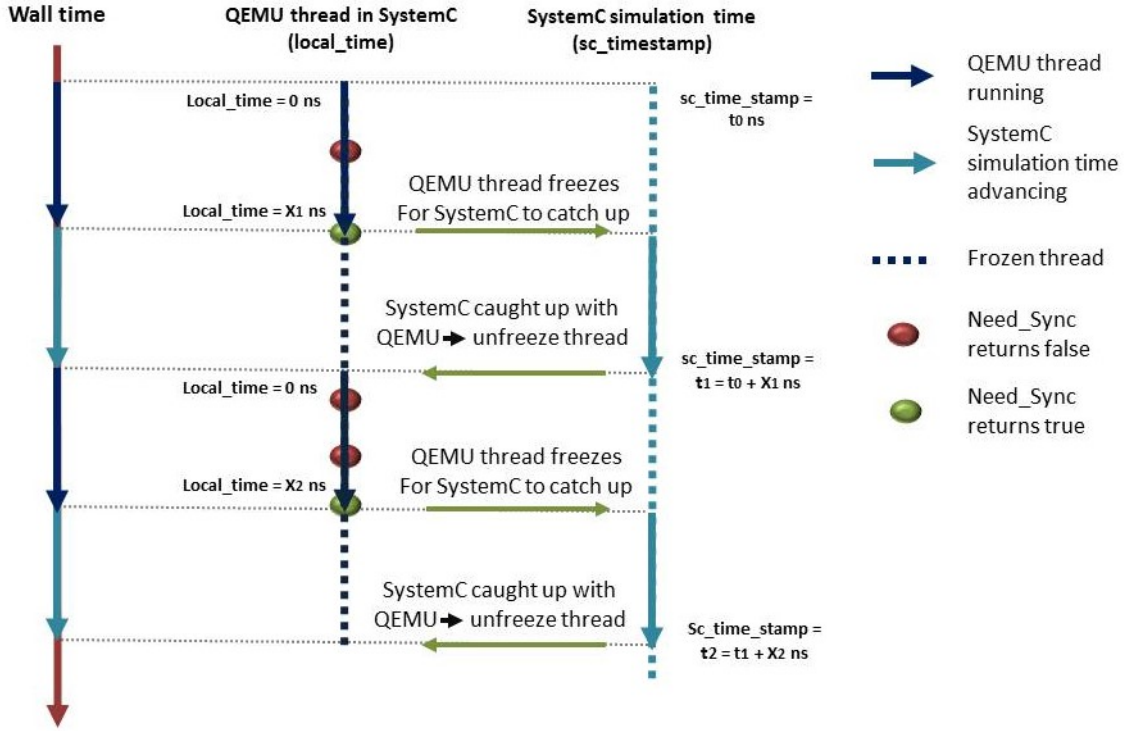


Figure 2.6: QEMU and SystemC threads execution.

to keep track of the time difference between QEMU and SystemC. Finally, the function *need_sync()* is called in the TLM quantum keeper object. This function evaluates if enough time has passed since the last synchronization for another synchronization to be needed. The time since the last synchronization is called the *local_time*. If it is evaluated as greater than the *local_quantum* time, a synchronization is performed. The *global_quantum* is a variable that is set by the user representing the time needed between synchronizations. If it is low, synchronizations will be more frequent and the execution time of the simulation will be longer. If it is high, the execution of the simulation will be faster, but at the cost of a lower accuracy. If the TLM quantum keeper determines that no synchronization is needed, the function returns and the time in QEMU keeps progressing until the next sync function call. When the TLM quantum keeper determines that a synchronization is needed, the *local_time* is reset to 0, the *local_quantum* is recalculated according to equation (2.1), and the QEMU thread is frozen by calling a *wait()* function on it, with the time difference between QEMU and SystemC as a parameter.

$$Local_quantum = Global_quantum - (sc_time_stamp \% Global_quantum) \quad (2.1)$$

All the other SystemC threads will then be able to catch up with this QEMU thread. Then QEMU will resume and continue executing instructions.

Figure 2.7 shows a synchronization example for which the *global_quantum* is set to 60 ns, so the initial *local_quantum* is also 60 ns.

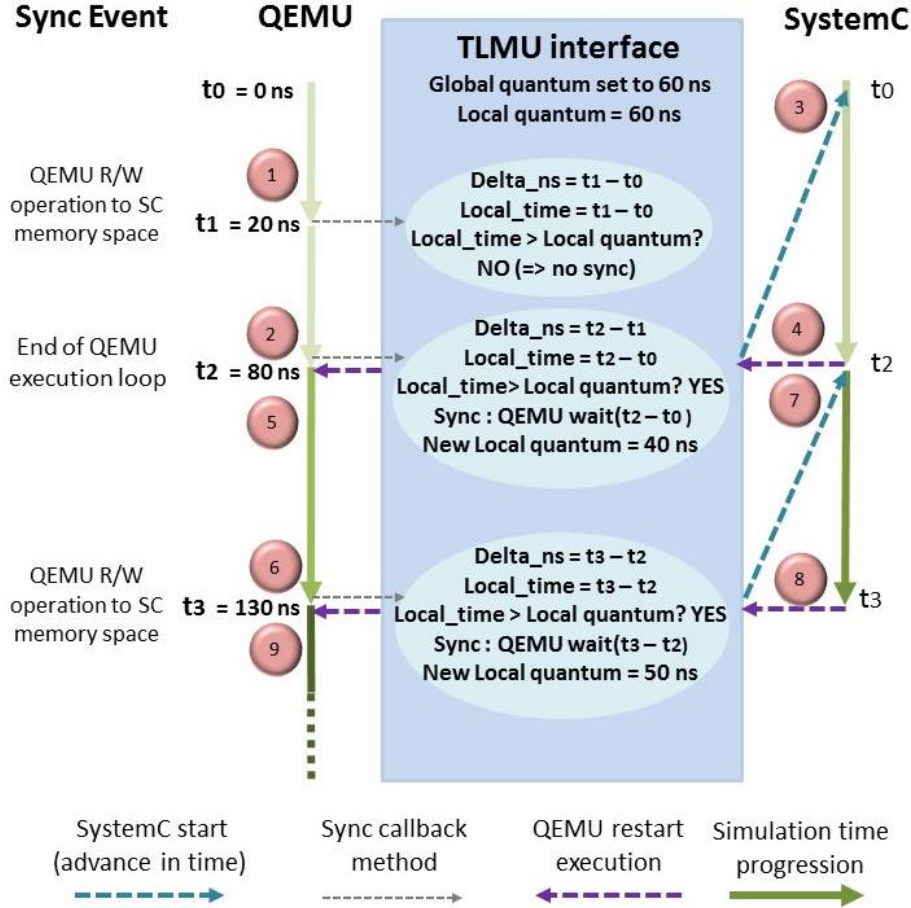


Figure 2.7: QEMU and SystemC synchronization through the TLMU interface.

2.2 Developing a platform model: The Atmel SAM3S

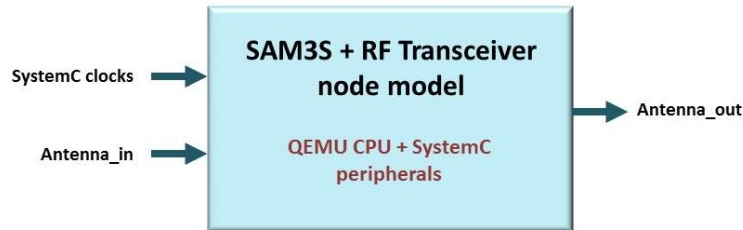


Figure 2.8: Simplified block diagram of the SAM3S platform model.

Many SoC models are already available with QEMU, and many others can be modeled by reusing the CPU models provided by QEMU. We modeled the Atmel SAM3S Microcontroller [17] and the AT86RF233 RF transceiver [18] as they compose one of the platforms used for the execution of the OCARI protocol, which we want to use as our experimentation subject.

Figure 2.8 shows the SAM3S and AT86RF233 transceiver SystemC module. The inputs to the module are the clocks which are connected to SystemC clock signals and Antenna_in in which is a TLM target socket. The output is the Antenna_out TLM initiator socket.

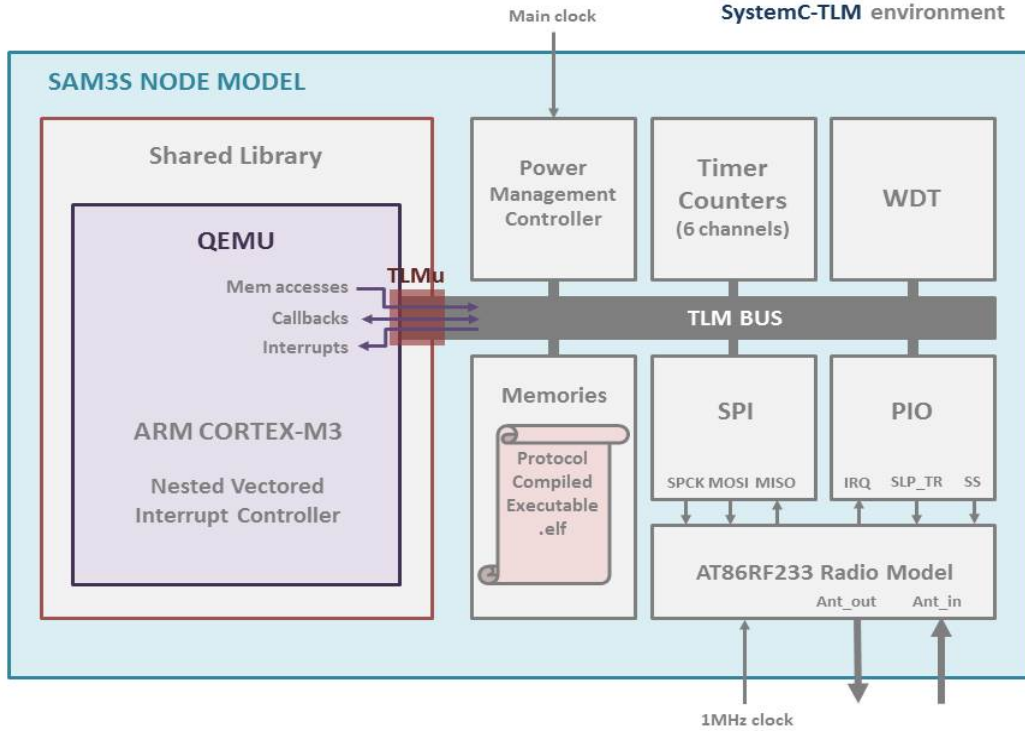


Figure 2.9: Simplified block diagram of the SAM3S platform modeled with SystemC and QEMU through the TLMu interface.

2.2.1 Cortex-M3 CPU emulated by QEMU

As shown in Figure 2.9, the Atmel SAM3S microcontroller is based on the ARM Cortex-M3 RISC processor. It is well suited for WSN applications as it offers a good ratio in terms of reduced power consumption, processing power and peripheral set.

The ARM Cortex-M3 is a high performance 32 bit processor for low power, cost sensitive, and highly deterministic real time operations. It is based on the ARMv7-M architecture and uses the thumb2 instruction set [19]. Its main advantage is its ultra-low power consumption with integrated sleep modes, while still providing an efficient processor core. A model of the cortex-M3 is available with QEMU as part of the Stellaris Luminary Micro platform models.

To use the Cortex-M3 CPU model with the TLMu wrapper, QEMU is launched through TLMu using the “Cortex-M3” parameter as the CPU model when instantiating the SystemC module called *tlmu_sc*. This module instantiates QEMU and the CPU. It is also used as interface between QEMU and SystemC, containing the callback functions for communications, such as the function *bus_access* handling the read and write operations coming from QEMU or callback functions for synchronization, such as *sync_time*. The *bus_access* function creates a TLM transaction containing the address of the memory access, a data pointer, a data length, and a command which is set to either read or write. If it is a write access, the data pointer points to the value to write. If it is a read access, the data pointer points to the value read after the transaction is sent to the intended peripheral. A TLM transaction also has a parameter indicating if the peripheral allows direct memory access. In this case, the peripheral sets this value to true, and when this information is returned to QEMU, it fetches the peripheral’s memory’s base address pointer through the function *get_direct_memory_pointer*. For all the subsequent memory accesses to this

peripheral, QEMU directly calls the `memcpy` function instead of the `bus_access` function. The `memcpy` function copies a number of bytes from a memory area to another. In this case, depending on if it is a read or write access, it copies 4 bytes to or from the address pointed by the base address pointer of the peripheral added to the access offset in the peripheral to or from the data pointer.

2.2.2 The hardware peripherals of the Atmel SAM3S

The different hardware peripherals of the SAM3S microcontroller were modeled as SystemC-TLM modules.

2.2.2.1 The memories

The memories module creates an address space for the different read and write access operations from the CPU. A part of the memory addressing space is dedicated to peripherals. To determine the required peripherals for the OCARI protocol's functionalities, or in other words which peripherals needed to be modeled, we tracked which peripherals' address spaces were accessed by the protocol at its initialization. Indeed, OCARI initializes the different parameters in the peripherals it has to use. If during simulation a peripheral which is not modeled is accessed, an error message is raised to indicate that this peripheral is not modeled. When a peripheral is modeled, the memory accesses (that pass through the `iconnect` type bus module) are diverted to the right peripheral according to the TLM transaction's address, instead of accessing the memories module by default.

The modeled SAM3S contains various memory spaces:

- The code memory space contains the boot memory, the internal ROM and flash memories
- The internal SRAM memory space
- The peripherals memory space is used to access all the peripherals
- The external SRAM
- The System memory space used for example to access the registers of the CPU's NVIC (Nested Vectored Interrupt Controller). This memory space is not accessed via TLM transactions, the access is internal to QEMU.

Memories other than peripheral memory are modeled as SystemC modules containing a table that is the same size as the memory. Each cell of the table represents a word. The memory modules can receive TLM transactions through a TLM target socket for read/write access.

The OCARI protocol's binary executable code is initially loaded in the internal flash memory which begins at address 0x400000, as it is usually loaded on physical nodes through a JTAG (a device used to transfer data into an internal non-volatile device memory).

2.2.2.2 The Power Management controller (PMC)

The PMC is used to optimize power consumption of the SoC by controlling the platform's clocks. The clock inputs to the different peripherals and the Cortex M3 CPU can be enabled or disabled by the PMC (i.e. Clock gating), their frequency can also be adjusted.

In the platform SystemC model, the clocks have to be generated from the *sc_main*. A clock signal is generated for the transceiver (RadioCLK) at a frequency of 1 MHz and another one is generated for the platform's hardware peripherals (MasterCLK). Both clock signals are connected to the SystemC module modeling the sensor node. The MasterCLK signal is connected to an *sc_in* port of the PMC. Depending on how the PMC's registers are configured by the protocol stack, the PMC can modify the frequency of the MasterCLK signal and supply some peripherals that are enabled with that signal. To optimize the time it takes to develop a platform model, only the peripherals having their clock input enabled or their registered accessed during the configuration phase of the execution on OCARI's protocol stack were modeled. If the stack tries to access peripherals that are not modeled, an error message is raised indicating that this peripheral needs to be modeled.

2.2.2.3 The parallel Input Output Controller (PIO)

The PIO manages 32 fully programmable input/output lines. Each line may be dedicated as a general purpose I/O or be assigned to a function of an embedded peripheral. The lines are programmed through set/clear registers. Interrupts can be triggered on input changes of the pins. A status register provides visibility of all the pin levels. In our model of the SAM3S, the PIO peripheral is mainly used to communicate with the model of the radio transceiver, the Atmel AT86RF233. To connect the PIO pins with the transceiver's pins, SystemC ports are used in the two modules connected through SystemC signals.

2.2.2.4 The Watchdog Timer (WDT)

A WDT is used to avoid system lock-ups in the case that the software gets trapped in a deadlock. In the SAM3S, it provides a 12 bit down counter offering a watchdog period of up to 16 seconds.

If the Watchdog Timer's countdown reaches 0, this means that a deadlock has occurred (watchdog underflow). An interrupt is sent causing the whole platform (microcontroller and transceiver) to reset. It is worth to note that it is not modeled as other interrupts in our platform description. In case of a WDT underflow, a TLM transaction is sent to the sam3s node module and sent from there to the TLMu interface through socket connections dedicated to this reset event. When the transaction arrives to the Sam3s node, the module triggers a reset of all peripherals. When the TLM transaction arrives in the TLMu interface, the callback method *tlmu_notify_event()* is called to notify TLMu of the reset event. This triggers a call in TLMu to the method *qemu_system_reset_request()*. When TLMu resets the TLM machine, we modified the code only for the case of the ARM processor being used. The initial Stack Pointer (SP) and Program counter (PC) are saved during the first boot in the *env* structure of QEMU (in the *boot_info* structure), to make sure that the right reset vector is fetched in case of resets. So, if the SP and PC have already been saved, at reset, they are copied back from the *boot_info* into the right CPU registers.

2.2.2.5 Serial Peripheral interface (SPI)

The SPI is a synchronous serial data link used to communicate with external devices. It is essentially composed of a shift register that serially transmits bits to another SPI device. When two SPIs are connected, there is a master SPI which controls the data flow, and a slave SPI, which has its data shifted in and out by the master SPI. The slave device is selected by the master through the slave select signal (NSS). An SPI bus is composed of two data links and two control lines:

- The Master Out Slave In (MOSI) data line sends the data shifted from the Master's shift register into the Slave's shift register
- The Master In Slave Out (MISO) data line transfers data from the slave to the master.
- The Serial Clock control line is driven by the master to regulate the flow of data bits
- The slave select (NSS) control line allows slaves to be turned on and off by the master

The serial clock is generated by the master SPI by dividing the Master clock's frequency by a value between 1 and 255. The SPI can use the peripheral DMA controller to reduce processor overhead. The DMA is not modeled as a separate module, but its mechanism is implemented, and the SAM3S's SPI possesses registers that are meant to be used specifically for DMA. In our use case of the SAM3S, this feature is used when a frame must be written or read from the radio transceiver, which is connected to the SoC through its SPI. Instead of the processor ordering a data read or write for every byte of the frame, there is only one order given to the SPI from the processor, then the data read or write is automated using DMA and a pointer indicating the memory space where to read or write the transferred bytes, as well as the length of the transfer in bytes. When a data transfer is initiated as the master SPI selects a slave, the first byte to be transferred serially on the MOSI data line is a command coded on 3 bits followed by a register address coded on 5 bits. The two commands that are supported by the SPI are *read* and *write* which target either a register or the frame buffer.

2.2.2.6 Timer counters

The SAM3S's timer counter peripheral possesses 6 identical 16-bit counter channels. Each channel can be independently programmed to do a wide range of functions including frequency measurement, event counting, interval measurement, pulse generation, delay timing and pulse width modulation. The 6 channels have 2 common control registers, one to enable and disable several channels at a time, and one to select the external clock input. Each channel also possesses individual control registers. There are 2 modes of operation for each channel. The capture mode provides measurement on signals and the waveform mode provides wave generation. Different types of trigger can reset or start the counters. The individual channels can count up to either 0xFFFF or a predefined value, or count down. There are 3 compare registers for each channel. Reaching the value of a compare register may trigger an interrupt if enabled, reset the timer or change the count from up to down.

There are 8 types of interrupt that can be sent from each timer channel:

- Counter overflow
- Load overrun
- RA, RB and RC compare (RA, RB and RC are 3 registers that can have their values set and compared with the current timer value)
- RA and RB loading
- External trigger

The main features that are used in our model is the countdown from a specified value to 0 to schedule certain protocol functions, when the counter reaches 0, if the interrupt is sent (if enabled).

2.2.3 The mechanism to relay HW Interrupts to the NVIC

The interrupt mechanism was partially implemented by TLMu, but there was no documentation on how to use interrupts. TLMu provides callback functions for events, one of the events is the interrupt event. Data is sent back to QEMU as this event notification function is called, but there was no indication to what this data should contain.

The Cortex-M3 processor integrates a configurable Nested Vectored Interrupt Controller (NVIC) which supports low latency interrupt controlling. A model of the NVIC is provided by QEMU which is used for the Stellaris platform. This model is used to add an NVIC in the TLM-machine (which is the TLMu platform name in QEMU), when the modeled CPU is an ARM Cortex-M3. A handler method for interrupt requests in the TLM-machine has also been developed.

In this section, we will explain how interrupts are produced by the modeled SAM3S peripherals, transmitted to QEMU and finally executed.

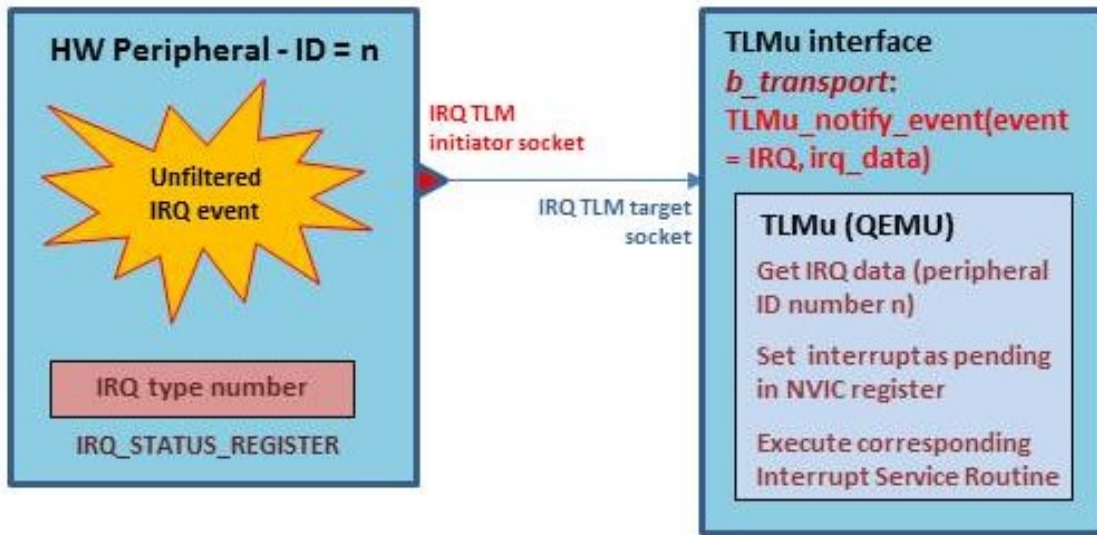


Figure 2.10: Simplified description of an interrupt request transmission from a HW peripheral modeled in as a SystemC module to the CPU modeled by QEMU.

As it is illustrated in Figure 2.10, when an event occurs in a peripheral of the SAM3S platform modeled in SystemC, if this event is supposed to cause an interrupt which is enabled, the bit corresponding to that interrupt is set in the peripheral's status register. Peripherals that can produce interrupts have an interrupt line connecting them to the TLMu module. This line is modeled a TLM transaction sent from the peripheral to the TLMu module containing the Cortex-M3 CPU. The TLMu SystemC module works as the interface between QEMU and SystemC.

Upon reception of the interrupt TLM transaction, the *b_transport* method that is called in the TLMu module extracts the payload from the TLM transaction and calls the *notify_event* callback method, passing as parameter the interrupt type of event and the data extracted from the TLM transaction. The data payload of the TLM transaction is set by the peripheral sending the interrupt, it contains information on the peripheral's ID number.

In the Cortex-M3, there are several sets of two 32 bit registers in QEMU, containing information on the interrupts, mainly, whether there are enabled, pending, active and their priority. Each bit corresponds to an interrupt number, so there are 64 possible interrupt

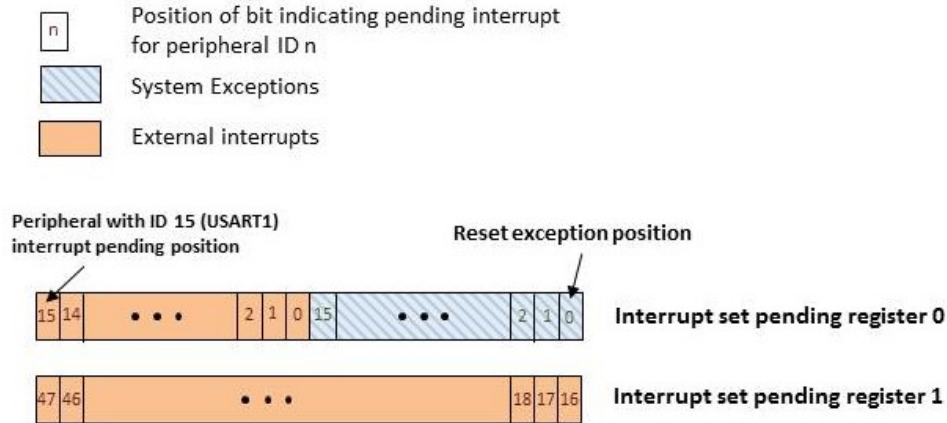


Figure 2.11: The two 32 bit set interrupt pending registers in QEMU.

sources. When QEMU receives the interrupt notification, it calls the method *qemu_set_irq* which sets the bit corresponding to the peripheral number to true on the pending interrupt registers of the NVIC (depicted in Figure 2.11). The method also sets the exception of QEMU's environment to hardware exception. The exception is taken at the end of the execution of the current translation block which has a variable length, possibly introducing a small latency issue. This could be solved by forcing QEMU to execute one instruction at a time instead of a whole block, but at the price of a slower execution. When the exception is taken, if no interrupt of higher priority is set as pending or active, the interrupt that was sent from the peripheral is set as active and no longer pending. The values of the program counter pointer and stack pointer from QEMU's environment are saved and the new program counter is calculated. The pointer value is fetched at the vector table address offset by the interrupt ID number and the corresponding Interrupt Service Routine (ISR) is executed. This routine generally accesses the interrupted peripheral to find out which type of interrupt was generated (read access to the peripheral's interrupt status register), and proceed accordingly.

2.2.4 The AT86RF233 Radio transceiver model

The radio transceiver used to complement the SAM3S microcontroller is the Atmel AT86RF233. It is a low power, 2.4GHz transceiver for ZigBee, RF4CE, IEEE 802.15.4, 6LoWPAN, and ISM applications. It has been modeled as a SystemC module based on its datasheet [20].

The transceiver communicates with the SAM3S microcontroller through the SPI bus, as well as a few pins controlled by the PIO peripheral of the SAM3S. The RF chip is the SPI slave while the SAM3S is the SPI master. The interface between the microcontroller and the transceiver is depicted in Figure 2.12.

The SPI data exchange is initiated by the microcontroller through the SPI select signal. The first exchanged byte is always a command from the microcontroller, which can be to read or write a specific register, the frame buffer or the SRAM.

The transceiver can be considered as a state machine. The transition between states can be triggered by a write to the TRX_CMD register, or through signals received on the SLP_TR pin or the RST pin.

The transceiver's state machines can be found in Appendix C. The different possible states of the AT86RF233 transceiver are:

- P_ON: Power on to perform an on-chip reset.

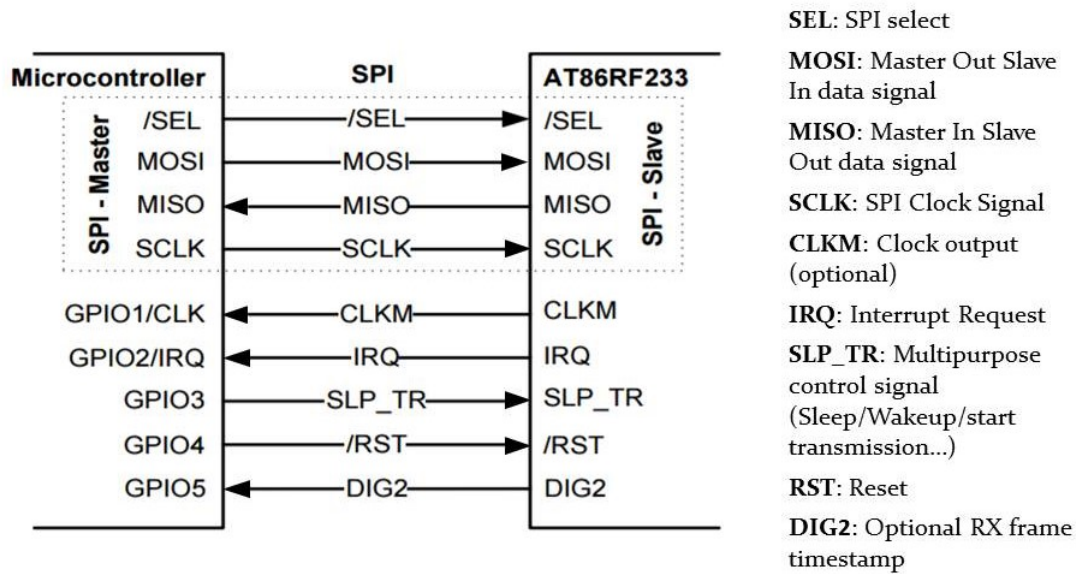


Figure 2.12: Microcontroller to AT86RF233 Interface [20].

- **BUSY_RX (RX_ON):** Listen and Receive State
- **BUSY_TX:** Data transmission state
- **TRX_OFF:** In this clock state, the crystal oscillator is running and the master clock is available if enabled
- **PLL_ON:** The PLL state enables the analog voltage regulator and the PLL frequency synthesizer to lock on the right channel's frequency.
- **SLEEP:** The SLEEP state is used when radio transceiver functionality is not needed.
- **PREP_DEEP_SLEEP:** Deep Sleep Preparation State
- **DEEP_SLEEP:** Deep Sleep State when the entire transceiver is disabled, only the SLP_TR pin is monitored.
- **RESET:** The reset state performs an on-chip reset.
- **BUSY_RX_AACK:** Receiving frame with Automatic Acknowledgement
- **BUSY_TX_aret:** Transmitting frame with Automatic Frame Retransmission and CSMA-CA Retry
- **RX_AACK_ON:** Ready to receive with Automatic Acknowledgement
- **TX_aret_ON:** Ready to transmit with Automatic Frame Retransmission and CSMA-CA Retry

A transceiver pin is connected with the microcontroller through the PIO peripheral to signal interrupt requests to the CPU. The different possible sources of interrupt of the transceiver are:

- A low battery

- Frame buffer access violation
- Received frame address matching
- End of CCA (Clear Channel Assessment) or ED (Energy Detection) measurement
- End of frame reception or transmission
- The start of PSDU reception (frame data units)
- PLL unlock
- PLL lock

Each of these events can trigger an interrupt request if the interrupt for this event is enabled. The interrupt status register is read by the microcontroller to know which interrupt caused the request.

The transceiver's frame buffer contains the frame to be sent or received, depending on the transceiver's state. When a frame is ready to be sent, a countdown is initiated corresponding to the time it would take for the radio antenna to produce all of the frame's symbols, knowing that a symbol is emitted in a 16 μ s time slot. At the end of the countdown, a TLM transaction is sent out from the *antenna_out* TLM initiator socket. The TLM transaction used to transmit a frame mainly contains a data pointer, pointing to an address where the bytes from the frame buffer have been copied, the length field contains the frame length and the address field contains the ID number of the sender node. Frames from other simulated nodes are received through an *antenna_in* target TLM socket. At the reception of the frame, if the radio is in a reception state, a frame filtering process takes place (if enabled) to check that this frame is destined to this transceiver's node by checking the frame type or destination address. The frame's CRC (Cyclic Redundancy Check) is also checked by the filter. If the frame passes the filter, an interrupt is sent back to the microcontroller and the bit corresponding to the end of frame reception interrupt is set to true in the transceiver's interrupt status register.

The Atmel AT86RF233 incorporates a two bit random number generator by observation of noise, which can be used by protocols using CSMA-CA medium access to generate random back-offs. These 2 bits are contained in the RND_VALUE field of the transceiver's PHY_RSSI register (a multi-purpose register). This was modeled by calling the *rand* C library function when the PHY_RSSI register is accessed. The two last bits of the pseudo-random value returned by *rand* are set in the RND_VALUE field. It is worth noting that not initializing the *rand* function does not allow to repeat simulations and get identical results. For example, if the random back-off before sending a frame changes between two simulation runs, the frames will not be sent at the same time, and one of them may collide with another frame or not be received by the node it was destined for, thus modifying the simulation results. This is not ideal to repeat simulations that caused transient bugs. Therefore, the random generator accessed through *rand* is initialized using the *srand* C library function and passing a seed as argument. The seed is set to the value of the node's ID number, which is set as the SAM3S model is instantiated.

2.2.5 Synchronization issues and solutions

Several issues arose from the synchronization method in TLMu, which is handled by the *sync_time* function in the SystemC source code files.

The *sync_time* method is called by the original TLMu in the following cases:

- At the end of the *cpu-exec* method loop (after the execution of a translation block by QEMU)
- During a read or write TLM bus access to SystemC modeled memories
- During a read or write direct memory access to SystemC modeled memories

2.2.5.1 Lack of precision in the conversion of the number of instructions executed to the corresponding time

The first issue had to do with the deduction of the time that has passed in QEMU by using the number of instructions executed, and incrementing the quantum keeper's *Local_time*. As seen in Section 2.1.4.3, in the original TLMu code, the time that has passed since the last *sync_time* function call is calculated using the value of QEMU's *vm_clock* at the time of the function call as seen in Algorithm 1.

Algorithm 1 ORIGINAL SYNC_TIME FUNCTION CALL WITH TLMU_TIME (VM_CLOCK) AS PARAMETER.

```

1: procedure SYNC_TIME(tlmu_time)
2:   delta_ns = tlmu_time - last_sync
3:   last_sync = delta_ns
4:   delta_ns = delta_ns ÷ 2
5:   delta_ns = delta_ns × Speed_factor
6:   Quantum_keeper → increment_local_time_by_delta_ns
7:   if Quantum_keeper → Need_sync then
8:     Quantum_keeper → sync
9:   end if
10: end procedure

```

The issue is that *delta_ns* may become a decimal number after being divided by 2 and multiplied by the speed factor, while the quantum keeper's *inc(delta_ns)* method uses an integer as parameter. Let us illustrate this issue with the following example.

The *Speed_factor* is set according to the platform's frequency. The CPU's frequency of the SAM3S platform is 32 MHz. In this case, the duration of a CPU cycle (also called *Speed_factor*) is $1 \div 32\text{MHz} = 31.25\text{ ns}$.

If the time since the last call of the *sync_time* function is 4 ns, knowing that the *vm_clock* is increased in TLMu by 2 ns for every instruction executed (see Section 2.7), by dividing this time by 2, we deduce that 2 instructions have been executed. Assuming that an instruction takes 1 cycle in average, the time that has passed since the last *sync_time* function call is $2 \times 31.25\text{ ns} = 62.5\text{ ns}$. This is the value of *delta_ns* that will be used to increase the local time of the Quantum Keeper. The problem is that 62.5 is not an integer, so the value that will be added to the Quantum Keeper's local time is actually 62 ns, and the 0.5 extra ns are not taken into account. In this case this corresponds to a 0.8 % time inaccuracy. This might not seem to be a big inaccuracy, but in real time systems, the timing of the platforms must be very precise as their tasks can be very time sensitive. This is why the crystals generating the clock signals on the hardware platform have a precision of the order of the thousandth of percent of their frequency.

To solve this problem, we have added a variable called *decimal_part*, that saves the decimal part that has not been taken into account from one function call to another, adding up the extra decimal parts until an integer is reached. The integer is then added to

the Quantum Keeper's *Local_time* as seen in Algorithm 2. The *floor* method is used to truncate a given number to isolate its integer and decimal part.

Algorithm 2 MODIFIED SYNC_TIME FUNCTION CALL WITH TLMU_TIME (VM_CLOCK) AS PARAMETER.

```
1: procedure SYNC_TIME(tlmu_time)
2:   delta_ns = tlmu_time - last_sync
3:   if delta_ns > 0 then
4:     last_sync = delta_ns
5:     delta_ns = delta_ns ÷ 2
6:     delta_ns = delta_ns × Speed_factor
7:     decimal_part = decimal_part + delta_ns - floor(delta_ns)
8:     if decimal_part > 1 then
9:       delta_ns = floor(delta_ns) + floor(decimal_part)
10:      decimal_part = decimal_part - floor(decimal_part)
11:    end if
12:    Quantum_keeper → increment_local_time_by_delta_ns
13:    if Quantum_keeper → Need_sync then
14:      Quantum_keeper → sync
15:    end if
16:  end if
17: end procedure
```

2.2.5.2 Removing needless calls to the `sync_time` method

In order to optimize the execution time, we have added a condition to check that the time that has passed since the last call to the `Sync_time` method (`delta_ns`) is greater than 0. Testing this condition avoids doing all the calculations on `delta_ns` and calling synchronization methods from the Quantum Keeper when it would be needless.

2.2.5.3 Bad synchronization leading to platform malfunctions

We noticed that the platform model was sometimes malfunctioning, the RF transceiver module would receive nonsensical data from the microcontroller through the SPI bus, which caused it to raise an error message. After tracing the origin of the problem, we realised it was linked to the SPI's slave select signal, which would not toggle properly because of a synchronization issue.

The communication through the SPI bus between the microcontroller and the transceiver is controlled by the Slave Select signal, which is an output signal of the PIO peripheral of the microcontroller. This signal is controlled by the protocol stack being executed by TLMu. The slave (transceiver) is selected when the Slave Select signal (NSS pin) is set to low by the PIO. The slave then knows that a new communication sequence is being initiated and the next byte to be received from the Master is a command. When the SPI communication is over this signal is set back to high.

A problem happened when a new SPI communication was initiated right after another one, or during another one in the case of an interrupt. In these cases, the NSS signal was low as a communication was already in progress. The new communication was signaled by raising the NSS signal to high and then resetting it back to low. This was modeled in SystemC by using the `write` method on the NSS signal, then the `slave_selected` method of the transceiver was supposed to be called as it is sensitive to the NSS signal. However, because the change from low to high and back to low on NSS happens in a short time, there was no synchronization in between the different `write` method calls on the signal. This means that the time in SystemC did not progress. The toggle of the NSS signal was not taken into account and the `slave_selected` method was not called, so the transceiver was not notified that the bytes received from the microcontroller were for a different communication sequence, making the data received by the transceiver nonsensical.

This problem was caused because the `sync_time` method being called does not automatically mean that the time will be synchronized between TLMu and SystemC. It only synchronizes if the Quantum Keeper's `Need_sync` method returns true. This happens if the Quantum Keeper's `Local_time` has become greater than its `Local_quantum`.

To solve this problem, a synchronization was forced in the case of a memory access to the PIO peripheral. This is done by calling directly the `sync` method of the Quantum Keeper without calling its `need_sync` method first. The synchronization allows the SystemC time to progress between the calls of the `write` method on the NSS signal, causing its toggle to be taken into account. Consequently, the transceiver's SPI interprets the bytes received from the microcontroller correctly.

2.2.6 Conclusion

In this chapter, we have described how we modeled a functional node model capable of sending and receiving frames as TLM transactions. The model we developed was for the Atmel SAM3S and the AT86RF233 RF transceiver, but the same method can be used to develop a model for a different hardware platform.

The method used by TLMu to co-simulate QEMU and SystemC (making QEMU into a shared object) makes it possible to use several instances of QEMU in the same simulation process. Therefore, several nodes with full platform models based on QEMU can be instantiated to form a network.

Bibliography

- [1] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the ground up*, vol. 71. Springer Science & Business Media, 2011. xi, 2, 31, 51, 52
- [2] F. Ghenassia *et al.*, *Transaction-level modeling with SystemC*. Springer, 2005. 53
- [3] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, 2005. 2, 55
- [4] “Official qemu github mirror,” 2017. github.com/qemu/qemu. 55
- [5] Y. Roch, “Qemu: Visite au cœur de l’émulateur,” *GNU Linux Magazine France*, no. 147, 2012. 56
- [6] M. Montón, J. Carrabina, and M. Burton, “Mixed simulation kernels for high performance virtual platforms,” in *Specification & Design Languages, 2009. FDL 2009. Forum on*, pp. 1–6, IEEE, 2009. xi, 56, 57
- [7] M. Montón, J. Engblom, and M. Burton, “Checkpointing for virtual platforms and systemc-tlm,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 1, pp. 133–141, 2013. 56
- [8] M. Gligor, N. Fournel, and F. Pétrot, “Using binary translation in event driven simulation for fast and flexible MPSoC simulation,” in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pp. 71–80, ACM, 2009. 57
- [9] TIMA, “Rabbits : an environment for fast and accurate mpsoC simulation,” 2017. tima.imag.fr/sls/research-projects/rabbits/, last accessed in February 2017. 57
- [10] Antfield, “Virtual prototyping adapted to your needs,” 2017. www.Antfield.fr, last accessed in February 2017. 57
- [11] T.-C. Yeh, G.-F. Tseng, and M.-C. Chiang, “A fast cycle-accurate instruction set simulator based on QEMU and SystemC for SoC development,” in *MELECON 2010-2010 15th IEEE Mediterranean Electrotechnical Conference*, pp. 1033–1038, IEEE, 2010. 57
- [12] T.-C. Yeh and M.-C. Chiang, “On the interfacing between QEMU and SystemC for virtual platform construction: Using DMA as a case,” *Journal of Systems Architecture*, vol. 58, no. 3, pp. 99–111, 2012. 57
- [13] G. Delbergue, M. Burton, F. Konrad, B. Le Gal, and C. Jégo, “Qbox: an industrial solution for virtual platform simulation using qemu and systemc tlm-2.0,” in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016. 57, 126

- [14] B. Koppelman, B. Messidat, M. Becker, C. Kuznik, W. Mueller, and C. Scheytt, "An Open and Fast Virtual Platform for TriCore™-based SoCs Using QEMU," in *Design and Verification Conference, DVCON Europe*, March 2014. 58
- [15] "Transaction level eMulator (TLMu)," 2011. edgarigl.github.io/tlmu/. Accessed on Sep. 27, 2016. 2, 58
- [16] C. Barnes, F. Verdier, A. Pegatoquet, D. Gaffé, and J.-M. Cottin, "Wireless sensor network protocol property validation through the system's simulation in a dedicated framework," in *Signal Processing and Communication Systems (ICSPCS), 2016 10th International Conference on*, pp. 1–9, IEEE, 2016. 59
- [17] "Sam3s arm cortex-m3 microcontroller," 2015. www.atmel.com/products/microcontrollers/arm/sam3s.aspx. Accessed on Sep. 27, 2016. 61
- [18] C. Barnes, J.-M. Cottin, F. Verdier, and A. Pegatoquet, "Towards the verification of industrial communication protocols through a simulation environment based on qemu and systemc," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pp. 207–214, ACM, 2016. 61
- [19] ARM, "Arm and thumb-2 instruction set," 2008. infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf. 62
- [20] "At86rf233," 2014. www.atmel.com/devices/AT86RF233.aspx. xi, 67, 68

Chapter 3

SNOOPS: Sensor Network simulatiOn for the validatiOn of Protocol implementations

Contents

3.1 What are the requirements for SNOOPS?	78
3.2 Developing a model of the network	80
3.2.1 Instantiating several nodes	80
3.2.2 The radio communication medium module	82
3.3 Traffic generation	83
3.3.1 Abstracted nodes	83
3.3.2 The Frame Generator: Re-injecting frames in simulation from real exchanges	84
3.4 Modeling and Verifying protocol properties	90
3.4.1 The validation of protocol properties	90
3.4.2 What is the role of the observer module?	92
3.4.3 Modeling protocol properties	92
3.4.4 Inserting properties into the observer module	94
3.4.5 Checking properties during simulation	97
3.5 Simulation output	99
3.5.1 The terminal output	99
3.5.2 Analyzing the simulation output with Wireshark	99
3.6 Conclusion	102
Bibliography	102

3.1 What are the requirements for SNOOPS?



Figure 3.1: SNOOPS: A toolbox for WSN protocol verification and validation

This section deals with a simulation environment that can be used as toolbox for the needs related to WSN protocol verification and validation. This simulation environment is called SNOOPS: Sensor Network simulatiOn for the validatiOn of Protocol implemen-tationS.

Despite the objective being to observe the execution of at least one protocol stack on one fully modeled node, this node needs to have an environment to interact with. The node under test should be able to send and receive packets from other nodes. That is why we need to develop a network simulator.

As we have stated in Chapter 1, these are the requirements for SNOOPS:

1. **Validating the compiled protocol as deployed on real nodes:** the bugs in the communication between sensors might stem from node platforms hardware limitations (memory, processing, bandwidth, available testing capabilities, etc). Therefore, an accurate node model is necessary if we desire to simulate the execution of the protocol stack binary code and reproduce bugs more realistically. In order to validate a stack's implementation as an executable file, we need to develop a platform emulator in SNOOPS. The description of how we developed this platform emulator was provided in Chapter 2. Associating QEMU and SystemC offers a good flexibility to model a wide range of hardware platforms.
2. **Visibility on the state of the protocol's execution** and the states of memories and registers of the hardware platform running the protocol. On physical testbeds, hardware debugging is intrusive as it modifies the execution timing of the software being inspected, causing Heisenbugs (a fault that stops causing a failure or that manifests

differently when one attempts to probe or isolate it [1]). Indeed, trying to investigate a failure can influence process scheduling in such a way that the failure does not occur again. As a consequence, starting a debug session may prevent the protocol to work at all leaving no chance to catch bugs affecting real-time features. At the opposite, when debugging using models, you are able to observe (at source code level) the unmodified real-time behaviors of the software (with respect to a simulated network environment). Node emulation offers the possibility to use debuggers to visualize the internal workings of the hardware node model, as well as debuggers to trace the stack's execution on the node model.

3. Validating protocol properties over a **wide range of scenarios**: the goal is not only to verify that a protocol stack works correctly under normal conditions. A reliable stack must be able to maintain a minimal quality of service in corner case scenarios, under a wide range of environmental conditions and for different types of topology. We must be able to simulate scenarios that are unlikely to happen to test the limits of the stack's robustness. A software test-bench has the power to express situations that are not easily reproducible in field tests.
4. **Reproducing testbed experiments** for which the source of bugs can not be determined. It may happen that while experimenting on a physical testbed, a transient bug appears, and the experiment may not be repeatable due to lack of control over the environmental conditions, or the sometimes random nature of medium access by the nodes (e.g. random backoffs before sending certain frames for CSMA-CA MAC protocol). When a transient bug happens, the only information obtained on this bug is recorded in a log file of the frames exchanged, recorded in a network analyzer by using a sniffer. This log file can be used in simulation to attempt reproduce the testbed experiment that led to the bug, and take advantage of the possibility to use debuggers in simulation to pinpoint the bug's source.
5. Being an **Open Source** solution: developing an open source software means that potential users may access the software's source codes more easily, and may help creating a community to contribute to its development and insure the software's durability.

In this chapter, we focus on how to create a wide range of scenarios, reproduce testbed experiments and test protocol properties in simulation.

Here are the requirements to create a wide range of possible scenarios:

- Instantiate several nodes
- Create different topologies
- Model the apparition and loss of connections between nodes (interference, obstacles)
- Model the radio channel
- Generate packets that might come from different interfering networks, or even packets that would be meant to attack or destabilize the network

By instantiating nodes as classical SystemC modules, it is easy to implement as many nodes as we want and to add them in the environment. However, a module that handles

the network's topology, the links between nodes, as well as the radio channel must be created.

SystemC can model systems at various levels of abstraction. Therefore, SystemC can also be used to model nodes at a higher level of abstraction than the SAM3S node models. These nodes can be used to produce specific frames dictated by the user or to produce frames based on a higher level of abstraction of the stack.

Finally, to reproduce testbed experiments, data from the log files recorded during the experiments must be extracted. A common format of log file for network analyzers is pcap (packet capture), which will be detailed in Section 3.5.2.2. A module, capable of interpreting pcap logs, has been developed to extract the data on the different packets exchanged in the network and figure out how to replay the scenario. This will be described in Section 3.3.2.

3.2 Developing a model of the network

3.2.1 Instantiating several nodes

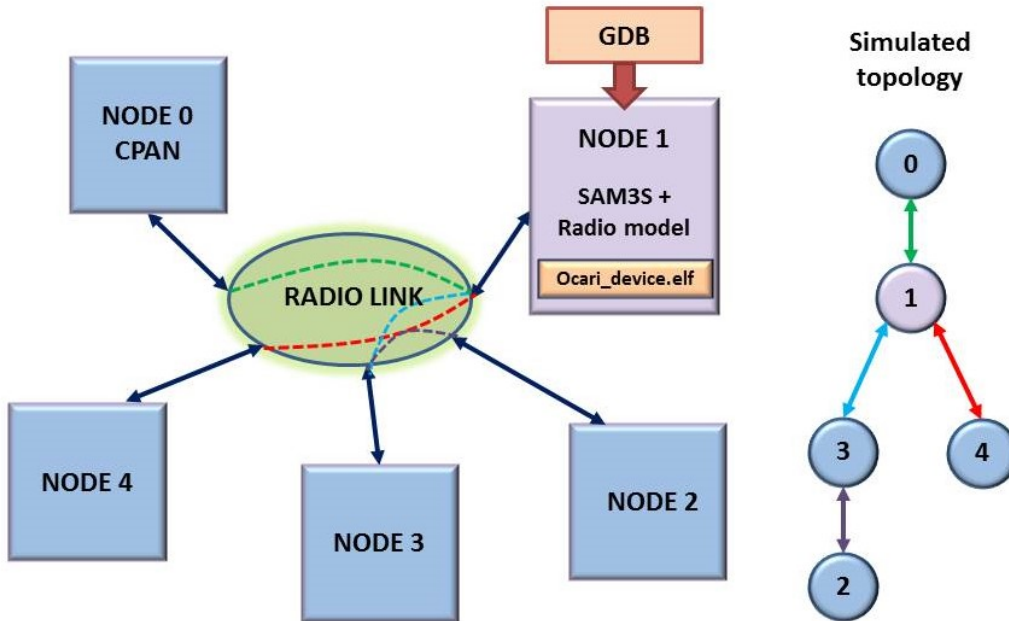


Figure 3.2: Block diagram of different node models interconnected through the Radio Link module.

As we have seen in the previous chapter, a model of a sensor node is developed using SystemC with TLMu. The node model is a SystemC module, so it can be instantiated at will in the simulation set-up, using different parameters for different types of nodes. To simulate a WSN, at least one of the simulated nodes must be a network coordinator (CPAN). Figure 3.2 shows a block diagram of several simulated nodes instantiated in the same network, and Figure 3.3 shows the code instantiating these nodes in the main source code file. Figure 3.3 is the code for SNOOPS's Top module, called from the `sc_main()` method. It shows the instantiation of the Radio Link module (`radio_observer_module`) which is described in Section 3.2.2, then the CPAN is instantiated and connected to the Radio Link

module through its *Antenna_in* and *Antenna_out* TLM sockets. The *gdb_port_table* contains the port numbers to connect remote debuggers to the device nodes. If the port number is NULL, no remote debugger is connected. Then several nodes are instantiated in the same manner. The number of device nodes to be instantiated is pre-set through a constant called *DEVICE_NR*. The parameters that can be set through a node's instantiation are:

- Its name
- The CPU's frequency
- The path to the executable file of the protocol stack
- A TCP port number to attach a remote debugger to trace the execution of the protocol stack in QEMU
- Whether the node is a network coordinator or not
- The node number for debugging purposes

If the user defines *TRAFFIC_GEN*, the traffic generator used to replay testbed scenarios is instantiated and connected to the Radio Link module. Two methods are used to retransmit the clock signals from the *sc_main* to the different devices instantiated. Finally, the value of the global quantum is set to define the default time between QEMU and SystemC synchronizations.

```
Top(sc_module_name name)
{
    radio_observer_module = new radio_observer("observer");

    sam3s_CPAN = new sam3s_node("CPAN", 32*TLMU_MHZ, "../ocari_src/ocari_CPAN.elf", "tcp::1234", true, 0);
    sam3s_CPAN->Radio->antenna_in(radio_observer_module->to_cpan_sk);
    sam3s_CPAN->Radio->antenna_out(radio_observer_module->from_cpan_sk);

    const char* gdb_port_table[DEVICE_NR];
    for (int i = 0; i < DEVICE_NR; i++)
        gdb_port_table[i] = NULL;

    for (int i = 0; i < DEVICE_NR; i++)
    {
        sam3s_DEVICE[i] = new sam3s_node("MOBILE", 32*TLMU_MHZ, "../ocari_src/ocari_DEVICE.elf", gdb_port_table[i], false, i+1);
        sam3s_DEVICE[i]->Radio->antenna_in(*(radio_observer_module->to_node_sk[i]));
        sam3s_DEVICE[i]->Radio->antenna_out(*(radio_observer_module->from_node_sk[i]));
    }

#ifdef TRAFFIC_GEN
    traffic_gen = new pcap_traffic_gen("PCAP_traffic_gen");
    traffic_gen->ant_in_sk(*(radio_observer_module->to_node_sk[0]));
    traffic_gen->ant_out_sk(*(radio_observer_module->from_node_sk[0]));
    traffic_gen->Radio_clk(GeneratorClockSig);
#endif

    SC_METHOD(copy_pmc);
    sensitive << MasterClock_in;

    SC_METHOD(RadioCLK_copy);
    sensitive << RadioMCLK;

    m_gk.set_global_quantum(sc_time(1000, SC_NS));
}
```

Figure 3.3: Top SystemC module in SNOOPS: instantiation of different nodes, the traffic generator, and connections to the radio module.

The simulated nodes do not all start working at the same time, as it is unlikely that wireless sensor nodes will all be deployed at the same time in real life. Letting them all start at the same time may cause the nodes to all answer to the beacon frame of the CPAN

at the same time to join the network. A delay proportional to the node number is introduced in the TLMU interface to freeze the CPU thread, hence stopping the stack's execution. The time does not advance in SystemC either. Indeed, the SystemC time will be increased only when QEMU starts executing instructions and requests a synchronization. Therefore, the protocol's code's execution to initialize the platform is also delayed, and the node appears to be off.

3.2.2 The radio communication medium module

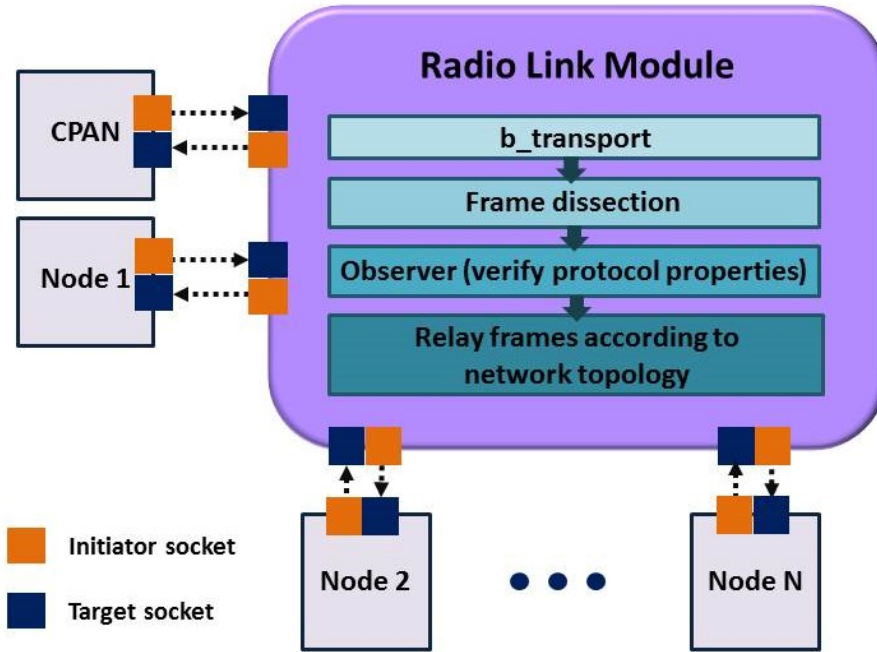


Figure 3.4: The radio link module.

To interconnect the SystemC node models, a Radio Link module is developed in SystemC. This module handles the network's topology and the links between the nodes, it also records the simulation results in a log file.

As seen in Figure 3.4, the Radio Link module connects to the SystemC node models using TLM sockets: two TLM sockets are instantiated for each simulated node, to be connected with the *Antenna_in* and *Antenna_out* TLM sockets of this node. When a TLM transaction arrives in a target socket of the Radio Link module, the *b_transport* method is called and the data contained in the transaction is treated. The address field of the transaction contains the sender node's ID.

The frame is then dissected to display information, and is analyzed by an observer module. It analyzes the correctness of the sequence of exchanged frames between nodes in relation to the protocol's specifications. This module will be described in more details in Section 3.4.

In order to simulate the network topology, the user has to define in the Radio Link module to which node a packet received from a specific node may be retransmitted. This is done by using the node's Id extracted from the transaction. This mechanism is represented in Figure 3.2, where we can see an example of topology (pictured on the right side of the Figure) modeled by the radio link:

- Frames from Node 0 are transmitted to Node 1
- Frames from Node 1 are transmitted to Nodes 0, 3 and 4
- Frames from Node 2 are transmitted to Node 3
- Frames from Node 3 are transmitted to Nodes 1 and 2
- Frames from Node 4 are transmitted to Node 1

Using the *sc_time_stamp* function, the user can set the timing at which links are created or lost between nodes. It is also possible to introduce a transmission probability on the Radio Link's relay of the packet to a specific node, by setting a condition on the retransmission of the TLM transaction.

The Radio Link handles the simulation output in the terminal from which simulation was launched. A log file of the simulation results is also recorded. These features are further explained in Section 3.5.

3.3 Traffic generation

Frames are generated in SNOOPS for two purposes. First, to generate frames based on a behavioral model of the protocol under test, to simulate a larger number of nodes in the simulation environment without slowing down the simulation's execution by using full node models. This is done by modules called Abstracted Node. The second objective is to generate frames based on captures from real exchanges between physical nodes, which is done by a module called the Frame Generator.

3.3.1 Abstracted nodes

Abstracted nodes are SystemC modules that can produce and receive frames as TLM transactions just as the ones produced by nodes with a fully simulated hardware platform. However, they do not contain a full QEMU-SystemC platform model and they do not run the protocol under test's binary executable file.

3.3.1.1 Abstracted nodes running a behavioral model of the protocol under test

Simple nodes can be programmed to produce frames according to a behavioral model of the protocol.

Different types of frame models (e.g. Beacon, Command) can be saved into buffers by the user before running the simulation. In the case where the simple node is sending frames based on the protocol's specifications, the simple node can be either a coordinator in the network or a simple device. In the case of a coordinator, when the simulation starts, it emits Beacon type frames at constant intervals defined by the protocol until another simulated node replies with an association request frame. If the node is not a coordinator, the emission of a frame is triggered by the reception of a frame from another simulated node after a delay defined by the protocol.

For example, in the OCARI protocol (see Appendix A), a Beacon frame from the network coordinator will trigger a beacon frame to be sent from an abstracted node 10 ms later, and a Hello Data type frame to be sent every other time a Beacon is received. Frame templates are recorded in dedicated buffers. Some fields of the frame templates must be modified before the frames are sent. For instance, the addressing fields in the Abstracted

nodes are modified: the short address is attributed by the network coordinator and the long address is randomly generated.

Replacing nodes with full platform models in the simulation environment by abstracted nodes allows accelerating the simulation's execution. Indeed, the simpler the model is, the faster the execution of the simulation is. For abstracted nodes, less events have to be handled by the SystemC simulation core and there is no need to synchronization SystemC and QEMU, which slows down the simulation's execution.

Only the node which is under observation executing the stack under test really needs to be simulated using a hardware platform emulator. This node can be stimulated either by frames from more abstracted node models or by a traffic generator. It is worth noticing that it is possible to use more than one node emulator, but the simulation's execution time increases with the number of node emulators instantiated.

3.3.1.2 Generating frames to destabilize the network

Abstracted nodes can be programmed for diverse tasks. It is also possible to program a simple node to send specific frames to try and destabilize the nodes running the protocol under test. Here are three examples of frames that can be generated for that purpose:

- Sending frames from a similar protocol (e.g. ZigBee) to the one being tested in order to test if the coordinator integrates the wrong node in its network or if other nodes confuse that node with one of their network.
- Generating different types of frames from various protocols to simulate a noisy environment where the WSN would be deployed around other wireless networks.
- Generating frames to simulate attacks on the network to test the protocol's resilience to such attacks. It is simpler and quicker to test attacks in a simulated environment as the simple nodes are much quicker to program than physical nodes [2]. This is an important feature to develop for the simulator as cybersecurity becomes more and more a major concern in WSN.

3.3.2 The Frame Generator: Re-injecting frames in simulation from real exchanges

During tests on physical testbeds, it may happen that a bug is detected, but its source remains difficult to determine. One of the advantages of WSN simulators over physical testbeds is a better visibility of the code's execution and the node's state. Indeed debuggers can be used to observe the state of the memories, peripheral registers, etc. Guest code debuggers can be used to follow the stack's execution instruction by instruction without being intrusive and modifying the stack's behavior. To our knowledge, there is no WSN simulator that offers to reproduce tests that have been conducted on physical testbeds and for which the source of bugs could not be determined.

We have extended SNOOPS with a feature of traffic generation based on logs from physical testbed experiments. This is done in three steps, as seen in Figure 3.5. First an experimentation is led on a testbed with nodes running the protocol under test. The frames exchanged between nodes are recorded by sniffers and displayed in a network analyzer. The log file of the exchanged frames is recorded by the analyzer. Finally, the log file can be used by SNOOPS to reproduce the scenario.

To keep the scenario as identical as possible to the original one (on the physical testbed), only one node of the topology is simulated. It receives frames extracted from the log file,



Figure 3.5: Steps to reproduce a testbed scenario in SNOOPS.

so that the simulated node receives the same stimuli as the physical node it is representing.

To do so, we developed a *pcap_reader* SystemC module which is capable of extracting packet data from logs recorded in the pcap format. This format is described in more detail in Section 3.5.2.2. The *pcap_reader* module is instantiated in a SystemC module called the Frame Generator, as seen in Figure 3.7. The *pcap_reader* uses libpcap library functions to extract packet data. However, it must ignore (filter out) the frames from the log that correspond to the ones that should be produced by the simulated node, as well as frames that this node would not have received given the network topology. Let us illustrate this mechanism with an example, depicted in Figure 3.6. In this scenario, the node that is related to the bug is the node which has 4 as its short address. When playing out the scenario in SNOOPS, node 4 is emulated. When the frame generator extracts frame from the recorded log, it filters out the frames that originated from node 4 in the log, as the emulated node 4 will be generating its own frames. It also filters out frames originating from nodes 1 and 3, given that with this scenario's topology, node 4 does not receive frames from them. Finally, the Frame generator will generate the frames originating from nodes 2 and 5, so that node 4 is stimulated by the same frames it was stimulated by during the test on the testbed.

The long and short address of the node that is simulated, as well as its neighbors' addresses (found in the log file) must be indicated by the user for the *pcap_reader* to determine which frames to ignore or send. Moreover, the *pcap_reader* also has to ignore frames that might be recorded in the log that are not from a WPAN network (e.g. Ethernet packets). The *get_next_frame* method gets the next frame recorded in the pcap file that is not ignored or filtered.

When testbed scenarios are reproduced in SNOOPS, two different practical cases can happen:

- The network coordinator (CPAN) is not the simulated node, its frames are generated based on the pcap log file (if it is a neighbor of the simulated node)
- The CPAN is the simulated node

When the simulation starts, the Frame Generator goes through several initialization phases which are different if the CPAN is the simulated node or not. These phases can be seen in the state machines depicted in Figure 3.8. Each state machine has 4 states, which are the different stages of initialization of the Frame Generator for the 2 different practical cases (CPAN emulated or not).

When the emulated node is not the CPAN, the user must indicate the frame number of this node's association request in the log file. All the frames preceding it are ignored, except for the neighbors' last beacon frames before the association, as they are used to

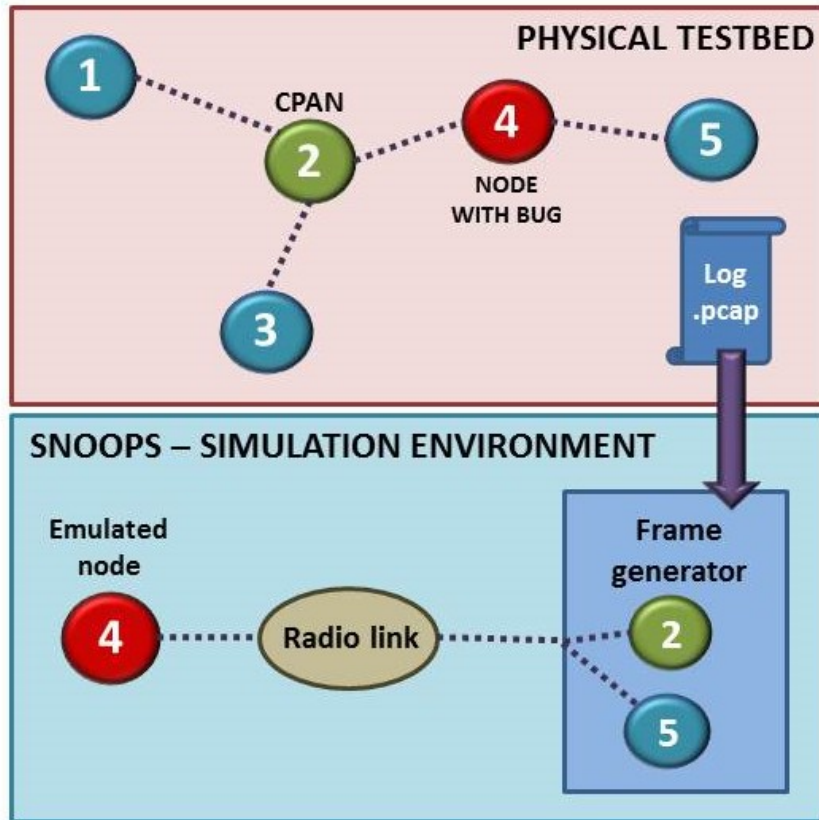


Figure 3.6: Example of testbed scenario reproduced in SNOOPS.

stimulate the emulated node to trigger its association request. The frame generator sends the association response.

When the emulated node is the CPAN, the frame generator runs through each frame of the pcap log file until it reaches an association request frame from one of the CPAN's 1-hop neighbors. When the emulated CPAN is done initiating the emulated hardware platform, it starts sending beacons. After the reception of the first beacon, the frame generator sends the first association request it found in the log, then waits for the CPAN's association response.

Once the first node association is done, in both cases, the Frame Generator switches to the RUN_PCAP state, and generates all the frames extracted from the pcap file which are not ignored or filtered. The time intervals between generated frames are calculated as the difference between the corresponding timestamps (found in the pcap file).

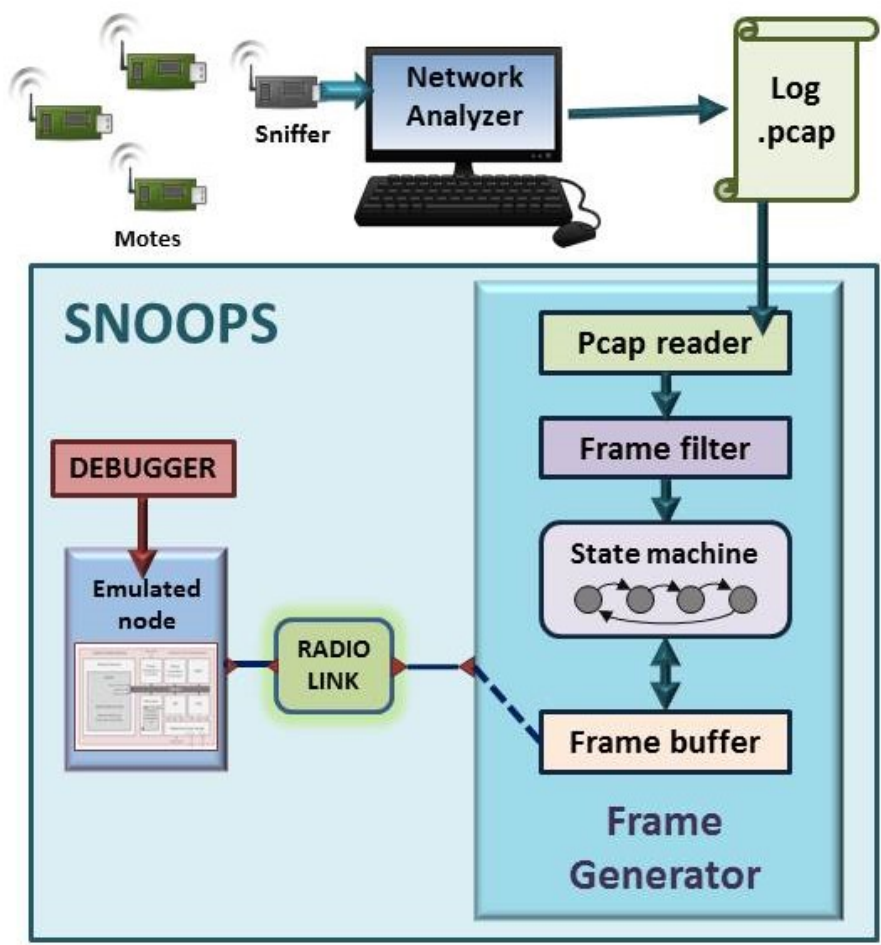
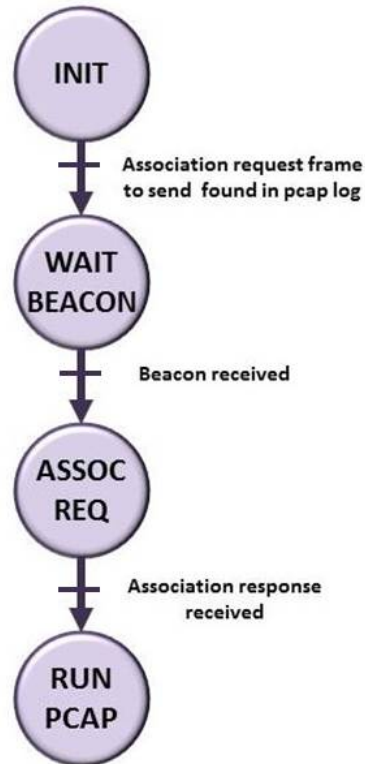


Figure 3.7: Block diagram of the Frame Generator communicating with a simulated node.

Figure 3.7 illustrates the mechanism of the Frame Generator. The state machine controls the pcap reader module to extract the next frames in the log. It loads frames to send in a frame buffer and sends them to the Radio Link module using TLM transactions. The Radio Link relays frames from the generator to the emulated node.

Figure 3.8 shows the Frame Generator’s state machines for both cases where the CPAN is simulated or not. The detail of each state and transition is explained in the following table. The two first columns correspond to the states of the first state machine of Figure 3.8, which are the initialization phases when the CPAN is the emulated node. The two last columns of the table correspond to the second state machine, representing the case where the CPAN is not emulated.

The CPAN is the
simulated node



CPAN frames are generated
based on the log file

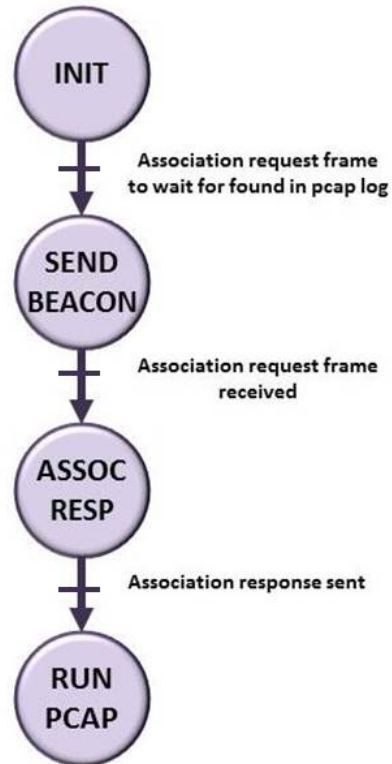


Figure 3.8: Frame Generator's Finite State Machines.

Different states of the frame generator when generating frames from pcap file			
CPAN frames from simulated node		CPAN frames are generated from pcap	
State	Description	State	Description
INIT	<ul style="list-style-type: none"> - Run through PCAP file frames (loop get next frame) until the extracted frame is of type association request - Go to WAIT_BEACON state 	INIT	<ul style="list-style-type: none"> - Go through first frames from the pcap file - Save beacon frame to be sent to stimulate simulated node (log frame number indicated by the user) - Get next frame until arriving to the packet of awaited association request - Go to SEND BEACON state
WAIT BEACON	<ul style="list-style-type: none"> - Wait until a beacon frame is received from the simulated CPAN node - When Beacon is received go to state ASSOC_REQ 	SEND BEACON	<ul style="list-style-type: none"> - Send beacon frame at a constant time interval defined by the protocol - When an ASSOCIATION REQUEST frame is received from the simulated node, save its long address indicated in the frame in generator's registers (address used to modify destination address fields in generated frames), then go to ASSOC_RESP state
ASSOC REQ	<ul style="list-style-type: none"> - After delay determined by protocol send the Association frame extracted in the INIT state - When association response is received from simulated node and the next CPAN beacon received, go to RUN_PCAP state - Save in registers the simulated CPAN's long address from association response frame 	ASSOC RESP	<ul style="list-style-type: none"> - Get next frame from pcap file until the extracted frame is of type association response - Change the destination long address field of the association response frame to the one saved in the SEND_BEACON state - Send the association response frame after delay set according to the protocol specifications - Go to state RUN_PCAP
RUN PCAP	<p>Loop:</p> <ul style="list-style-type: none"> - extract next unfiltered frame from pcap file - compute delay to send the frame by using the pcap frames timestamps (subtraction between extracted frame timestamp and last generated frame timestamp) - After end of delay countdown send extracted frame <p>End loop</p>	RUN PCAP	<p>Loop:</p> <ul style="list-style-type: none"> - extract next unfiltered frame from pcap file - compute delay to send the frame by using the pcap frames timestamps (subtraction between extracted frame timestamp and last generated frame timestamp) - After end of delay countdown send extracted frame <p>End loop</p>

A simulated node stimulated by frames produced by the Frame Generator should produce the same frames produced during the testbed experiment. Therefore, if the simulated node reproduces the same frames, it should reproduce the problem. The debugging capabilities provided by the simulation environment can then be used to find the source of the node's problem.

The Frame Generator was tested with a pcap file of frame exchanges from physical nodes running the OCARI protocol. The simulated node, which was running the same binary file as the physical nodes, was able to reproduce the expected frames when stimulated by the Frame Generator. An example of frame generation is shown in Chapter 4.

3.4 Modeling and Verifying protocol properties

3.4.1 The validation of protocol properties

One of the requirements of this work is to be capable of validating protocol properties during simulation. Protocol properties can be found in the protocol specifications, and we must verify that the execution of the protocol's implementation complies with those specifications.

Here are a few examples of OCARI properties that could be verified:

- The Hello message generation: Once a node is associated to the network, it must send a Hello type message every other cycle, a cycle beginning with a beacon frame from the coordinator
- The disassociation property: If a child node does not send a Hello message for 20 subsequent cycles, it must be dissociated from the network.
- CPAN coloration:
 - The CPAN can trigger the network coloring once it is stable. The network is considered stable when all the CPAN's children nodes have sent a Tree Status message indicating that their neighbors are stable.
 - The CPAN must always pick the smallest color, color 0.
- Node coloration:
 - A node must pick the smallest color that is not already picked by a conflicting neighbor (1, 2 and 3-hop neighbors)
 - A node can pick its color if all conflicting neighbors with a higher priority have already picked their color
- Once the network is colored, nodes can only send data frames in the time slot associated to their color.

Properties can be verified by observing that a node executing the protocol stack under test is responding to stimuli as is dictated by its specifications. In the case of a WSN node, the stimuli are incoming packets, the response is the outgoing packets. Therefore, we must observe the frames that are being sent and received by the node executing the protocol under test. This can be done using the Radio Link module, as it controls the packet exchanges between nodes.

In order to check protocol properties in simulation, we must model these properties, and insert these models in the simulator. There are some methods that are well suited to model protocol properties as cited in Chapter 1, such as Petri Networks or automata. We chose to use Light Esterel (LE), which is based on Finite State Machines (a type of automata) because it is able to express the evolution of concurrent process. Indeed, this synchronous programming language has been designed to model reactive systems such as communication protocols. We can model the timing of properties such as those mentioned previously by using the beginning and ending of time slots as events, and coordinator beacons to time the beginning of cycles. Moreover, Light Esterel can be translated into C code, which makes it easy to insert into SNOOPS's source code, which is based on C++ and SystemC.

Finally, all the tools used to develop, compile and test LE modules are available for free.

For these reasons, we decided to use this language to model protocol properties defined by the protocol's specifications.

In the following sections, we describe the various steps taken to model and verify a protocol property in simulation. As seen in Figure 3.9, the first step is finding a property to verify in the protocol specifications, then to write code in Light Esterel to model the property. The property can then be compiled in Blif format, and tested on a simulator called Blif_simul to verify that the property model is correct. The Light Esterel model of the property can then be compiled into C code and inserted into one of SNOOPS's module called the Observer. The observer will then be able to detect if frame exchanges comply or not with the modeled property.



Figure 3.9: The different stages used for property verification in SNOOPS.

3.4.2 What is the role of the observer module?

The observer module is integrated into the simulator to analyze the frames exchanged between nodes. To have the capability of analyzing any frame that is exchanged in the network, the observer module is contained in the radio link module. Its role is to halt simulation if a protocol property is violated, so that the source of the bug causing the violation may be pinpointed thanks to the debugging capabilities provided by the simulation framework. The observer focuses on the frames sent and received from a specific node which is running the binary protocol stack under test.

3.4.3 Modeling protocol properties

A communication protocol is considered to be a reactive system, and general purpose programming languages are not well suited to design reactive systems: they are clearly inefficient to deal with the inherent complexity of such systems [3]. Various synchronous languages dedicated to reactive systems have been designed such as Esterel [4]. Based on concurrency and synchronism, they are model-based languages to allow formal verification of the system behavior. Their execution model is simple: first, the initial state is initialized and then, for each input event set, outputs are computed and then the state is updated [3].

3.4.3.1 Light Esterel

Overview

Light Esterel (LE) [5] is a reactive synchronous programming language derived from Esterel V5 [4] that allows the efficiency and reusability of the system's design as well as the formal verification of the system's behavior, which is known to increase the reliability of a system. Just like Esterel, it is able to maintain a permanent interaction with its environment. This language is ideal for applications such as communication protocols, man-machine interface drivers or VLSI chips.

LE is an imperative language that allows unifying different styles of specifications to conceive applications [6]:

- A textual language resembling to Esterel that allows designing event driven applications or parts of one
- A textual and graphical format for automata that allows specifying controllers
- Some parts of data flow applications can be written with equations systems resembling the Lustre language [7].

The Light Esterel language units are named modules. Communication takes place between modules or between a module and its environment. Sub-systems communicate instantaneously via events. They are able to immediately take into account the reaction of other sub-systems and can have a snowball effect. Besides it is the main work of the LE compiler to check the stability of the system for all possible sequences of inputs. The module interface declares the set of input events it reacts to and the set of output events it emits. The body of the module is expressed using operators, some of which are dedicated to the manipulation of logical time.

The textual or graphical Light Esterel views gives the possibility to write compact specifications for very complex systems. A system with thousands of states can generally be

specified by an Esterel program of only a few hundred lines thanks to the explicit pre-emption and parallel operator. Moreover, this deterministic concurrent programming language can be compiled into a Finite State Machine (FSM) classically represented by an automaton. LE automata are Mealy machines and they have a set of input signals, which can be valued, to define transition triggers and a set of output signals that can be emitted when a transition is raised.

Consequently, Light Esterel programs can also be compiled into popular languages such as C, VHDL, C#, or other synchronous languages like Lustre. In our case we used the C program compilation of Light Esterel.

3.4.3.2 Example of a property modeled in Light Esterel

Let us take an example of property for wireless sensor networks. As soon as a node is associated to a network coordinator node, it must send a Hello type frame every other cycle, knowing that a cycle begins with a Beacon type frame from the network coordinator.

Figure 3.10 shows a state machine representation of this property. The system stays in the first state until the node is associated. The next state represents the beginning of a loop. When a beacon from the coordinator is received (begin cycle), two branches are executed in parallel. In one branch the emission of a Hello frame from the node will trigger the emission of a *hello_ok* signal. On the other branch, if two new beacons from the CPAN (begin cycle) are subsequently received without the *hello_ok* signal being triggered from the other branch, an *Alarm* signal will be emitted. The node's disassociation makes the state machine return to its initial state.

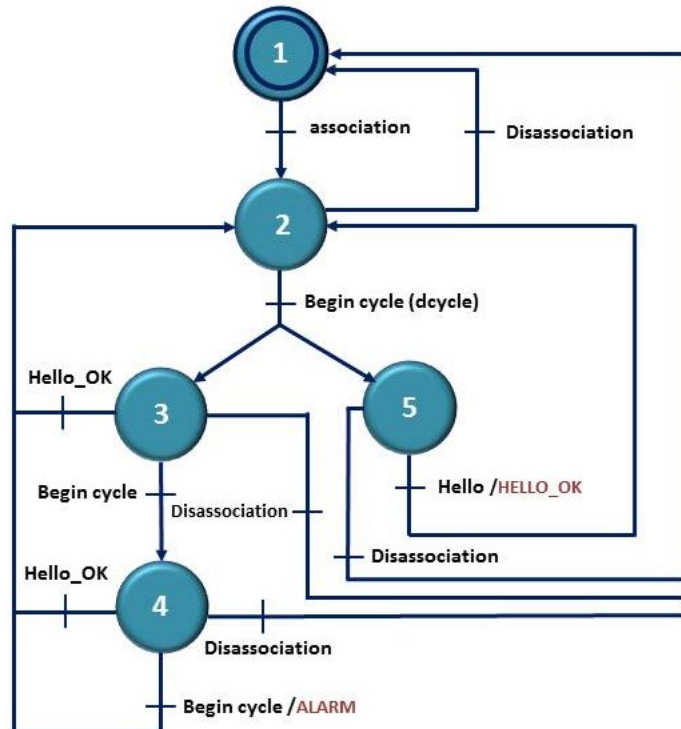


Figure 3.10: Hello property modeled as a state machine.

Figure 3.11 shows the code written for the Hello property in LE. We can see the declaration of the name of the module (hello), the input and output signals, and the declaration

```

module hello:
Input: dcycle,
      hello,
      association,
      disassociation;
Output: alarm;

local hello_ok
{
  loop
  {
    wait association >>
    weak abort
    {
      loop
      {
        wait dcycle >>
        {
          {wait hello >> emit hello_ok }
          ||
          weak abort
          {
            wait dcycle >> wait dcycle >> emit alarm
          } when hello_ok
        }
      }
    } when disassociation
  }
}
end

```

Figure 3.11: Hello property modeled in LE.

of a local signal *Hello_Ok*. The code is equivalent to the state machine in Figure 3.10. A *wait* precedes the signal awaited for a transition, the operator '||' represents a parallel execution. Some sections can be aborted at the reception of a specific signal (e.g. weak abort {...} when Hello_Ok).

3.4.4 Inserting properties into the observer module

3.4.4.1 Testing and compilation of modeled properties

Clem

The Clem (Compilation of LE Modules) toolkit [8] [9] is capable of generating code to run real applications. It is a modular compiler of the LE language. For software models, Lustre and Esterel code can be generated. For hardware description applications, VHDL can be generated, and for software applications C code can be generated. A block diagram of the Clem toolbox is given in Figure 3.12.

Clem relies on the equation semantics of LE and implements the semantic rules to compile each program into a linear quadri-valued equation system, that enables calculating the status of local and output signals based on the status of the input signals in an

algebra with four values: absent, present, bottom and error [5]. Absent and present represent the absence or presence of a signal in an environment. Bottom is the status of a signal for which there is no information (undefined). The error status represents the case when the signal is both absent and present. This equation system is then translated into a boolean equation system (each equation system is translated into two boolean equations). The LEC format is an intermediate compilation format for Clem. This compact format enables representing boolean equations in a symbolic way as a Binary Decision Diagram (BDD). Generating an LEC format file enables reusing more efficiently the code for modules that are already compiled.

When the compilation of all the different modules into the LEC format is done, in a *finalization* phase, the equation system is simplified, the quadri-valued signals are projected onto only two values (present, absent) and all bottom signals become absent. In this phase, several code generators provide code for different target applications (Esterel, VHDL, C,...). Some SMV (Symbolic Model Verification) code is also generated to be interfaced with the model checking tool NuSMV [10], which is an open source BDD based model checker. A BLIF (Berkeley Logic Interchange Format) file format well suited to the application of verification tools can also be generated. Moreover clem can be used as frontend of an automatic test generator [11].

In conclusion, Light Esterel compilation insures a modular compilation of reusable modules by linking the principal system of equation's module with an already compiled sub-module. It also enables verifying the behavior of the system under test. Formally verifying that a system is correct means insuring that its model satisfies the formula describing its expected behavior, and LE generates an equation system that can represent the behavior of a system.

The Blif_Simul simulator

To simulate LE programs, the boolean equation system created by Clem is translated into BLIF format. This format can be used to simulate the system described in LE when using it in the Blif_simul simulator, which is a graphical simulation tool for implicit automaton written in blif format. It is part of a toolkit which also includes an automaton generator for implicit blif automaton, a blif converter, a symbolic blif checker, an automatic test generator, and a symbolic behavioral comparator [12].

It is used to test LE code compiled into blif format. As shown in Figure 3.13, the different inputs and outputs of the modeled system appear on the GUI. The signals shown in this Figure are those related to the Hello property. For each logical instant, the user updates the state of the inputs by ticking the corresponding box on the far left side of the GUI, the output signal states are updated by the simulator when the user enter "Go". The output signal's bow color is either yellow (meaning absent), or blue (meaning present). The user can verify that the outputs correspond to the ones expected for the given input signal state sequences.

For example, for the Hello property, a Hello message must be sent by the node every other cycle after association. Therefore, ticking the inputs association, then dcycle (begin cycle) 3 subsequent times should cause the alarm signal to be present. Indeed, it turns blue after entering this sequence of inputs.

3.4.4.2 Including the properties to check in the observer

Different properties of the protocol can be modeled independently as LE modules based on the protocol's specifications. They are compiled using Clem into blif format and into

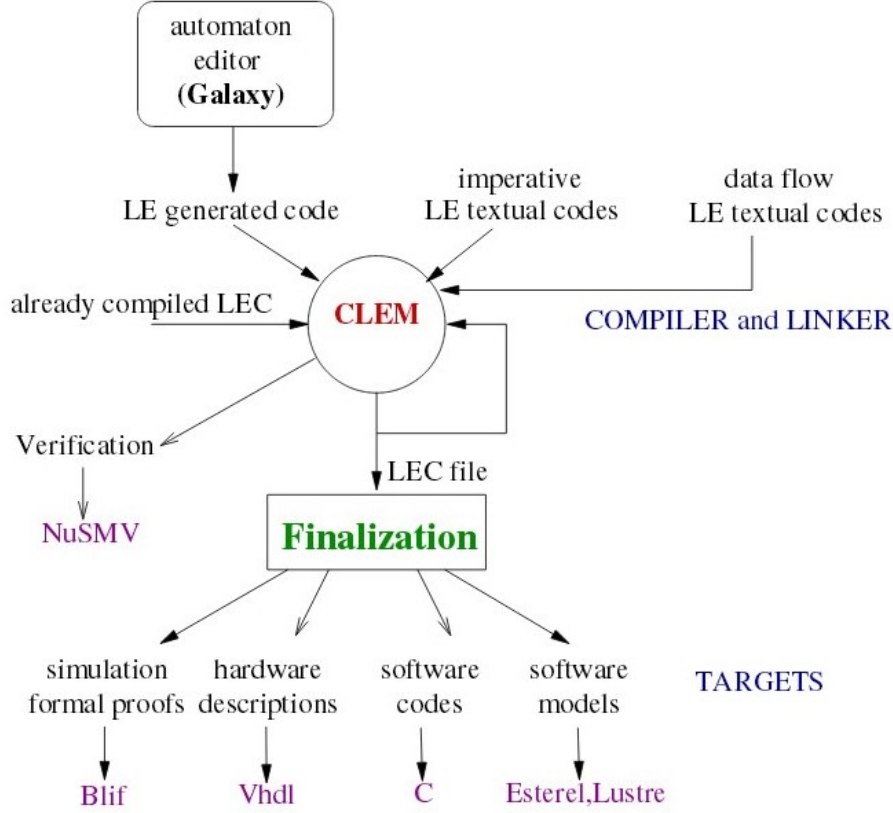


Figure 3.12: Block diagram of the Clem toolbox [9].

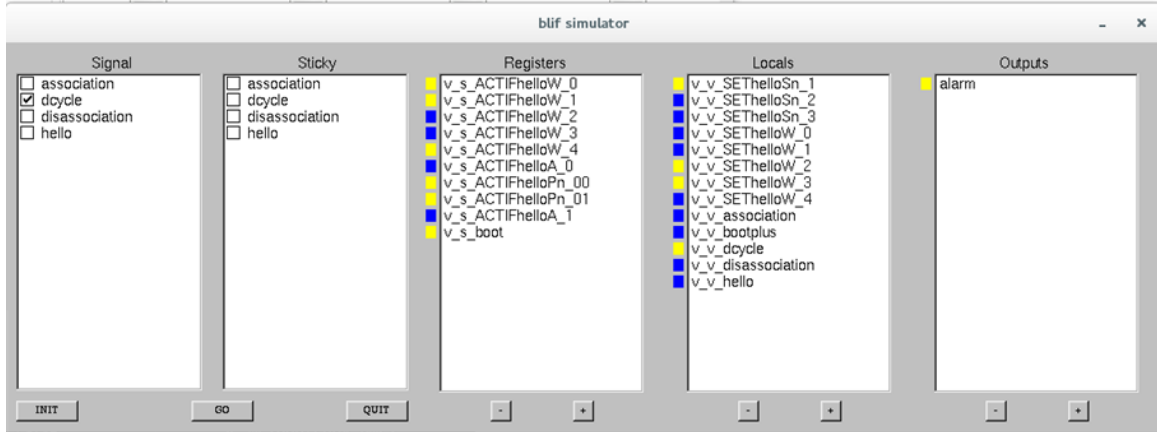


Figure 3.13: Graphical user interface of Blif_simul testing the Hello property.

C code.

We use the blif_simul simulator to check that the behavior of the LE module that we developed complies with the property it models. When we are sure that the LE module's behavior is correct, its compiled C code can be introduced into SNOOPS's observer module.

As previously explained, a LE module can be viewed as an automaton. When C code

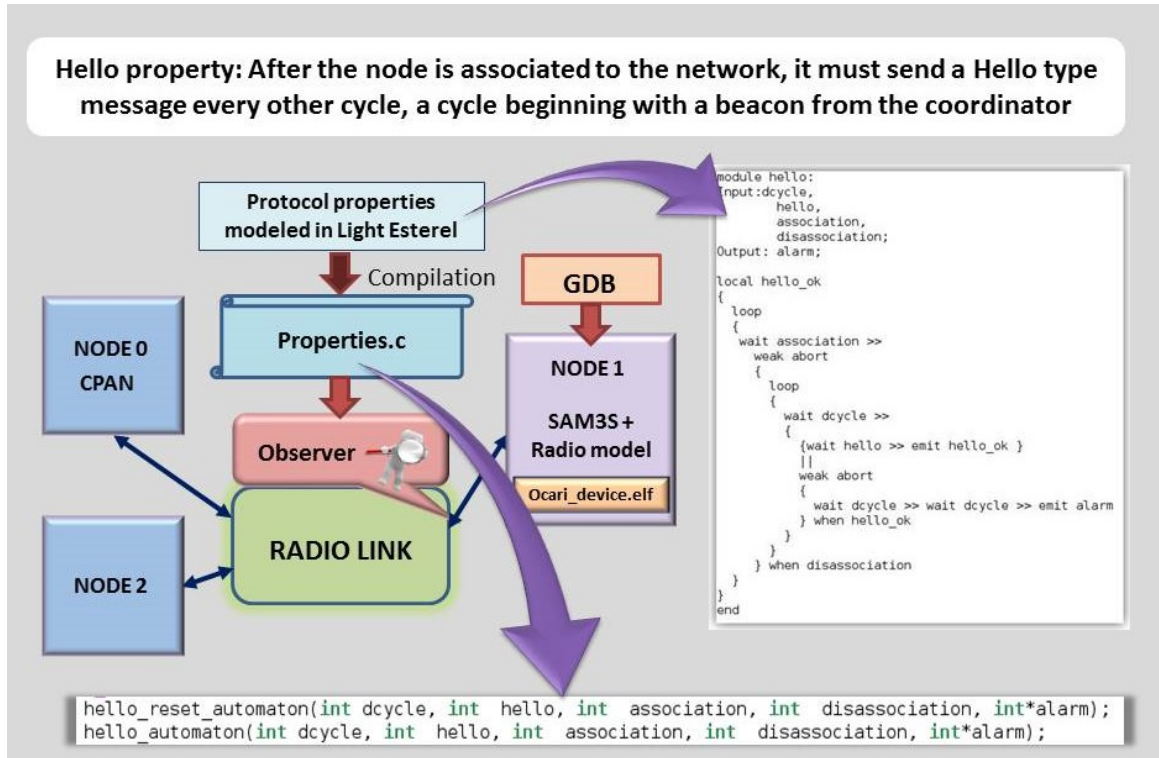


Figure 3.14: Block diagram representing the injection of the Hello property to check in the observer module of SNOOPS.

is generated at compilation, two files are generated. One file is the C code describing the behavior of the automaton, the other file is a header file declaring two functions: a *reset_automaton* function and an *automaton* function. The *automaton* function contains as many parameters as there are inputs and outputs to the LE module the automaton represents as seen in Figure 3.14. The *reset_automaton* function returns the automaton to its initial state.

The observer dissects the frames sent and received from a specific node that is running the binary protocol stack under test. If the information contained in those frames are relevant to the modeled protocol property, the corresponding Light Esterel C program function is called with the corresponding boolean presence predicat set to 1. For example, if the observer's dissector detects that an outgoing frame from the node under observation is a Hello type frame, the function *hello_automaton*(0, 1, 0, &alarm) is called. If alarm is true after the function call, the property has been violated.

3.4.5 Checking properties during simulation

3.4.5.1 Simulation halted as property is violated

The main role of the observer is to halt the simulation if a property modeled in LE, compiled, and included in the code of the observer is detected as violated.

The observer dissects the frames and calls functions updating the properties automaton that depend on events related to the different analyzed frames (e.g. a node association in the case of the Hello property seen in Section 4.10). After the function call, the Light Esterel program outputs are checked to see whether the property is violated (presence of the output alarm signal). If this alarm signal is present after returning from a function

that made a property automaton progress, SNOOPS is halted to find the bug that caused the property to be violated.

We can also add a *Property_checked* output signal, that is present if the conditions of the property have been met. The latter information helps in statistics on property coverage. Indeed, in certain scenarios, some properties might not be checked at all while other properties might be checked numerous times. The simulation scenarios have to be adapted in this case to make sure that all properties have been extensively put under test.

3.4.5.2 Debugging capabilities of the simulation framework

Debugging the SNOOPS framework with GDB

Visibility of the state of the hardware platforms and all other aspects of the simulation framework can be monitored using the GNU debugger (GDB).

A debugger is a program that runs other programs a user wishes to debug. With GDB, the user can follow the program's execution, examine what happened when the program stops unexpectedly, examine the different variables and modify them to experiment on correcting the effect of a bug. The user can also call functions independently from the program's behavior. Breakpoints can be placed on specific lines, or variables in the code to stop the program's execution when the execution reaches this line or this variable was modified.

GDB does not offer a graphical user interface, but uses a command-line interface. However, a few front-ends have been developed for it, to benefit from a graphical interface for GDB, such as UltraGDB, Data Display Debugger (DDD), KDbg, or Xcode debugger.

Debugging the guest code with GDB

GDB offers a remote mode to debug embedded systems. When using remote debugging, GDB runs on one machine and the program being debugged runs on another. GDB communicates with a remote stub using a specific GDB protocol via Serial or TCP/IP.

A GDB stub is provided by QEMU to debug the guest code being executed by the modeled processor. GDB stubs are made to remotely debug a program running on another machine, and in this case, the other machine is the one emulated by QEMU (e.g. the TLM machine). The gdb stub is invoked by launching QEMU with the `-gdb` or `-s` option. With the `-s` option, the default tcp port is 1234. With the `-gdb` option, the user can specify a tcp port number as an argument. QEMU's `-S` option freezes the CPU on startup, to attach GDB and get it set up, then the guest code's execution continues.

The protocol under test's binary file can be debugged using this stub. The `-s` and `-S` options can be added to QEMU's launch in TLMu source files. The execution of SNOOPS is halted until another terminal is opened and a debugger is connected to the GDB stub opened in SNOOPS by using the `-s` option.

If the user wishes to debug more than one node at a time with SNOOPS, the `-gdb` option must be used, because only one node can use the default 1234 port. The port number argument in TLMu is called *gdb_conn*. It is a parameter of TLMu's constructor. In SNOOPS, it is passed down as a parameter of the *sam3s_node* constructor. If its value is NULL, the `-gdb` option is not added to QEMU's launch, otherwise QEMU is launched with the arguments `-gdb gdb_conn`. We added the `-S` argument for the node expecting the connection to be frozen until GDB is attached to the *gdb_conn* port. We also added a terminal output message to indicate to the user what is the port number for the GDB connection. This allows visibility on all the emulated nodes at the same time.

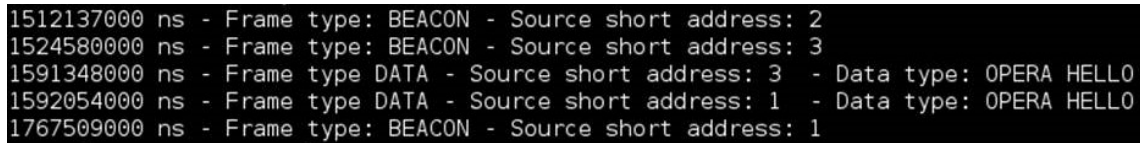
For an ARM microprocessor such as the Cortex-M3, the GNU Debugger for this bare-metal target is called ARM-EABI (arm-none-eabi-gdb).

The stack's code's execution can be traced, followed step by step, and even instruction by instruction. Thanks to this feature, when SNOOPS halts because a property is violated, the user can trace which functions were being executed by the stack at that time, if the stack was stuck in an infinite loop, etc.

3.5 Simulation output

3.5.1 The terminal output

When a TLM transaction from a node is received by the Radio Link module, this frame's data is extracted. A dissector was developed to dissect the frames into fields (frame control field, addressing fields, payload, etc.) and interpret those fields. The first information to be extracted is the frame length, then the frame type (e.g. command, data,...), the address of the node that sent the frame, the destination address, and the data (payload) which is also interpreted depending on the frame type. As shown in Figure 3.15, the basic information on the frames is printed out in the terminal during the simulation along with the SystemC timestamp of the frame. To analyze the frames in more detail, a log file compatible with various network analyzers is created.



```
1512137000 ns - Frame type: BEACON - Source short address: 2
1524580000 ns - Frame type: BEACON - Source short address: 3
1591348000 ns - Frame type DATA - Source short address: 3 - Data type: OPERA HELLO
1592054000 ns - Frame type DATA - Source short address: 1 - Data type: OPERA HELLO
1767509000 ns - Frame type: BEACON - Source short address: 1
```

Figure 3.15: A section of a terminal output as SNOOPS is running.

3.5.2 Analyzing the simulation output with Wireshark

3.5.2.1 What is Wireshark?

Wireshark [13] is an open source network protocol analyzer, which started as a project by Gerald Combs in 1998. It is similar to tcpdump but with a graphical frontend. It enables live captures and offline analysis. Live data packets can be read from Ethernet, IEEE 802.11, PPP/HDLC, ATM, Bluetooth, USB, Token Ring, Frame Relay, FDDI, etc. The captured frames can be visualized through a GUI, and a wide variety of filters can be applied on the displayed frames. Individual frames can be selected, and the different bytes composing the frame can be viewed along with their signification as Wireshark can decode frames for a wide range of protocols.

In Figure 3.16, a block diagram of Wireshark can be found, the Epan block representing the Ethereal Packet ANalyzer, the packet analyzing engine. It contains dissectors to dissect information from individual packets. Dissector plugins allow users to add their own dissectors. Display filters make it possible to filter the frames to display on the GUI. The wiretap library is used to read and write capture files in libpcap, pcapng, and many other file formats.

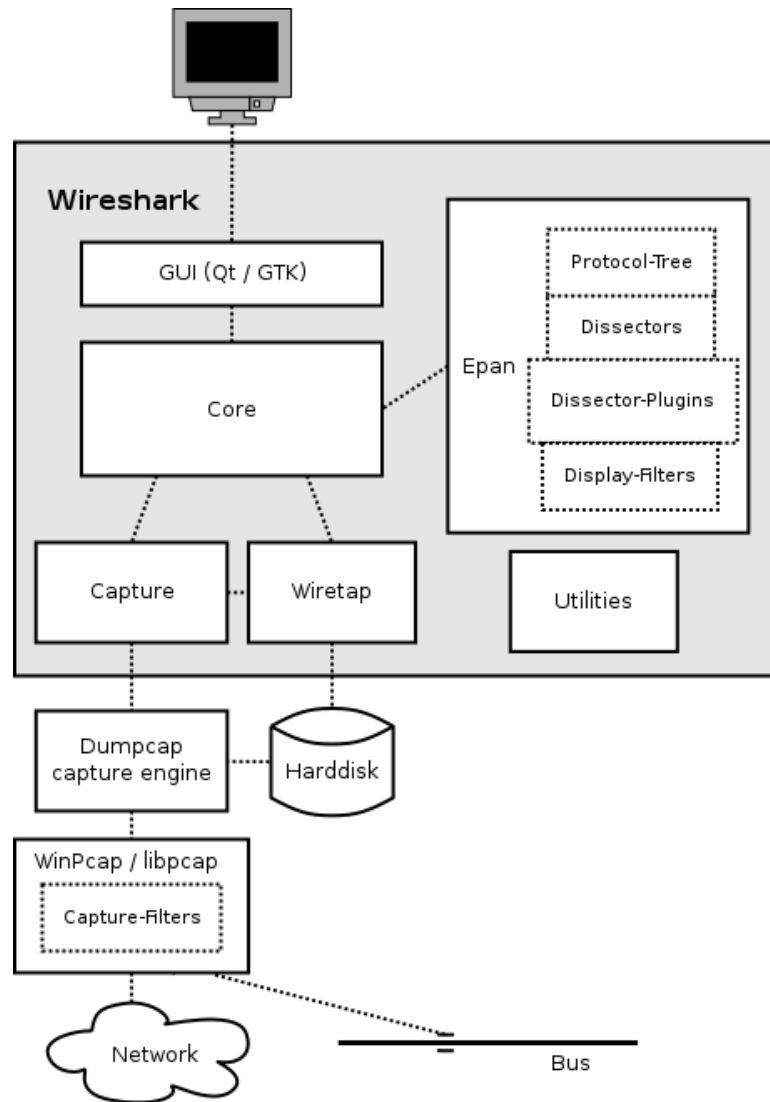


Figure 3.16: Wireshark block diagram.

3.5.2.2 What is the PCAP format?

The libpcap file format [14] is the main capture file format used in many networking tools such as Wireshark to log captured frames exchanged in a network. The extension of libpcap based files is .pcap.

The format that is currently used for libpcap has not changed since 1998 (libpcap 0.4), except for the nanosecond resolution of timestamps which was introduced in libpcap 1.5.0. The file starts with a global header, then for each captured frame logged in the libpcap file, there is a packet header section and a packet data section.

3.5.2.3 A custom plugin to dissect OCARI frames in Wireshark

There are many protocols that are known to Wireshark, meaning that Wireshark can dissect the frames of these protocols to indicate the meaning of its different sections.

The Frame dissector is used to start the dissection, it dissects the packet details of the capture file itself. From there it passes the data on to the lowest-level data dissector. The payload is then passed on to the next dissector and so on. At each stage, details of the packet will be decoded and displayed [15].

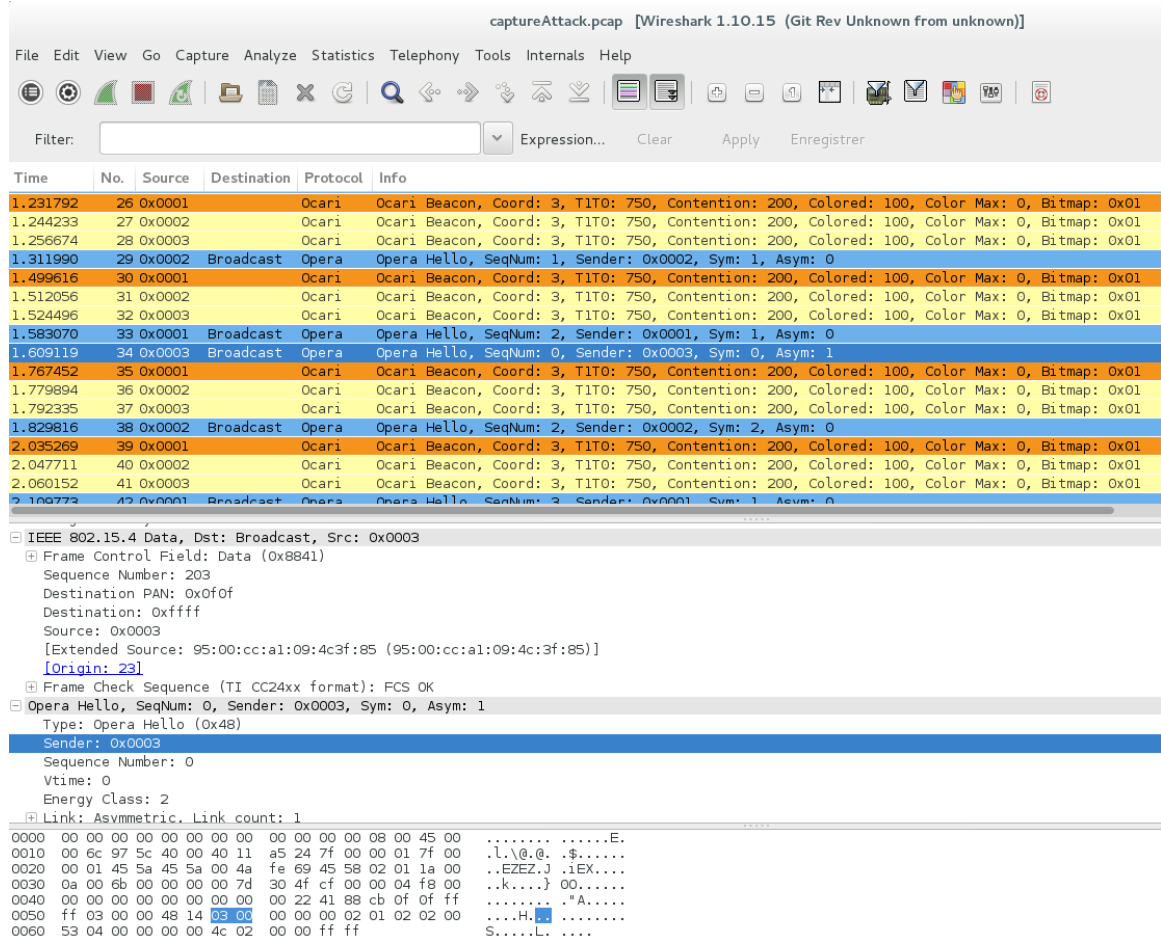


Figure 3.17: Wireshark used to analyze the information on a frame from the OCARI protocol.

If a protocol's dissector module is not already compiled into the main program, it is possible to create a dissector plugin that handles the frame's dissection using a shared library or DLL (Dynamic Link Library).

Wireshark is able to dissect frames from many popular protocols (e.g. IEEE 802.15.4), but in order to use Wireshark to dissect OCARI frames, a specific plugin was created by the INRIA (member of the OCARI Alliance).

3.5.2.4 Creation of a Pcap file during simulation

Analyzing the simulation results and traces of frames exchanged in the terminal only is not very practical. To analyze simulation results with a network analyzer, as well as to benefit from a graphical interface, the packets exchanged between the simulated nodes are recorded in a pcap format log file. This task is handled by the *pcap_dumper* SystemC module, which is instantiated in the Radio Link module.

When a packet arrives as a TLM transaction in the Radio Link module, the transaction's data is extracted. This data and the SystemC timestamp of the transaction's reception are used to display information about the frame on the terminal (using *printf* functions), and they are recorded in the pcap format log by the *pcap_dumper* module. The *dump_packet* function of the *pcap_dumper* module is called in the Radio Link module. The frame data and the SystemC timestamp are passed as arguments in the function call. The *dump_packet* function converts the SystemC timestamp to be compatible with the pcap timeval

structure members *tv_sec* and *tv_usec*, which are the number of seconds since January 1st, 1970, and the extra microseconds. The values of *tv_sec*, *tv_usec*, and the packet length are set in the packet header structure. The frame's data is copied into the packet data array after a series of bytes that are fixed and are interpreted as packet encapsulation protocol information by network analyzers. Then using the pcap library's *pcap_dump* function with the log file, the packet data and packet header as arguments, the packet information is saved in the pcap format log file.

The pcap file created during simulation can be analyzed with Wireshark as seen in Figure 3.17. In this Figure, we can observe the output of SNOOPS while simulating a scenario to test the OCARI protocol. Different types of frames appear in different colors. The selected frame is a data frame of type OCARI HELLO, and we can observe the meaning of various field. The selected field indicates that the source address is 3 and the corresponding bytes are highlighted.

3.6 Conclusion

SNOOPS is a network simulator implemented in SystemC. It instantiates either node emulators executing the binary files of a protocol under test, or high level node models which can handle diverse tasks, to avoid slowing down the simulation's execution.

The different node modules that are instantiated communicate through TLM transactions that transit through the Radio Link. The Radio Link controls the network's topology by creating or deleting links as the simulation time advances. It also controls the links' quality (packet loss).

The Frame Generator added to SNOOPS makes it possible to reproduce scenarios recorded during protocol tests on testbeds inside the simulation environment, in order to have a better visibility on the protocol's execution and the state of the node.

Information on the frame exchanges in the simulation environment can be observed in a terminal or using a network analyzer such as Wireshark as a log file in pcap format is produced during simulation.

These various features of this network simulator are assets to facilitate debugging WSN protocols. Another asset is to detect bugs automatically while emulating the protocol's execution. That is the purpose of the observer module.

SNOOPS offers the possibility to check that protocol property requirements are met as the stack is being emulated on a node model. The properties one wants to check can be modeled in Light Esterel, their models can be validated on a simulator such as Blif_simul or Cles, and then the properties are compiled in C and introduced into the simulation framework. Each property modeled in LE can be seen as a Finite State Machine and encoding by a set of Binary Decision Diagram (BDD) to compute outputs and next states. The observer module contains the property models, and controls the state transitions of the corresponding BDD. It dissects the frames transiting through the Radio link module to isolate events that may trigger a state change. If a BDD's progression leads to the emission of an alarm signal from that BDD, the simulation is halted and the user can debug the stack's code to figure out the source of the bug that caused the property to be violated.

A few case studies of how SNOOPS can contribute to testing a protocol and debugging it are described in the next Chapter.

Bibliography

- [1] M. Grottke and K. S. Trivedi, “A classification of software faults,” *Journal of Reliability Engineering Association of Japan*, vol. 27, no. 7, pp. 425–438, 2005. 25, 79
- [2] C. Barnes, F. Verdier, A. Pegatoquet, D. Gaffé, and J.-M. Cottin, “Validating a wireless protocol implementation at binary level through simulation using high level description of protocol properties in light esterel,” *ICWMC 2016*, p. 70, 2016. 84
- [3] A. Ressouche, D. Gaffé, and V. Roy, “Modular compilation of a synchronous language,” in *Software Engineering Research, Management and Applications*, pp. 157–171, Springer, 2008. 92
- [4] G. Berry, *The Esterel v5 language primer: version v5_91*. Centre de mathématiques appliquées, Ecole des mines and INRIA, 2000. 92
- [5] D. Gaffé and A. Ressouche, “Algebraic framework for synchronous language semantics,” in *Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on*, (Birmingham, UK), pp. 51–58, IEEE, July 1-3 2013. 2, 92, 95
- [6] D. Gaffé and A. Ressouche, “Compilation modulaire d’un langage synchrone,” *Technique et Science Informatiques (TSI), no. spécial" Méthodes Formelles*, vol. 30, pp. 441–471. 92
- [7] N. Halbwachs, F. Lagnier, and C. Ratel, “Programming and verifying real-time systems by means of the synchronous data-flow language lustre,” *IEEE Transactions on Software Engineering*, vol. 18, no. 9, pp. 785–793, 1992. 92
- [8] A. Ressouche and D. Gaffé, “Clem toolkit homepage.” www-sop.inria.fr/pulsar/projects/Clem/introduction.html. 94
- [9] D. Gaffé and A. Ressouche, “The clem toolkit,” in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 495–496, IEEE Computer Society, 2008. xii, 94, 96
- [10] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, “Nusmv 2: An opensource tool for symbolic model checking,” in *International Conference on Computer Aided Verification*, pp. 359–364, Springer, 2002. 95
- [11] M. Abdelmoula, *Génération automatique de jeux de tests avec analyse symbolique des données pour les systèmes embarqués*. PhD thesis, University of Nice Sophia Antipolis, 2014. Internship Report. 95
- [12] D. Gaffé, “Software tools about a subset of blif format,” 2017. www.unice.fr/dgaffe/recherche/blif_tools.html. 95

- [13] G. Combs *et al.*, “Wireshark,” *Web page: [http://www. wireshark. org/](http://www.wireshark.org/)last modified*, pp. 12–02, 2007. 99
- [14] “Libpcap file format,” 2015. wiki.wireshark.org/Development/LibpcapFileFormat. 100
- [15] “Wireshark developer’s guide,” 2014. www.wireshark.org/docs/wsdg_html_chunked/index.html. 100

Chapter 4

Simulation results for some case studies

Contents

- 4.1 SNOOPS usage and benefits with OCARI 106**
 - 4.1.1 Verifying the disassociation property 106
 - 4.1.2 An undefined command according to specifications 108
 - 4.1.3 A node coloring issue 109
 - 4.1.4 Replay of a testbed scenario 113
 - 4.1.5 Network destabilization by a false node 116
- 4.2 SNOOPS simulation time 119**
- 4.3 Conclusion 123**

To demonstrate the benefits of the proposed simulation environment, we tested it with the OCARI protocol, to verify that the implementation of the protocol complies with the properties from its specifications.

4.1 SNOOPS usage and benefits with OCARI

4.1.1 Verifying the disassociation property

To test the insertion of a property modeled in Light Esterel in the simulation framework, we decided to model one of OCARI's properties regarding node disassociation.

This property states that if a parent node hasn't received a beacon from one of its children nodes for 20 subsequent cycles (a cycle beginning with the network coordinator's beacon), it must request the child's disassociation to the network coordinator. If a disassociation request is not issued after 20 cycles, the property is considered violated. If the request is issued, the property is considered as checked and that in those particular scenario conditions, the implementation of the protocol complies with the property in question. Despite the fact that this property has not been found to be violated by the protocol, we want to show that the integration of properties into the simulation framework is successful. For the purpose of proving that the observer is capable of detecting the violation of a modeled property, we also modeled a disassociation property that OCARI does not comply with, derived from the original disassociation property by decreasing the maximum number of beacons before disassociation.

The property modeled in Light Esterel focused on the status between the parent node under observation and one of its children. Figure 4.2 shows the Light Esterel model of this property. As seen in Figure 4.1, the input events/signals are *Begin_cycle* (produced when a CPAN beacon is emitted), *Association*, *Disassociation* and *Beacon* (Beacon from the child node in question). The output signals are an alarm if the property is violated and *property_ok* if the property is correctly implemented. The signals *beacon_received*, *missed_beacon* and *property_fail* are local signals. As it can be understood from the code of Figure 4.2, after the node's child's association to the network, two branches of the FSM that is represented by this code are executed in parallel. Those branches abort if the child node is disassociated, and return to a state waiting for the child's association. The first branch keeps track of the beacons being missed or received during each cycle, and emits either a *missed_beacon* or a *beacon_received* signal. The second branch keeps track of the subsequent *missed_beacon* signals that are from the first branch. If 20 *missed_beacon* are received without a *beacon_received* signal, the alarm signal is emitted signaling that the property is violated. However, if the child's disassociation is received before the 20 subsequent missed beacons, the *property_ok* signal is emitted and the branch counting the missed beacons is aborted. The FSM state returns to waiting for the child's association.

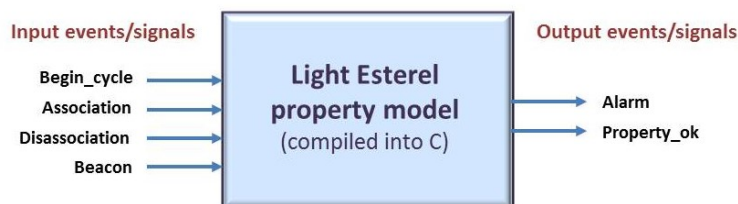


Figure 4.1: Box representing Disassociation property with input and output signals.

```

loop
{
  wait association >>
  {
    weak abort
    {
      wait begin_cycle >>
      loop
      {
        weak abort
        {
          wait beacon >>
          {emit beacon_received || wait begin_cycle}
        } when missed_beacon
        ||
        weak abort
        {
          wait begin_cycle >> emit missed_beacon
        } when beacon_received
      }
    } when disassociation
    ||
    {
      weak abort
      {
        weak abort
        {
          wait 20 missed_beacon >>
          {emit alarm || emit property_fail}
        } when beacon_received
        ||
        weak abort
        {
          wait disassociation >> emit property_ok
        } when property_fail
      } when disassociation
    }
  }
}

```

Figure 4.2: Main loop of the program in Light Esterel representing the Disassociation property.

The correct functionality of the modeled property was tested with the *Blif_simul* simulator, meaning that once a child node was associated if there were 20 subsequent beacon frames that are missed without a disassociation, the alarm signal was produced. However, if the disassociation happened before, the signal *property_ok* was produced.

The Light Esterel file is then compiled into C and the function *disassociation_automaton* is created. The function *disassociation_automaton* is called when the observer dissects the frames sent and received by the node under focus from the observer if one of the input events (seen in Figure 4.1) is detected. This is done for a node with one child, but by adding a few lines to the Light Esterel code so that a disassociation automaton can be run in parallel for each child a node has. In this case, the function called in C is the same, but a value is added to the input signals that represents the event's corresponding child number. Then according to the value of the child number, the corresponding *disassociation_automaton*'s state is changed according to the input event.

The property is tested in a scenario where the nodes have a topology as seen in Figure 4.3. A while after nodes 2 and 3 have associated to the CPAN and started sending it their sensed data, the link between nodes 3 and 2 is removed. Node 2 can no longer receive the frames emitted from node 3. Under this scenario the property checks out: as seen in Figure 4.4, the disassociation happens in due time: a disassociation command frame is sent from node 2 to the CPAN indicating that node 3 is disassociated from the network.

This disassociation happens before there are 20 cycles without a beacon from node 3.

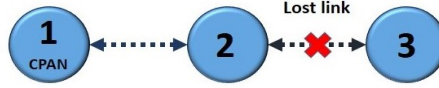


Figure 4.3: Topology of the nodes in the tested scenario.

```

-----Observer: new cycle-----
31872806000 ns - Frame type: BEACON - Source short address: 1
31885249000 ns - Frame type: BEACON - Source short address: 2
31992025000 ns - Frame type: DATA - Source short address: 1 - Data type: OPERA HELLO
31996272000 ns - Frame type: DATA - Source short address: 1 - Data type: OPERA STC
32127505000 ns - Frame type: DATA - Source short address: 2 - Transmitting SENSED DATA

-----Observer: new cycle-----
32299344000 ns - Frame type: BEACON - Source short address: 1
32311787000 ns - Frame type: BEACON - Source short address: 2
32331862000 ns - Frame type: COMMAND - Dest short address: 1 - Command type: DISASSOCIATION
-----Observer: node 3 disassociation-----

*****PROPERTY OK*****

32382586000 ns - Frame type: DATA - Source short address: 2 - Data type: OPERA HELLO

```

Figure 4.4: Simulation traces with observer signaling that the property is verified.

However, we deliberately modified the property to change the number of maximum lost beacons before a disassociation request must be issued: here a disassociation is expected after 5 lost beacons in a row instead of 20. This is not a property that OCARI is supposed to comply with, but we use it to prove that a property violation can be detected by the observer, which then would halt the simulation. We integrated it into the simulation framework, and as can be seen in Figure 4.5, after 5 cycles without a Beacon from node 3, and because there has been no disassociation request, the alarm signal was successfully generated and the simulation was stopped.

This example demonstrates that the insertion of properties modeled in Light Esterel into the simulation framework works as expected.

4.1.2 An undefined command according to specifications

In a simulation, we observed that one of the nodes produced a command frame, with a command ID that was undefined in the OCARI specifications and unrecognized by the dissector used in Wireshark. Figure 4.6 shows the simulation log file produced by SNOOPS viewed in Wireshark.

In this simulation, 6 nodes are instantiated with fully modeled platforms running OCARI stack executable files: *Ocari_CPAN.elf* for the network coordinator and *Ocari_DEVICE.elf* for other nodes. The topology is defined in the Radio link module so that all nodes can communicate bidirectionally with each other, except for node 6 which can only communicate with node 5.

As seen in the log file viewed on Wireshark in Figure 4.6, the undefined command is sent by node 5 to node 1, the network coordinator. The command identifier (0x18) is unknown to the dissector and could not be found in the OCARI specifications.

We placed a debugger on node 1 and restarted the simulation to observe how node 1 interprets the frame received from node 5 with the unknown command. We placed a breakpoint on the frame reception interrupt handler at the time it was supposed to receive the unknown command frame, and traced the stack's code execution.

```

-----Observer: new cycle-----
23877750000 ns - Frame type: BEACON - Source short address: 1
23890192000 ns - Frame type: BEACON - Source short address: 2
23902635000 ns - Frame type: BEACON - Source short address: 3
23943349000 ns - Frame type: DATA - Source short address: 1 - Data type: OPERA HELLO
24132446000 ns - Frame type: DATA - Source short address: 2 - Transmitting SENSED DATA

-----Observer: new cycle-----
24304299000 ns - Frame type: BEACON - Source short address: 1
24316742000 ns - Frame type: BEACON - Source short address: 2
24375599000 ns - Frame type: DATA - Source short address: 2 - Data type: OPERA HELLO
24559000000 ns - Frame type: DATA - Source short address: 2 - Transmitting SENSED DATA

-----Observer: new cycle-----
24730836000 ns - Frame type: BEACON - Source short address: 1
24743278000 ns - Frame type: BEACON - Source short address: 2
24850006000 ns - Frame type: DATA - Source short address: 1 - Data type: OPERA HELLO
24985529000 ns - Frame type: DATA - Source short address: 2 - Transmitting SENSED DATA

-----Observer: new cycle-----
25157387000 ns - Frame type: BEACON - Source short address: 1
25169829000 ns - Frame type: BEACON - Source short address: 2
25231658000 ns - Frame type: DATA - Source short address: 2 - Data type: OPERA HELLO
25276535000 ns - Frame type: DATA - Source short address: 1 - Data type: OPERA STC
25412083000 ns - Frame type: DATA - Source short address: 2 - Transmitting SENSED DATA

-----Observer: new cycle-----
25583942000 ns - Frame type: BEACON - Source short address: 1
25596385000 ns - Frame type: BEACON - Source short address: 2
25619787000 ns - Frame type: DATA - Source short address: 1 - Data type: OPERA HELLO
25661137000 ns - Frame type: DATA - Source short address: 2 - Data type: OPERA STC
25838639000 ns - Frame type: DATA - Source short address: 2 - Transmitting SENSED DATA

-----Observer: new cycle-----
26010482000 ns - Frame type: BEACON - Source short address: 1
26022925000 ns - Frame type: BEACON - Source short address: 2
26069905000 ns - Frame type: DATA - Source short address: 2 - Data type: OPERA HELLO
26265177000 ns - Frame type: DATA - Source short address: 2 - Transmitting SENSED DATA

-----Observer: new cycle-----
*****ALARM: DISASSOCIATION PROPERTY VIOLATED*****

```

Figure 4.5: Simulation traces with observer signaling that the property is violated.

This led us to the function *process_data_ind_not_transient*, where the frame's data is parsed, first the frame type is interpreted (in this case COMMAND), then through a *switch case*, the mac command is interpreted, and the case the debugger led us to is MACARI_CMD_DISASSOC_RELAY_REQUEST, as seen on the highlighted line in figure 4.7, which displays the terminal with the remote debugger.

The unknown command to the dissector and the specifications is therefore a disassociation relay request command, which is coherent because according to the log file, node 6 had stopped emitting frames, so node 5 was notifying the coordinator that node 6 was disassociating from the network.

While the protocol stack's execution is not impacted by that bug, the behavior is not compliant with the OCARI protocol specifications. This issue is actually an oversight in those specifications, which should be corrected.

4.1.3 A node coloring issue

During some simulations, an issue occurred with the node coloring process. According to the OCARI specifications, once the network is considered stable, the coordinator triggers the coloration of nodes. Nodes generate a color message at constant intervals until a stop condition is met (all nodes are colored or the coloration is canceled due to an unstable network). A time slot is associated to each color, and nodes select their color in order to avoid interference with other nodes of their network. Here are the coloration rules:

1. Two nodes with the same color must be separated by at least 3 hops. Therefore, a

Time	No.	Source	Destination	Protocol	Info
39.040884	846	0x0002	Broadcast	Opera	Opera Color, Seq Num: 12, Sender: 0x0002, Tree Seq Num: 0, Max Color: 0, Color: 1, Priority: 2
39.041844	846	0x0002	Broadcast	Opera	Opera Hello, SeqNum: 42, Sender: 0x0002, Sym: 4, Asym: 0
39.045041	847	0x0002	Broadcast	Opera	Opera Color, Seq Num: 519, Sender: 0x0002, Tree Seq Num: 0, Max Color: 5, Color: 2, Priority: 1
39.053633	848	0x0004	Broadcast	Opera	Opera Color, Seq Num: 12, Sender: 0x0004, Tree Seq Num: 0, Max Color: 5, Color: 4, Priority: 1
39.390125	849	0x0001	Ocari	Ocari	Ocari Beacon, Coord: 6, TlT0: 1350, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
39.402274	850	0x0002	Ocari	Ocari	Ocari Beacon, Coord: 6, TlT0: 1350, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
39.412191	851	0x0004	Ocari	Ocari	Ocari Beacon, Coord: 6, TlT0: 1350, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
39.422114	852	0x0005	Ocari	Ocari	Ocari Beacon, Coord: 6, TlT0: 1350, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
39.432039	853	0x0003	Ocari	Ocari	Ocari Beacon, Coord: 6, TlT0: 1350, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
39.452046	854	0x0005	0x0001	IEEE 802	Unknown Command
39.452206	855			IEEE 802	Ack
39.458148	856	0x0003	Broadcast	Opera	Opera Hello, SeqNum: 39, Sender: 0x0003, Sym: 4, Asym: 0
39.461115	857	0x0004	Broadcast	Opera	Opera Hello, SeqNum: 38, Sender: 0x0004, Sym: 4, Asym: 0
39.464726	858	0x0001	Broadcast	Opera	Opera Hello, SeqNum: 43, Sender: 0x0001, Sym: 4, Asym: 0
39.468912	859	0x0001	Broadcast	Opera	Opera STC, SeqNum: 12, Sender: 0x0001, Tree SeqNum: 1, Parent: 0x0001, Cost: 0, To Be Colored, Stable
39.508925	860	0x0005	Broadcast	Opera	Opera Hello, SeqNum: 37, Sender: 0x0005, Sym: 4, Asym: 0
39.513106	861	0x0005	Broadcast	Opera	Opera STC, SeqNum: 12, Sender: 0x0005, Tree SeqNum: 1, Parent: 0x0001, Cost: 1, To Be Colored, Stable
39.878152	862	0x0001	Ocari	Ocari	Ocari Beacon, Coord: 5, TlT0: 1150, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
39.890356	863	0x0002	Ocari	Ocari	Ocari Beacon, Coord: 5, TlT0: 1150, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
[Frame 854: 99 bytes on wire (792 bits), 99 bytes captured (792 bits) on interface 0]					
[Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)]					
[Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)]					
[User Datagram Protocol, Src Port: zep (17754), Dst Port: zep (17754)]					
[ZigBee Encapsulation Protocol, Channel: 26, Length: 25]					
[IEEE 802.15.4 Command, Dst: 0x0001, Src: 0x0005]					
[Frame Control Field: Command (0x8823)]					
Sequence Number: 179					
Destination PAN: 0x0f0f					
Destination: 0x0001					
Source PAN: 0x0f0f					
Source: 0x0005					
[Extended Source: d7:46:a8:ca:35:681d:cc (d7:46:a8:ca:35:68:1d:cc)]					
[Origin: 62]					
Command Identifier: Unknown (0x18)					
[Data (11 bytes)]					
Data: 836358794c6242be010500					

Figure 4.6: Simulation output produced by SNOOPS and viewed with Wireshark showing a command frame (highlighted) sent with an unknown command identifier.

node cannot pick the same color as its 1 hop and 2 hop neighbors.

2. A node can pick its color if it has the highest priority among its conflicting neighbors. The priority is based on a node's number of descendants.

The algorithm calculating the node's priority, its neighbors' priorities and the color is called OSERENA and is handled at the network layer level. Here is an example of experimentation where OSERENA does not pick the node color correctly: 2 nodes that are 1-hop neighbors pick the same color (color number 1).

In the proposed simulation, there are 7 nodes all running on detailed node platforms with QEMU. One node is running the *Ocari_CPAN.elf* executable file. It is the network coordinator, which has 1 as a short address. The 6 other nodes are running the *Ocari_DEVICE.elf* executable file. The network topology that is simulated is shown in Figure 4.8. All nodes are 1-hop neighbors, except for the node with short address 6, which can only exchange packets bidirectionally with node 5. The priority of the CPAN is 7 (1 + 6 descendants). All the other nodes have a priority of 1, except for node 5, which has a priority of 2 (1 + 1 descendant). This behavior can be observed in Figure 4.9, which is the simulation output log file viewed on Wireshark. According to this log file, the node with short address 2 and the node with short address 5 pick the color 1 at the same coloring iteration, as they are both uncolored at the precedent iteration. The node with color 0 is the CPAN and the nodes with color 255 are uncolored nodes. Knowing that node 5 had a priority of 2 and node 2 has a priority of 1, node 5 should pick its color first, as it has the highest priority. As can be seen in Figure 4.10 in the highlighted field of the color message from node 2, we can see that node 2 considers node 5 a neighbor.

The bug (2 neighbors picking the same color) therefore comes from the node with address 2. In order to investigate this issue, we connected GDB to node 2 through a remote stub and restarted the simulation to understand why node 2 had picked its color when it was not supposed to as its priority was lower than that of node 5.

```

(gdb) l
476                                     case MACARI_CMD_ASSOC_RELAY_REQ:
477                                     macari_assoc_relay_ind(b_ptr);
478                                     processed_in_not_transient = true;
479                                     break;
480                                     case MACARI_CMD_DISASSOC_RELAY_REQ:
481                                     macari_disassoc_relay_ind(b_ptr);
482                                     processed_in_not_transient = true;
483                                     break;
484     #if (MAC_PAN_ID_CONFLICT_AS_PC == 1)
485                                     case PANIDCONFLICTNOTIFICATION:
(gdb) backtrace
#0 process_data_ind_not_transient (f_ptr=0x200016a8 <buf_pool+2112>,
  b_ptr=0x20002a4c <buf_header+96>)
  at ./OcarIBML/MAC/MAC/Src/mac_data_ind.c:481
#1 mac_process_tal_data_ind (msg=0x20002a4c <buf_header+96> "\250\026")
  at ./OcarIBML/MAC/MAC/Src/mac_data_ind.c:168
#2 0x00406660 in dispatch_event (event=<optimized out>)

```

Figure 4.7: View of the terminal in which the remote debugger is launched - tracing the coordinator's stack's code to find out how the unknown command is interpreted.

After tracing the code's execution, we found out that the decision to color a node and the choice of its color is done in the *serena_update_color* method, which is called from the *serena_internal_generate_message* method, which generates the color message. We placed a breakpoint on the latter method to catch its call as it generated the message picking the wrong color. In this method, the highest priority of uncolored neighbors is computed through the *serena_compute_max_prio* method. Basically, first for 1-hop, then 2-hop and 3-hop neighbors, for every neighbor, if it is not yet colored, the maximum priority is updated depending on that neighbor's priority value. We realized while tracing the stack's code's execution that the priority of node 5 was not taken into account in the update of the maximum priority at that iteration.

In Figure 4.11, we can observe that the priority is not taken into account because in the method *has_color*, the condition testing if node 5's address is part of the last implicit colored neighbor table returns true. Indeed, when displaying the node's last implicit colored structure, we can see in the last highlighted section in Figure 4.11 that it contains nodes 5 and 2, that are not in fact colored at that point. Because node 5's priority is ignored, node 2 considered that it should be the next node to be colored, and picks the smallest unused color (color 1).

From studying the code, we finally found that nodes are considered implicitly colored if they disappear from neighbor nodes priority lists in the color messages.

Using Wireshark, we found that nodes 2 and 5 disappeared from node 3's 1-hop neighbor's uncolored priority list as seen in Figure 4.12 in the detail of frame number 847 (1-hop Neighbor Max Priorities field).

By using a remote debugger on node 3, we figured out why the nodes had disappeared from its priority lists.

Node 3 had itself considered nodes 2 and 5 implicitly colored because it interpreted an erroneous frame. By setting a breakpoint on the method *notify_implicit_node_coloring*, we were able to observe the frame that was being interpreted by the protocol by printing its buffer table. The frame recorded in the buffer contained the beginning of the bytes of a Color Message, and the ending bytes of a Hello message. This is why node 3 considered that nodes 2 and 5 were colored, it interpreted wrong information.

This error in the frame data happened because while the node platform's SPI was copying the RF transceiver's frame buffer byte by byte to an address in the platform's memory pointed by the buffer pointer, another frame was received (Hello), and the color frame was replaced in the buffer. In conclusion, two frames were received at a too short interval.

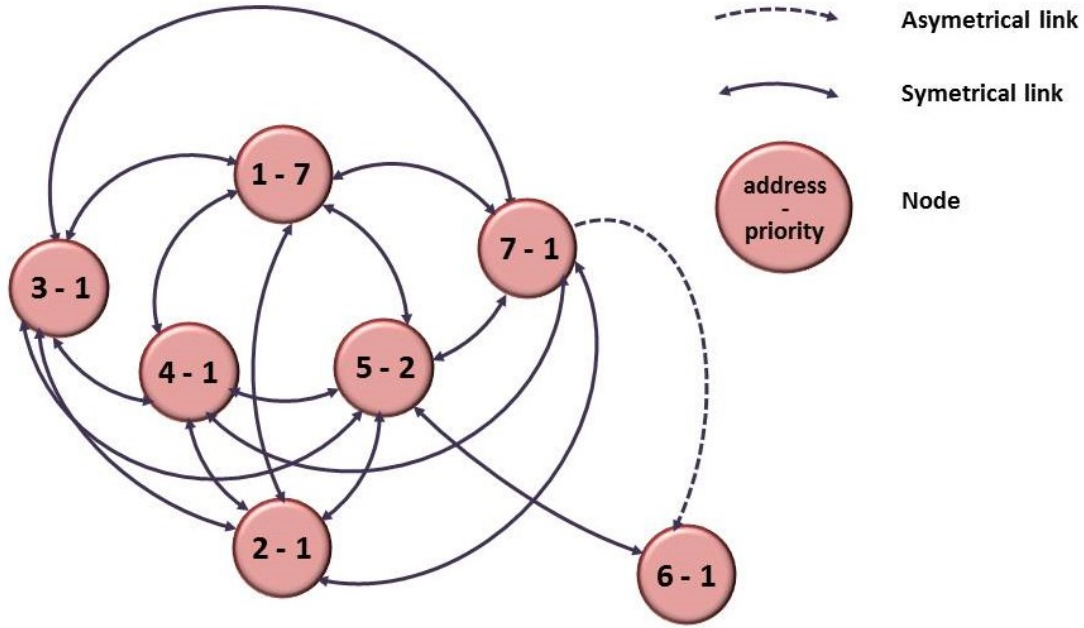


Figure 4.8: Simulated topology: 7 nodes, all nodes are 1-hop neighbors except for the node with short address 6.

Time	No.	Source	Destination	Protocol	Info
42.687313	853	0x0005		Ocari	Ocari Beacon, Coord: 7, TlT0: 1550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
42.699405	854	0x0006		Ocari	Ocari Beacon, Coord: 7, TlT0: 1550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
42.707153	855	0x0007		Ocari	Ocari Beacon, Coord: 7, TlT0: 1550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
42.746448	856	0x0006	Broadcast	Opera	Opera Hello, SeqNum: 27, Sender: 0x0006, Sym: 1, Asym: 1
42.750655	857	0x0006	Broadcast	Opera	Opera Color, Seq Num: 4, Sender: 0x0006, Tree Seq Num: 0, Max Color: 0, Color: 255, Priority: 1
42.753103	858	0x0003	Broadcast	Opera	Opera Hello, SeqNum: 30, Sender: 0x0003, Sym: 5, Asym: 0
42.768203	859	0x0007	Broadcast	Opera	Opera Color, Seq Num: 5, Sender: 0x0007, Tree Seq Num: 0, Max Color: 0, Color: 255, Priority: 1
42.768908	860	0x0001	Broadcast	Opera	Opera Hello, SeqNum: 34, Sender: 0x0001, Sym: 5, Asym: 0
42.771610	861	0x0005	Broadcast	Opera	Opera Hello, SeqNum: 28, Sender: 0x0005, Sym: 6, Asym: 0
42.773126	862	0x0001	Broadcast	Opera	Opera Color, Seq Num: 6, Sender: 0x0001, Tree Seq Num: 0, Max Color: 0, Color: 0, Priority: 7
42.774401	863	0x0002	Broadcast	Opera	Opera Color, Seq Num: 5, Sender: 0x0002, Tree Seq Num: 0, Max Color: 0, Color: 1, Priority: 1
42.775836	864	0x0005	Broadcast	Opera	Opera Color, Seq Num: 5, Sender: 0x0005, Tree Seq Num: 0, Max Color: 0, Color: 1, Priority: 2
42.783154	865	0x0004	Broadcast	Opera	Opera Hello, SeqNum: 29, Sender: 0x0004, Sym: 5, Asym: 0
42.787362	866	0x0004	Broadcast	Opera	Opera Color, Seq Num: 5, Sender: 0x0004, Tree Seq Num: 0, Max Color: 0, Color: 255, Priority: 1
43.206875	867	0x0001		Ocari	Ocari Beacon, Coord: 7, TlT0: 1550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
43.218977	868	0x0002		Ocari	Ocari Beacon, Coord: 7, TlT0: 1550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01

Figure 4.9: Simulation output viewed with Wireshark - 2 nodes pick the same color (color 1).

These are frames number 836 and 837 in Figure 4.13.

According to the AT86RF233 datasheet, in the section on data management of the frame buffer, it states that "Data in Frame Buffer (received data or data to be transmitted) remains valid as long as:

- No new frame or other data are written into the buffer over SPI
- No new frame is received (in any BUSY_RX state)
- No state change into SLEEP or DEEP_SLEEP state is made
- No RESET took place

By default there is no protection of the Frame Buffer against overwriting. Therefore, if a frame is received during a Frame Buffer read access of a previously received frame, an interrupt (IRQ_6 standing for TRX_UR) is issued and the stored data might be overwritten."

We can observe in the RF transceiver module that TRX_UR is masked by OCARI, according to data in the irq_mask register.

Time	No.	Source	Destination	Protocol	Info
41.564446	825	0x0005		Ocari	Ocari Beacon, Coord: 7, TlT0: 1550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
41.576537	826	0x0006		Ocari	Ocari Beacon, Coord: 7, TlT0: 1550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
41.584285	827	0x0007		Ocari	Ocari Beacon, Coord: 7, TlT0: 1550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
41.600713	828	0x0002	Broadcast	Opera	Opera Color, Seq Num: 4, Sender: 0x0002, Tree Seq Num: 0, Max Color: 0, Color: 255, Priority: 1
41.609410	829	0x0003	Broadcast	Opera	Opera Hello, SeqNum: 29, Sender: 0x0003, Sym: 5, Asym: 0
41.617621	830	0x0006	Broadcast	Opera	Opera Hello, SeqNum: 26, Sender: 0x0006, Sym: 1, Asym: 1
41.621823	831	0x0005	Broadcast	Opera	Opera Color, Seq Num: 3, Sender: 0x0005, Tree Seq Num: 0, Max Color: 0, Color: 255, Priority: 1
41.624654	832	0x0005	Broadcast	Opera	Opera Hello, SeqNum: 27, Sender: 0x0005, Sym: 6, Asym: 0
41.628877	833	0x0005	Broadcast	Opera	Opera Color, Seq Num: 4, Sender: 0x0005, Tree Seq Num: 0, Max Color: 0, Color: 255, Priority: 2

Sequence Number: 4
Strategic Address: 0x0000
Tree Sequence Number: 0
Max Color: 0
Color: 255
Priority: 1
1-hop Neighbor Max Priorities: 4
Neighbor address: 0x0001, Priority: 7
Neighbor address: 0x0005, Priority: 2
Neighbor address: 0x0003, Priority: 1
Neighbor address: 0x0004, Priority: 1
2-hop Neighbor Max Priorities: 3

0020	00	01	45	5a	45	5a	00	5a	fe	69	45	58	02	01	1a	00	..EZEZ.Z.iEX...
0030	0a	00	6b	00	00	00	00	7d	30	4f	cf	00	00	04	f8	00	..k....} 00.....
0040	00	00	00	00	00	00	00	00	00	32	41	88	36	0f	0f	ff2A.6...
0050	ff	02	00	00	43	24	02	00	00	00	00	00	00	ff	01	04CS.....
0060	07	01	00	d2	05	00	01	03	00	01	04	00	03	07	01	00
0070	02	05	00	01	02	00	00	00	00	04	ff	ff				

Figure 4.10: Simulation output viewed in Wireshark - highlighted Hello message from node 2 shows that node 5 is considered as a neighbor .

The transceiver's data sheet also states that "the microcontroller should check the transferred frame data integrity by an FCS check." The FCS is the Frame Check Sequence which contains a 16 bit CRC (Cyclic Redundancy Check), it is the last byte of a frame. However, the FCS is not checked by OCARI but only by the radio filter.

In reality, the Color frame and the Hello frame were sent so close to each other that the beginning of the Hello frame transmission would have interfered with the ending of the Color frame transmission. However, SNOOPS sends the packet instantly, after waiting the delay required to transmit each of its symbols. The received frame would have been erroneous and the RF transceiver should have filtered it out if the CRC was wrong. However, there is always a chance that an erroneous frame may not be filtered in the unlikely but possible case that the CRC, which is a 16-bit value (65536 possible values), be compatible. This will lead to the same type of bug.

This bug does not stem from a mistake in the protocol implementation, but it could be more robust by bringing several enhancements:

- Check the FCS of each frame it interprets
- Enable the RF transceiver's interrupt to notify the CPU that another frame is received during the frame buffer access to read a previously received frame
- Not consider a node implicitly colored without knowing the color chosen by that node

4.1.4 Replay of a testbed scenario

An experimentation was led to see if we can repeat using SNOOPS a scenario recorded from physical nodes on a testbed. In fact, this testbed experimentation does not lead to a bug. The objective here is to show that the frame generator is capable of generating frames to stimulate a node in the same way it was stimulated on testbed.

The testbed was composed of 4 nodes. One node, the coordinator, is flashed with the Ocari_CPAN.elf executable file, the others are flashed with the Ocari_DEVICE.elf executable file. The nodes are integrated into industrial equipment as seen in Figure 4.14,

```

Fichier  Édition  Affichage  Rechercher  Terminal  Aide
424                                     *address)){
425                                     // XXX: address
426
427                                     return HIPSENS_TRUE;
(gdb) n
422                                     if (!IS_PRIORITY_NONE(state->last_implicit_colored[i].priority)
(gdb)
423                                     && hipsens_address_equal(state->last_implicit_colored[i].address,
(gdb)
421                                     for (i=0;i<MAX_IMPLICIT_COLORED; i++)
(gdb)
422                                     if (!IS_PRIORITY_NONE(state->last_implicit_colored[i].priority)
(gdb)
423                                     && hipsens_address_equal(state->last_implicit_colored[i].address,
(gdb)
427                                     return HIPSENS_TRUE;
(gdb)
432     }
(gdb)
serena_compute_max_prio (state=state@entry=0x20002b84 <opera+664>,
    local_max2_prio1=local_max2_prio1@entry=0x20008548,
    local_max2_prio2=local_max2_prio2@entry=0x2000853c,
    local_max_prio3=local_max_prio3@entry=0x20008534)
    at ./OcariBML/OCARI/NWK/OPERA/Src/hipsens-osserena.c:773
773     switch (j){
(gdb) p state->last_implicit_colored
$270 = {{address = "\001", priority = {value = 7 '\a'}}, {address = "\005",
    priority = {value = 2 '\002'}}, {address = "\002", priority = {
    value = 1 '\001'}}, {address = "\000\001", priority = {
    value = 2 '\002'}}, {address = "\000S", priority = {value = 7 '\a'}}, {
    address = "\000", priority = {value = 4 '\004'}}, {address = "\000",

```

Figure 4.11: View of the terminal in which the remote debugger is launched - tracing the stack's code to find the bug, due to node 5 being considered implicitly colored.

Time	No.	Source	Destination	Protocol	Info
42.145/14	845	0x0007	Ocari	Ocari Beacon	Coord: 7, TIT0: 1550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
42.167875	846	0x0002	Broadcast	Opera	Opera Hello, SeqNum: 33, Sender: 0x0002, Sym: 5, Asym: 0
42.182971	847	0x0003	Broadcast	Opera	Opera Color, Seq Num: 4, Sender: 0x0003, Tree Seq Num: 0, Max Color: 0, Color: 255, Priority: 1
42.221439	848	0x0007	Broadcast	Opera	Opera Hello, SeqNum: 26, Sender: 0x0007, Sym: 5, Asym: 0
42.645452	849	0x0001	Ocari	Ocari Beacon	Coord: 7, TIT0: 1550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01


```

Color: 255
Priority: 1
1-hop Neighbor Max Priorities: 2
  Neighbor address: 0x0004, Priority: 1
  Neighbor address: 0x0007, Priority: 1
2-hop Neighbor Max Priorities: 3
  Neighbor address: 0x0005, Priority: 2
  Neighbor address: 0x0002, Priority: 1
  Neighbor address: 0x0003, Priority: 1
1-hop Neighbor Color Count: 1, Colors, 0

```

Figure 4.12: Output log viewed in Wireshark where we can see the Hello message (highlighted) with wrong priority list for 1-hop neighbors.

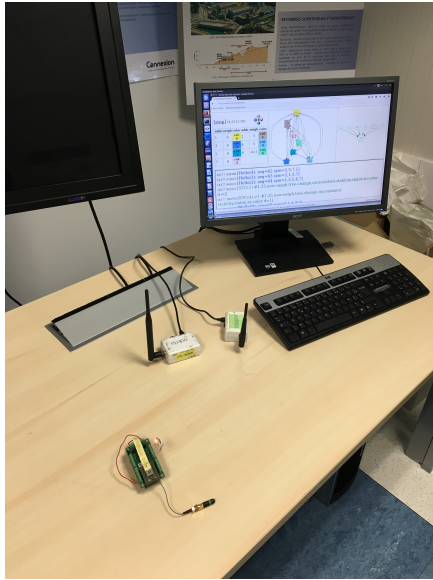
except for the CPAN. Sniffers are connected to a computer, and the frames received by the sniffer are recorded in a pcap format log using Wireshark.

The network topology is shown in Figure 4.15. Node 1 is the network coordinator (CPAN), and this will be the emulated node in SNOOPS. Its only neighbor, as seen on Figure 4.15, is node 2. Therefore, the frames produced by the frame generator in SNOOPS to repeat this scenario are frames produced by node 2.

Node 2's short address and long address are specified in the frame generator (which can be found in the log), as an unfiltered node. We also indicate that the CPAN is the emulated node. In its initialization stage, the frame generator runs through the pcap log packets until it finds the association request packet from node 2. Then the generator waits for a beacon of the emulated node which is running the Ocari_CPAN.elf executable file. When the beacon is received, the association request frame is sent and the generator starts waiting for an association response. Once the emulated node sends it, the generator waits for the next beacon from the emulated CPAN, then starts running through the pcap file and sends all packets produced by node 2 to the emulated node. The time to wait before sending the next packet for the generator is calculated as the difference between

Time	No.	Source	Destination	Protocol	Info
41.624654	832	0x0005	Broadcast	Opera	Opera Hello, SeqNum: 27, Sender: 0x0005, Sym: 6, Asym: 0
41.628977	833	0x0005	Broadcast	Opera	Opera Color, Seq Num: 4, Sender: 0x0005, Tree Seq Num: 0, Max Color: 0, Color: 255,
41.652203	834	0x0001	Broadcast	Opera	Opera Hello, SeqNum: 33, Sender: 0x0001, Sym: 5, Asym: 0
41.656422	835	0x0001	Broadcast	Opera	Opera Color, Seq Num: 5, Sender: 0x0001, Tree Seq Num: 0, Max Color: 0, Color: 0, R
41.669139	836	0x0007	Broadcast	Opera	Opera Color, Seq Num: 4, Sender: 0x0007, Tree Seq Num: 0, Max Color: 0, Color: 255,
41.669238	837	0x0004	Broadcast	Opera	Opera Hello, SeqNum: 28, Sender: 0x0004, Sym: 5, Asym: 0
41.673461	838	0x0004	Broadcast	Opera	Opera Color, Seq Num: 4, Sender: 0x0004, Tree Seq Num: 0, Max Color: 0, Color: 255,

Figure 4.13: Simulation output viewed with Wireshark - Frame 837 is sent 99 microseconds after frame 836.



(a) CPAN node and sniffers connected to computer and network analyzer



(b) Industrial equipment with integrated OCARI sensor nodes

Figure 4.14: A small testbed setup with sensor nodes communicating using the OCARI protocol.

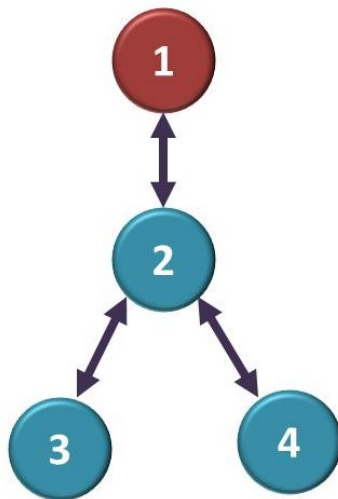


Figure 4.15: Topology of the network.

the timestamps of the last packet sent and the next one to send.

Some simulation results can be seen in Figures 4.16 and 4.17.

Figure 4.16 shows traces recorded from the testbed experiment during the association of node 3 to the network. In Figure 4.16a, we can see an association request frame from node 3 to node 2, followed by a frame from node 2 to node 1. Although this frame is an

association relay request type command, it is displayed as an unknown type command. The OCARI dissector in Wireshark may not recognize the command, but the dissector in SNOOPS does and the command is displayed correctly in the terminal output. The CPAN answers the association relay request frame with an association relay response frame to node 2, also displayed as an unknown command by Wireshark. Node 2 then sends an association response frame to node 3.

Figure 4.16b shows the scenario repeated in SNOOPS, where frames from node 2 are produced by the frame generator, and frames from node 1 are produced by the emulated node. We also see in this Figure the frames related to node 3's association to the network. The same association relay request frame that was recorded from the testbed (frame 60 of Figure 4.16a) is sent in SNOOPS to the CPAN (frame 57 of Figure 4.16b). The CPAN answers with an association relay request (displayed as unknown command). The next frame is the command frame of the association response from node 2 to node 3. The Hello messages of the CPAN show that it considers nodes 2 and 3 associated to its network.

Time	No.	Source	Destination	Protocol	Info
9.400895	55	0x0001		Ocarl	Ocarl Beacon, Coord: 2, TlTO: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
9.410523	56	0x0002		Ocarl	Ocarl Beacon, Coord: 2, TlTO: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
9.474248	57	0x0002	Broadcast	Opera	Opera Hello, SeqNum: 2, Sender: 0x0002, Sym: 1, Asym: 0
9.595345	58	0x0001		Ocarl	Ocarl Beacon, Coord: 2, TlTO: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
9.604974	59	0x0002		Ocarl	Ocarl Beacon, Coord: 2, TlTO: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
9.615599	60	89:c6:a7:d3:c9:96:37:1d	0x0002	IEEE 802	Association Request
9.616159	61			IEEE 802	Ack
9.617088	62	0x0002	0x0001	IEEE 802	Unknown Command
9.617248	63			IEEE 802	Ack
9.618329	64	0x0001	0x0002	IEEE 802	Unknown Command
9.618488	65			IEEE 802	Ack
9.619458	66	44:8b:76:f7:17:30:54:ad	89:c6:a7:d3:c9:96:37:1d	IEEE 802	Association Response, PAN: 0x0f0f Addr: 0x0003
9.619617	67			IEEE 802	Ack
9.698492	68	0x0001	Broadcast	Opera	Opera Hello, SeqNum: 3, Sender: 0x0001, Sym: 1, Asym: 0
9.789861	69	0x0001		Ocarl	Ocarl Beacon, Coord: 3, TlTO: 750, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
9.799507	70	0x0002		Ocarl	Ocarl Beacon, Coord: 3, TlTO: 750, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
9.809134	71	0x0003		Ocarl	Ocarl Beacon, Coord: 3, TlTO: 750, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01

(a) Frames recorded from the testbed experiment

Time	No.	Source	Destination	Protocol	Info
9.235070	54	0x0002	Broadcast	Opera	Opera Hello, SeqNum: 2, Sender: 0x0002, Sym: 1, Asym: 0
9.353263	55	0x0001		Ocarl	Ocarl Beacon, Coord: 2, TlTO: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
9.365796	56	0x0002		Ocarl	Ocarl Beacon, Coord: 2, TlTO: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
9.377910	57	0x0002	0x0001	IEEE 802	Unknown Command
9.378070	58			IEEE 802	Ack
9.379159	59	0x0001	0x0002	IEEE 802	Unknown Command
9.380280	60	44:8b:76:f7:17:30:54:ad	89:c6:a7:d3:c9:96:37:1d	IEEE 802	Association Response, PAN: 0x0f0f Addr: 0x0003
9.411793	61	0x0001	Broadcast	Opera	Opera Hello, SeqNum: 3, Sender: 0x0001, Sym: 1, Asym: 0
9.547785	62	0x0001		Ocarl	Ocarl Beacon, Coord: 3, TlTO: 750, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
9.560329	63	0x0002		Ocarl	Ocarl Beacon, Coord: 3, TlTO: 750, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
9.583234	64	0x0002	Broadcast	Opera	Opera Hello, SeqNum: 3, Sender: 0x0002, Sym: 1, Asym: 0

(b) Frames from SNOOPS output log

Figure 4.16: Association of node 3 through node 2 .

Figure 4.17 shows packet exchanges of the device nodes sending their sensed data to the CPAN which can happen after the nodes have been colored. Nodes 3 and 4 first send their data packets to node 2 (frames 597 and 599 of Figure 4.17a), then node 2 relays its own sensed data, as well as data from nodes 3 and 4 to the CPAN (frames 601, 603 and 605).

Figure 4.17b shows the same frames which were extracted from the testbed pcap file and generated by SNOOPS's frame generator. It is worth noticing that the same frames can be observed because the same random values are sent as sensed data.

Although we do not reproduce a bug found with a testbed, we showed with this experiment that it is possible to replay frames from a pcap format log file and repeat a scenario. A few values may vary though for the node that is emulated as some values are random (random backoff, frame sequence number...).

4.1.5 Network destabilization by a false node

In this section, we describe a particular scenario where an attack on the network is simulated to see how the nodes running the OCARI protocol binary stack reacts.

Time	No.	Source	Destination	Protocol	Info
35.426444	588	0x0001		Ocari	Ocari Beacon, Coord: 4, TlTl: 1650, Contention: 200, Colored: 100, Color Max: 4, Bitmap: 0x01
35.436094	589	0x0002		Ocari	Ocari Beacon, Coord: 4, TlTl: 1650, Contention: 200, Colored: 100, Color Max: 4, Bitmap: 0x01
35.445733	590	0x0003		Ocari	Ocari Beacon, Coord: 4, TlTl: 1650, Contention: 200, Colored: 100, Color Max: 4, Bitmap: 0x01
35.455645	591	0x0004		Ocari	Ocari Beacon, Coord: 4, TlTl: 1650, Contention: 200, Colored: 100, Color Max: 4, Bitmap: 0x01
35.513161	592	0x0004	Broadcast	Opera	Opera Hello, SeqNum: 36, Sender: 0x0004, Sym: 1, Asym: 0
35.513421	593	0x0002	Broadcast	Opera	Opera STC, SeqNum: 10, Sender: 0x0002, Tree SeqNum: 1, Parent: 0x0001, Cost: 1, To Be Colored, ST
35.531413	594	0x0001	Broadcast	Opera	Opera Hello, SeqNum: 41, Sender: 0x0001, Sym: 1, Asym: 0
35.548917	595	0x0003	Broadcast	Opera	Opera Hello, SeqNum: 37, Sender: 0x0003, Sym: 1, Asym: 0
35.550031	596	0x0003	Broadcast	Opera	Opera STC, SeqNum: 10, Sender: 0x0003, Tree SeqNum: 1, Parent: 0x0002, Cost: 2, To Be Colored, ST
35.719837	597	0x0004	0x0001	Connexion Origin	Address: 6b:57:2a:59:1c:b5:4b:8d, Random Value: 67,326, Quality: Good
35.719997	598			IEEE 802. Ack	
35.721928	599	0x0003	0x0001	Connexion Origin	Address: 1d:37:96:c9:d3:a7:c6:89, Random Value: 84,054, Quality: Good
35.722088	600			IEEE 802. Ack	
35.784323	601	0x0002	0x0001	Connexion Origin	Address: ad:54:30:17:f7:76:8b:44, Random Value: 76,288, Quality: Good
35.784483	602			IEEE 802. Ack	
35.786056	603	0x0004	0x0001	Connexion Origin	Address: 6b:57:2a:59:1c:b5:4b:8d, Random Value: 67,326, Quality: Good
35.786216	604			IEEE 802. Ack	
35.787794	605	0x0003	0x0001	Connexion Origin	Address: 1d:37:96:c9:d3:a7:c6:89, Random Value: 84,054, Quality: Good
35.787954	606			IEEE 802. Ack	
35.958198	607	0x0001		Ocari	Ocari Beacon, Coord: 4, TlTl: 1650, Contention: 200, Colored: 100, Color Max: 4, Bitmap: 0x01

(a) Frames recorded from the testbed experiment

35.274243	327	0x0002	Broadcast	Opera	Opera STC, SeqNum: 10, Sender: 0x0002, Tree SeqNum: 1, Parent: 0x0001, Cost: 1, To Be Colored, ST
35.297255	328	0x0001	Broadcast	Opera	Opera Hello, SeqNum: 41, Sender: 0x0001, Sym: 1, Asym: 0
35.298362	329	0x0001	Broadcast	Opera	Opera STC, SeqNum: 10, Sender: 0x0001, Tree SeqNum: 1, Parent: 0x0001, Cost: 0, To Be Colored, ST
35.545145	330	0x0002	0x0001	Connexion Origin	Address: ad:54:30:17:f7:76:8b:44, Random Value: 76,288, Quality: Good
35.545305	331			IEEE 802. Ack	
35.546878	332	0x0004	0x0001	Connexion Origin	Address: 6b:57:2a:59:1c:b5:4b:8d, Random Value: 67,326, Quality: Good
35.547038	333			IEEE 802. Ack	
35.548616	334	0x0003	0x0001	Connexion Origin	Address: 1d:37:96:c9:d3:a7:c6:89, Random Value: 84,054, Quality: Good
35.548776	335			IEEE 802. Ack	
35.589036	336	0x0001		Ocari	Ocari Beacon, Coord: 4, TlTl: 1050, Contention: 200, Colored: 100, Color Max: 1, Bitmap: 0x01
35.728677	337	0x0002		Ocari	Ocari Beacon, Coord: 4, TlTl: 1650, Contention: 200, Colored: 100, Color Max: 4, Bitmap: 0x01

(b) Frames from SNOOPS output log

Figure 4.17: Packets from nodes transferring sensed data to CPAN.

No.	Time	Source	Destination	Protocol	Info
474	24.195097	0x0001		Ocari	Ocari Beacon, Coord: 3, TlTl: 1250, Contention: 200, Colored: 100, Color Max: 3
475	24.207537	0x0002		Ocari	Ocari Beacon, Coord: 3, TlTl: 1250, Contention: 200, Colored: 100, Color Max: 3
476	24.219978	0x0003		Ocari	Ocari Beacon, Coord: 3, TlTl: 1250, Contention: 200, Colored: 100, Color Max: 3
477	24.245811	0x0001	Broadcast	Opera	Opera Hello, SeqNum: 42, Sender: 0x0001, Sym: 1, Asym: 0
478	24.259961	0x0003	Broadcast	Opera	Opera Hello, SeqNum: 40, Sender: 0x0003, Sym: 1, Asym: 0
479	24.420573	0x0002	0x0001	Connexion Origin	Address: 85:3f:4c:09:a1:cc:00:95, Random Value: 78,169, Quality: Good
480	24.420673			IEEE 802. Ack	
481	24.449789	0x0002	0x0001	Connexion Origin	Address: bb:02:26:2e:52:0b:81:37, Random Value: 17,509, Quality: Good
482	24.449889			IEEE 802. Ack	
483	24.454131	0x0003	0x0001	Connexion Origin	Address: 85:3f:4c:09:a1:cc:00:95, Random Value: 78,169, Quality: Good
484	24.454231			IEEE 802. Ack	
485	24.621632	0x0001		Ocari	Ocari Beacon, Coord: 3, TlTl: 1250, Contention: 200, Colored: 100, Color Max: 3
486	24.634073	0x0002		Ocari	Ocari Beacon, Coord: 3, TlTl: 1250, Contention: 200, Colored: 100, Color Max: 3
487	24.646514	0x0003		Ocari	Ocari Beacon, Coord: 3, TlTl: 1250, Contention: 200, Colored: 100, Color Max: 3
488	24.716770	0x0002	Broadcast	Opera	Opera Hello, SeqNum: 42, Sender: 0x0002, Sym: 2, Asym: 0
489	24.847110	0x0003	0x0001	Connexion Origin	Address: 85:3f:4c:09:a1:cc:00:95, Random Value: 76,410, Quality: Good
490	24.847210			IEEE 802. Ack	
491	24.876323	0x0002	0x0001	Connexion Origin	Address: bb:02:26:2e:52:0b:81:37, Random Value: 39,951, Quality: Good
492	24.876423			IEEE 802. Ack	
493	24.880665	0x0003	0x0001	Connexion Origin	Address: 85:3f:4c:09:a1:cc:00:95, Random Value: 76,410, Quality: Good
494	24.880765			IEEE 802. Ack	

Figure 4.18: Simulation output frame traces from 2 regular OCARI protocol cycles viewed with Wireshark.

In the scenario in question, 3 nodes running the OCARI protocol on full SAM3S platform models are instantiated, one is the network coordinator (CPAN) loaded with the `ocari_CPAN.elf` binary code, and the other two nodes are running the `ocari_DEVICE.elf` binary code. A fourth node is added to the simulation and acts as a node attacking the network. The topology of the network can be seen in Figure 4.19.

The simulation is set up so that the attacker node is a simple node that copies a frame from the beginning of the simulation, more specifically a Beacon type frame from node 2. After 25 seconds of simulation, the attacker nodes starts sending the Beacon frame copied from node 2 at constant intervals of 50 ms. The fake beacon is received by all three other nodes in the network.

The simulation results can be observed through the terminal where the simulation was launched. The frame exchanges during the simulation can also be viewed through Wireshark, as they are saved in a `pcap` format. Before the intervention of the attacker node, no anomaly is detected in the exchanges. In Figure 4.18, two cycles can be seen,

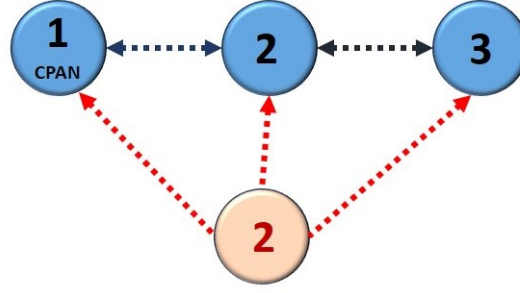


Figure 4.19: Network topology.

a cycle beginning with a beacon frame from the CPAN which are frames 474 and 485 in Figure 4.18. Those traces are recorded before the attacker node starts sending frames, with nodes 3 and 2 correctly relaying data to the CPAN, which are Connexion type frames with random data values for this particular application (e.g. frames 479 and 481).

After 25 seconds of simulation time, the attacker node starts sending Beacon frames pretending to be node 2. The simulation is stopped at 26.754397 seconds, as the observer in the simulator detects and indicates a "Hello" property violation from node 2. The property in question states that once nodes are associated to the CPAN, a Hello message must be sent every 2 cycles. In the frame traces observed on Wireshark, we can see that nodes 2 and 3 completely stop sending frames after 25.55445 seconds of simulation time. In Figure 4.20, we can observe the two last cycles (starting with frames 527 and 537) before the simulation stopped where no Hello messages are sent by nodes 2 and 3, causing the Hello property to be violated. The violation is detected as the last Beacon frame from the CPAN is received indicating the beginning of a new cycle without a Hello frame having been received.

526	25.900000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
527	25.901230	0x0001	Ocari	Ocari Beacon, Coord: 3, TlT0: 1250, Contention: 200, Colored: 100, Color Max: 3, Bitmap: 0x01
528	25.943008	0x0001	Broadcast Opera	Opera Hello, SeqNum: 44, Sender: 0x0001, Sym: 1, Asym: 0
529	25.950000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
530	26.000000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
531	26.050000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
532	26.100000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
533	26.150000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
534	26.200000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
535	26.250000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
536	26.300000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
537	26.327774	0x0001	Ocari	Ocari Beacon, Coord: 3, TlT0: 1250, Contention: 200, Colored: 100, Color Max: 3, Bitmap: 0x01
538	26.350000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
539	26.400000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
540	26.450000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
541	26.500000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
542	26.550000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
543	26.600000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
544	26.650000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
545	26.700000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
546	26.750000	0x0002	Ocari	Ocari Beacon, Coord: 2, TlT0: 550, Contention: 200, Colored: 100, Color Max: 0, Bitmap: 0x01
547	26.754397	0x0001	Ocari	Ocari Beacon, Coord: 3, TlT0: 1250, Contention: 200, Colored: 100, Color Max: 3, Bitmap: 0x01

Figure 4.20: Simulation traces from the two last OCARI protocol cycles before the property violation viewed with Wireshark.

In order to understand what went wrong with node 2, and why it stopped sending frames, we use GDB to determine which function of the `ocari_DEVICE.elf` code was being executed by node 2 when the simulation stopped. We have observed that node 2 was stuck in a while loop waiting for a timer to expire, and that the while loop was initiated by the `mac_wrapper_reset_MaCARI` function, indicating that the node could not handle the frames sent by the attacker node and had to reset itself.

This experiment is an example that demonstrates the ability of this simulation environment to facilitate testing protocols under more flexible scenarios and environmental

conditions. Indeed, adding such an attacker node in simulation is relatively easy and would be more complicated in a lab environment. The version of the OCARI protocol has been updated (with enhanced security features) since this test was performed.

4.2 SNOOPS simulation time

In this section we study the impact of different parameters on the simulation's execution time.

Here are the parameters investigated in this study:

- The global quantum which regulates the maximum time difference between SystemC and QEMU before a synchronization is performed
- The number of emulated nodes in the same simulation running the protocol's binary code
- The working frequency of the node's hardware platform
- TLM's Direct Memory Interface for the CPU model to access memory models instantiated in SystemC directly (without using TLM transactions) being enabled or disabled

To evaluate the impact of the number of emulated nodes on the simulation execution time, we ran 7 scenarios, simply modifying the number of emulated nodes in the same simulation from 1 to 7.

For each of these scenarios, we modified the value of the global quantum, which is the default time interval between successive quantum boundaries (local time offset of a SystemC process relative to the current simulation time). Scenarios are tested with global quanta set at 100 ns, 200 ns, 500 ns, 1000 ns, and 1500 ns. For more than 1500 ns, the lack of accuracy caused bugs in simulation.

We used the linux *time* command to determine the duration of execution of a particular command, and in this case, the time of SNOOPS's execution. When the execution completes, *time* reports how long the execution took in terms of user CPU time (amount of time that was spent in user mode), system CPU time (the amount of time spent in kernel mode waiting for the operating system to perform tasks for the command), and real time (total time is took to run the command).

We considered the CPU time, as it corresponds to the time the CPU spent actually executing the program. We set up SNOOPS to stop the simulation's execution using the method *sc_stop* when time in SystemC reached 10 seconds. For the emulation of only one node working at a 32 MHz frequency and with a global quantum of 100 ns, the simulation executed in 3 minutes and 36.409 seconds. In this case, the emulation takes roughly 21 times longer to execute than the same experiment would on a testbed. This might seem slow, but emulation holds several advantages over testbeds (e.g. tracing the stack's execution, repeatable experimentation) that can be worth the price of this longer testing time.

To compare simulation execution times, we decided to normalize the time results by dividing them all by the time to emulate one node with a global quantum of 100 ns (3 minutes and 36.409 seconds), meaning that for that scenario (1 node, 100 ns global quantum), the normalized simulation time is 1. The exact time the simulation takes is not important here, as the end of simulation at a SystemC time of 10 seconds was chosen randomly, and the CPU execution time may vary from one machine to another. The objective is rather to observe the relation between simulation times when different parameters vary.

We originally expected that the length of the global quantum would have an important impact on simulation time. A greater global quantum leads to longer time intervals between QEMU and SystemC synchronization, meaning fewer synchronizations, and should therefore reduce the simulation time. Figure 4.21 shows the relation between global quantum and the normalized simulation time for the scenarios with 1 and 2 emulated nodes. For 1 and 2 emulated nodes, changing the global quantum from 100 ns to 1000 ns (multiplying it by 10), only decreases the simulation time by about 6% in both cases.

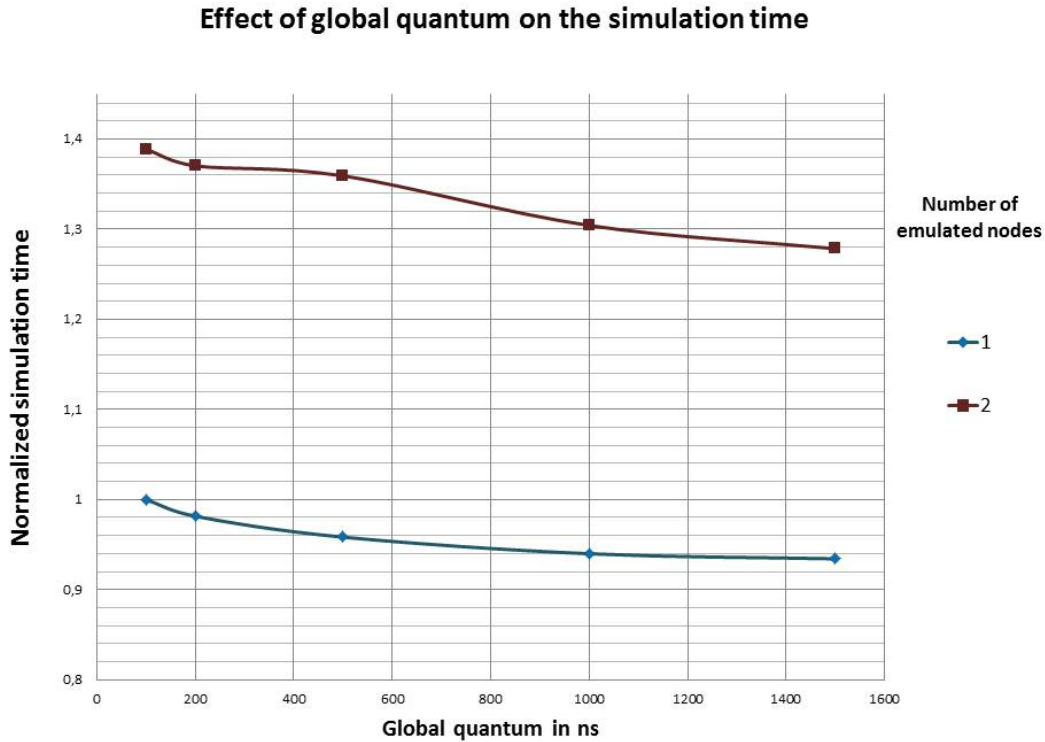


Figure 4.21: Normalized simulation execution time in relation to the global quantum for simulations with 1 and 2 emulated nodes.

Adding a second emulated node in simulation for a global quantum of 100 ns increases the simulation time by approximately 40% compared to a simulation with only one emulated node. Therefore, the number of emulated nodes has more impact on the simulation time than the global quantum.

Figure 4.22 shows the normalized simulation execution time (based on the emulation time of 1 node with a 100 ns global quantum), for 1 to 7 nodes in the same simulation. Each curve represents the results for a different value of global quantum.

An interesting outcome is that the simulation's execution time increases linearly with the number of emulated nodes. It takes about 4 times longer to emulate 7 nodes than to emulate 1.

We can also observe that the more nodes there are in simulation, the greater the effect of the global quantum will be on the execution time. Indeed, changing the global quantum from 100 ns to 1500 ns only saves 6% of simulation time for 1 emulated node, but about 15% of simulation time for 7 emulated nodes.

We can also observe that the difference of simulation time for a global quantum of 1000 ns and 1500 ns is very small as the two corresponding curves are almost superposed in Figure 4.22. While doing our other experimentations, we used a global quantum of

1000 ns, as it reduces simulation time almost as much as when using a global quantum of 1500 ns, while offering simulations with a better accuracy.

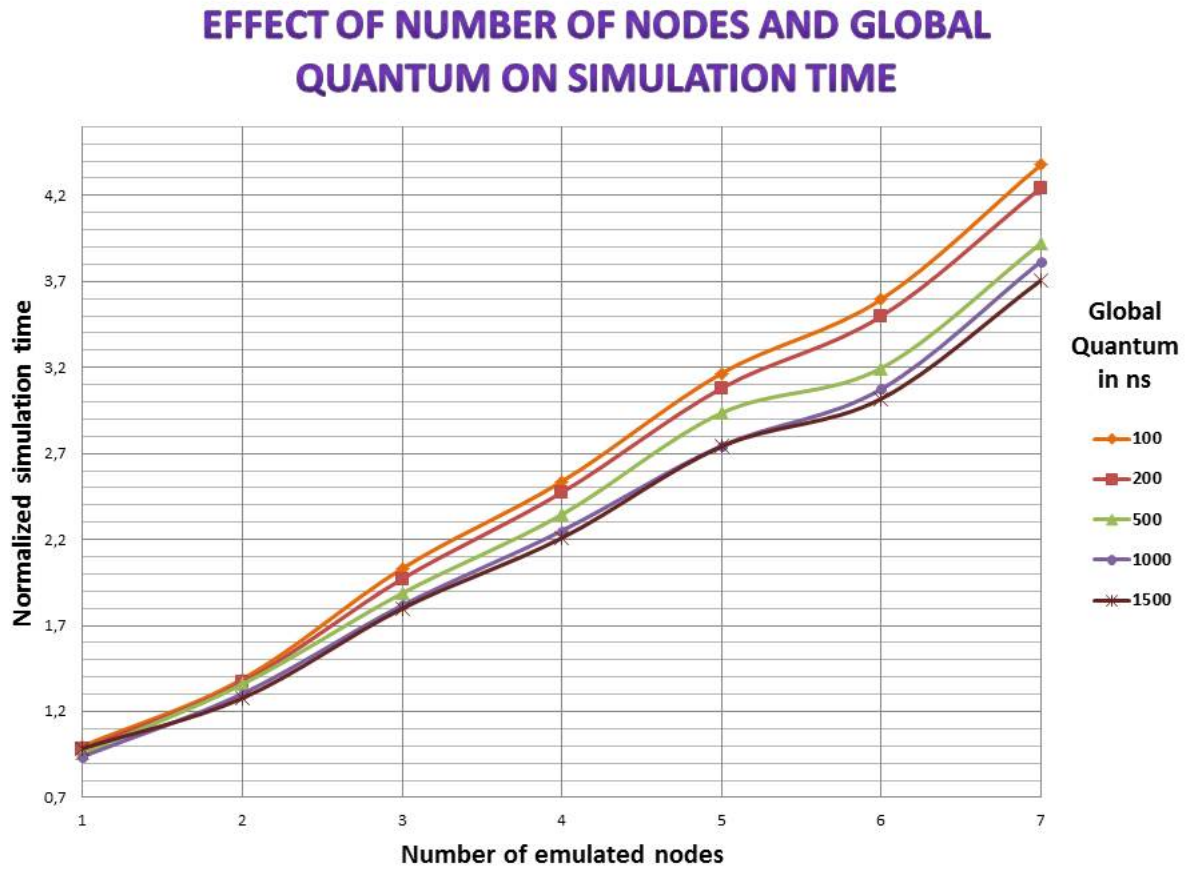


Figure 4.22: Impact of the number of emulated nodes and the global quantum on the simulation's execution time.

We compared the effect of disabling the TLM direct memory interface (DMI) that is used for the emulated CPU to access memories instantiated in SystemC without using TLM transactions. When the modeled CPU wrapped by TLMu accesses a memory in SystemC for the first time, it checks if DMI is enabled for that memory. If so, a method called *get_dmi_pointer* is called, and the memory modeled in SystemC as an array returns the pointer to the first value of this array. Then, for the subsequent memory accesses, instead of using a TLM blocking transaction, the memory is accessed directly by using the *memcpy* method. This access is faster than when using TLM transactions and does not block the CPU SystemC thread while the transaction is being handled.

Figure 4.23 shows a comparison of the normalized simulation's execution time when DMI is enabled and disabled for 1, 2 and 3 emulated nodes. For 1 emulated node, enabling DMI access saves about 20% of simulation time with respect to when it is disabled, it allows saving about 37.5% of the simulation time when two nodes are emulated, and finally, it allows saving about 40% of the simulation time for 3 emulated nodes. The fact that a smaller percentage of simulation time is saved by enabling DMI in the simulation where only one node is emulated can simply be explained by the fact that in that simulation, the node does not have any other node to interact with. Therefore, it does not receive any frames and has less processing to do, which means fewer memory accesses. A smaller

percentage of the simulation time is spent by this node on memory accesses than when there are more nodes in the simulation. To reduce the simulation time, we enabled DMI access to the flash and code memory regions in the platform model, as it does not change the result of the memory access if it is done directly. However, DMI is not enabled to access peripheral registers, as their access through TLM triggers reactions in their SystemC models (e.g. access to timer control registers can modify the timer's frequency).

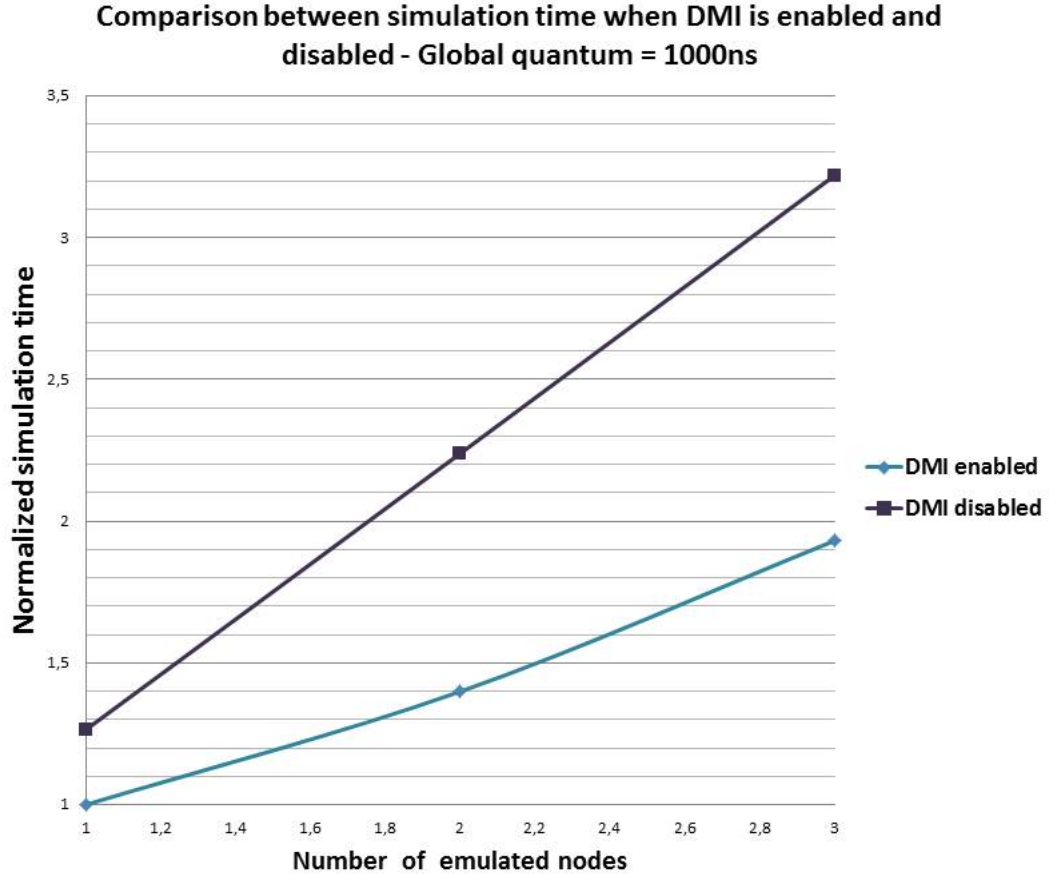


Figure 4.23: Impact of enabling the Direct Memory Interface on the simulation's execution time.

The relation between the SystemC time in simulation and the simulation's execution time varies based on the events to be handled in SystemC. SystemC is an event driven simulator, the time in SystemC advances according to the events and their timestamps. If a great number of events are supposed to happen in a SystemC time lapse of 1 second, it will take long to simulate, if no events happen in that time lapse, this 1 second time in SystemC will advance in an instant. This behavior can be seen by changing the frequency of the SystemC clock supplied to the platform models as Master clock. The events driven by the clock signals that happen inside the platform models do not change, but as the clock's period is longer, SystemC will increase its time more between each clock edge.

We tested the effect of modifying that main clock's frequency in a simulation with 2 emulated nodes with a 1000 ns global quantum. The clock's frequency was originally set to 32 MHz. As seen in Figure 4.24, by dividing it by 2 (16 MHz), the time to simulate 10 SystemC seconds changes from 4 minutes and 44.167 seconds to 2 minutes and 36.427 seconds. Therefore the execution time decreases by approximately 44% by dividing the frequency by 2. The reason is that less events happened in those 10 SystemC seconds.

However, the scenario does not go as far (fewer frames exchanged) as when the main frequency was set to 32 MHz.

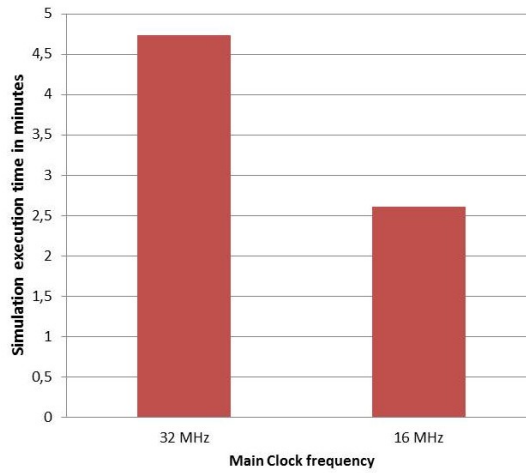


Figure 4.24: Impact of the frequency of the SystemC clock fed to the platform models on the simulation's execution time.

4.3 Conclusion

The various experimentations we have conducted show different aspects of SNOOPS capability in assisting with WSN protocol validation and verification.

Using SNOOPS, the following actions can be conducted:

- Introducing properties in simulation modeled in Light Esterel to check if these properties are validated or violated
- Tracing the stack's code's execution to trace bugs
- Reproducing experiments that have been led on a physical testbed
- Using high level models of nodes to generate frames to destabilize the network

Moreover, we investigated the impact of the global quantum, the number of emulated nodes, DMI, and the node platform's frequency on the simulation's execution time.

While these experimentations did not put in evidence major bugs of the protocol under test, they have led to advice on how to improve the protocol and make it more robust.

Conclusion and Perspectives

In this thesis, we have brought to light the need for adequate validation methods to guarantee the reliability of WSN communication protocols. Reliability is an essential criteria in order for WSN technology to be more widely adopted in areas such as the industry, military or medical sectors. We have studied various methods that can be used to verify and validate WSN communication protocols. After concluding that to this date, no method had shown to be optimal, we have decided to propose a new simulation framework to test protocols at a binary level: SNOOPS.

Features and advantages of the developed simulation framework

SNOOPS was developed to validate protocol implementations, it is therefore made to validate a protocol in its final stages of development, but is no substitute to the use of testbeds. It is a complementary validation method.

SNOOPS offers the possibility to execute the compiled protocol binary code on models of the nodes hardware platforms. Those models are based on the association of QEMU and SystemC, using TLMu as an interface. QEMU offers a wide range of CPU, hardware peripherals, and full platform models, and SystemC has the advantage of integrating both hardware and software modules at different abstraction levels, to perform HW/SW co-simulation. Therefore, the co-simulation of QEMU and SystemC offers the advantage of a great flexibility to model a wide range of hardware platforms, with reusable models, making it possible to develop (and reuse) component libraries.

SystemC allows simulating not only the HW/SW of a network node but also the communication environment in which the node operates. We developed a simple model of the channel for nodes to communicate using TLM transactions. A SystemC module (Radio Link) handles the topology of the network and produces a log file in pcap format to analyze frames exchanged using a network analyzer such as Wireshark.

An observer module analyzes ingoing and outgoing frames from a node running the protocol under test. It detects if protocol properties are violated and stops the simulation for the user to search for the bug that caused the violation using the debugging capabilities provided by the simulation environment (GDB). In order for properties to be detected as violated, we proposed a way to model protocol properties using a synchronous language, Light Esterel, and insert them into the observer after their compilation into C code.

SNOOPS also offers the possibility to use a frame generator, to reproduce scenarios conducted on testbeds that have led to bugs of undetermined origin. Scenarios led on testbeds are not always repeatable. By using the log file containing the frames exchanged during the testbed scenario, SNOOPS may reproduce the bug. If the bug reappears, the visibility of the stack's code's execution as well as the visibility of the state of the memories

and hardware peripherals is a great advantage to find the source of the bug.

The robustness of a protocol can be tested by changing parameters in the simulation environment, to test the limits of the protocol. For example, the user can test for which platform frequency range the protocol functions correctly. Slightly modifying the platform frequencies between several modeled nodes could allow the user to test the limits of the clock drift that the protocol can withstand.

Extending SNOOPS simulation environment with added features

There are a few areas in which SNOOPS could be improved:

- **The simulation's execution time** is relatively slow compared with the speed at which scenarios unfold on real testbeds, as we have seen in Chapter 4. Two main reasons can be raised to explain this relative slow simulation's execution time. First, it is due to the nodes being modeled at a very low level. Moreover, SystemC is executed as one process, and therefore on only one of the host machine's CPUs. By using TLMu as a wrapper, QEMU is instantiated as a separate SystemC thread, but in the same process as the SystemC simulation core. Several solutions could speed-up the simulation time. There is work in progress to make SystemC multi-threaded [1], which would allow the simulation to be executed on several CPUs of the host machine, hence decreasing the simulation's execution time. Another solution is porting SNOOPS codes on QBox [2] instead of TLMu which handles QEMU and SystemC communications and synchronization. QBox is also a SystemC wrapper for QEMU, but the difference with TLMu is that QEMU is executed in a different thread than the SystemC core. Therefore, all emulated nodes can be executed in different threads, on more than one host machine CPU, which would also decrease the simulation's execution time. QBox has not been used in this work because the source codes were not available when this thesis started (in 2014). Therefore we decided instead to develop our own simulation framework based on TLMu as an interface between QEMU and SystemC/TLM. The source codes for QBox are now available, and we are currently porting SNOOPS on QBox. We should then be able to measure the gain in simulation time and better evaluate the differences between QBox and TLMu.
- TLMu is based on a **version of QEMU-linaro** that dates back to 2011 (version 0.15.50-2011.10) and cannot be updated to include (merge) the latest improvements on QEMU as the git information on the original QEMU-linaro branch is unavailable. We tried to upgrade TLMu with the latest version of QEMU without using git. For that, we started to track all the differences between TLMu and the version of QEMU-linaro it was based on. Then, we investigated the volume of work to port all those differences on the latest version of QEMU. Unfortunately, we quickly understood that this task was not trivial and could last for months of work. As the objective was to investigate the validation capabilities of our simulation framework, we decided to keep going with TLMu based on this old version of QEMU. However, QBox works with one of the latest versions of QEMU and can be easily updated. Therefore, porting SNOOPS's codes on QBox could solve this problem.
- A **radio link module** (developed in SystemC) has been used to interconnect the SystemC node models, as well as to handle the network's topology and the links be-

tween the nodes. However, several improvements could be proposed for this module. As an example, it is important to model collisions as they can be a source of bug. We have shown in Chapter 4 that a bug occurs when 2 packets arrive at too short an interval from each other, when in real life these packets would just have collided. Modeling path loss, obstacles, reflections can also be improvements on the radio channel model, as they are environmental conditions that can affect the behavior of the protocol. A solution to improve the SNOOPS radio channel model could be to associate SNOOPS with SCNSL [3] which is also based on SystemC.

A possibility to use SNOOPS to test security aspects of WSN

Security is going to be an important aspect in the development of WSN technologies. Deploying nodes in unattended environments makes them vulnerable to a wide variety of attacks, and the node limitations in terms of memory and power supply render using conventional security solutions difficult or even impossible [4].

Standard security goals for WSN include data authentication, data integrity, data confidentiality and availability (a failure of the coordinator node can threaten the entire network).

We believe that SNOOPS is also well suited to test security aspects of wireless protocols. We have shown in Chapter 4 that it is possible to use a high level node model to generate malicious frames that destabilized the network. It is easier and faster to model this high level malicious node in SystemC, than to develop the code and flash a physical node in a testbed to produce this malicious behavior.

Examples of attacks that could be tested using SNOOPS are the following:

- Denial of service: an adversary attempts to subvert, disrupt, or destroy a network, or diminishes a network's capability to provide a service (e.g. Jamming, tampering)
- Message corruption: Modification of the content of a message by an attacker
- False node: an adversary could add a node to the network causing the injection of malicious data or preventing the passage of true data.
- Node Replication Attacks: an attacker adds a node to a WSN by copying the ID of an existing sensor node. The malicious node can severely disrupt the network's performance. Packets may be corrupted or even misrouted.
- Node outage: The network's robustness can be tested in the case where a node stops functioning, which could be caused by a physical attack where an attacker destroys a node.

Modeling more complex protocol properties and enhancing the observer module

We have shown that the observer module is capable of verifying protocol properties by observing frame exchanges in the simulation environment. We only modeled a few of OCARI's properties using Light Esterel to demonstrate the observer's potential. More properties could be modeled.

Pure Light Esterel uses the presence or absence of signals at specific moments to describe a system. Valued Light Esterel is an additional feature to Light Esterel, which makes it possible to associate a signal with a value, a structure (e.g. matrix, array), or even a function call. For example, in the case where one wants to monitor a node's children's association and disassociation, the signals *node association* and *node disassociation* could be associated to the value of the address of the node being associated or disassociated, instead of having as many *node association* and *node disassociation* signals as there are children nodes. A signal could also be associated to a frame or a frame field. This additional feature can therefore allow modeling more complex properties more easily than with pure Light Esterel.

The observer module could also be used to assess if all properties are covered, to check that properties are verified during simulation. Indeed, it is possible that a property is modeled, but the conditions leading to its verification never happen in the tested scenarios. In that case, the observer not signaling that the property is violated does not mean that it is verified. Scenarios could also be generated to specifically target verifying these properties.

Bibliography

- [1] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann, “Systemc-link: Parallel systemc simulation using time-decoupled segments,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pp. 493–498, IEEE, 2016. 126
- [2] G. Delbergue, M. Burton, F. Konrad, B. Le Gal, and C. Jegou, “Qbox: an industrial solution for virtual platform simulation using qemu and systemc tlm-2.0,” in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016. 57, 126
- [3] F. Fummi, D. Quaglia, and F. Stefanni, “A systemc-based framework for modeling and simulation of networked embedded systems,” in *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pp. 49–54, IEEE, 2008. 31, 127
- [4] D. G. Padmavathi, M. Shanmugapriya, *et al.*, “A survey of attacks, security mechanisms and challenges in wireless sensor networks,” *arXiv preprint arXiv:0909.0576*, 2009. 127

Appendix A

Publications

Published:

1. Barnes, Calypso, Jean-Marie Cottin, François Verdier, and Alain Pegatoquet. "Towards the verification of industrial communication protocols through a simulation environment based on QEMU and systemC." In Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, pp. 207-214. ACM, 2016.
2. Barnes, Calypso, François Verdier, Alain Pegatoquet, Daniel Gaffé, and Jean-Marie Cottin. "Wireless sensor network protocol property validation through the system's simulation in a dedicated framework." In Signal Processing and Communication Systems (ICSPCS), 2016 10th International Conference on, pp. 1-9. IEEE, 2016.
3. Barnes, Calypso, François Verdier, Alain Pegatoquet, Daniel Gaffé, and Jean-Marie Cottin. "Validating a Wireless Protocol Implementation at Binary Level through Simulation Using High Level Description of Protocol Properties in Light Esterel." ICWMC 2016 (2016): 70.
4. Barnes, Calypso, Jean-Marie Cottin, François Verdier, Alain Pegatoquet, and Daniel Gaffé. "Developing a Framework Dedicated to Wireless Protocol Property Validation during Simulation." In Euromicro Conference on Digital System Design (DSD), pp. Work-in. 2016.
5. Affes, Hend, Amal Ben Ameer, Michel Auguin, François Verdier, and Calypso Barnes. "An ESL framework for low power architecture design space exploration." In Application-specific Systems, Architectures and Processors (ASAP), 2016 IEEE 27th International Conference on, pp. 227-228. IEEE, 2016.
6. Barnes, Calypso, Jean-Marie Cottin, Davide Quaglia, Enrico Fraccaroli, Alain Pegatoquet, François Verdier, and Stefano Angeleri. "Network-aware virtual platform for the verification of embedded software for communications." In Digital System Design (DSD), 2015 Euromicro Conference on, pp. 518-525. IEEE, 2015.
7. Barnes, Calypso, Jean-Marie Cottin, François Verdier, and Alain Pegatoquet. "Modeling a Node's System-On-Chip for a More Reliable Simulation of Wireless Sensor Networks." In Journée thématique Energy Aware Network, Labex UCN@ SOPHIA. 2014.

Submitted:

1. Barnes, Calypso, François Verdier, Alain Pegatoquet, Daniel Gaffé, Jean-Marie Cottin, and Alexis Arcaya-Jordan. "Using Virtual Platform Emulation to Validate WSN Protocol Implementations" In GDR SoC/SiP, 2017.

Appendix B

The OCARI protocol

OCARI (Optimization of Communication for Ad hoc Reliable Industrial network) is an energy efficient protocol that targets industrial wireless sensor networks. It is adapted to data gathering applications where a sink, called CPAN, collects data generated by sensor nodes. To reach the sink, data are routed according to a data gathering tree rooted at the sink.

The OCARI protocol stack is built from 4 layers as depicted in Figure B.1:

- The Physical Layer (PHY) which is the same as in the IEEE 802.15.4 specification [1], running on 2.4GHz ISM band
- The Medium Access Layer (MAC), called MaCARI, which initializes the network (synchronization and address assignment) and manages data message access to the medium in a free-collision way
- The Network Layer (NWK), called OPERA, which ensures the energy efficient data routing using EOLSR and the scheduling of node activities in time slots using OSER-ENA
- The Applicative Layer (APP) which handles the applicative measurement sampling and tunes the parametrization of the protocol stack

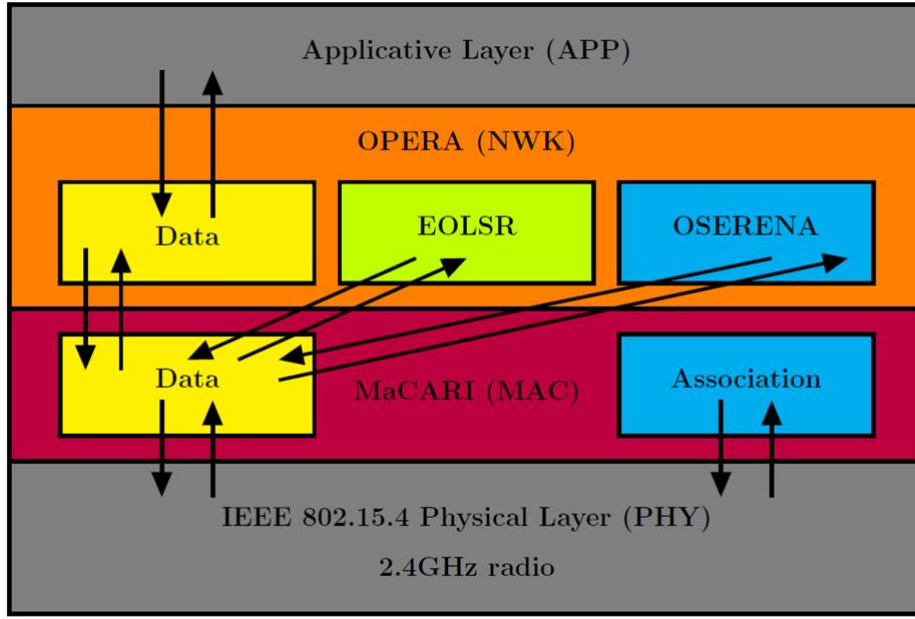


Figure B.1: The OCARI protocol stack.

B.1 The MAC layer

The medium access protocol following the global activity cycle depicted by Figure B.2.

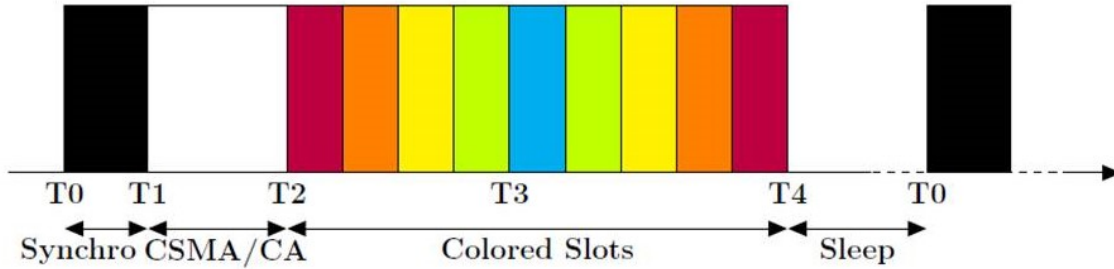


Figure B.2: The OCARI global cycle.

MaCARI is a synchronized MAC layer, which splits a global cycle, call the Superframe, in 5 periods: T0-T1, T1-T2, T2-T3, T3-T4 and T4-T0.

- Period T0-T1 is dedicated to the deterministic multihop synchronization of nodes using beacons in cascade.
- Period T1-T2 is reserved for competitive communication between nodes (including the CPAN) using CSMA/CA. This period is mostly used for control traffic (OPERA and MaCARI control messages). It can also be used for data traffic if the coloring is not yet available.
- Period T2-T3 is dedicated to upstream data traffic when the coloring done by OSERENA is available:

- Routing is done independently of the association tree by EOLSR inside OPERA Network layer.
 - Furthermore, when the coloring of nodes is done, a resulting scheduling of activity (also calculated by OSERENA) is then applied by MaCARI: any node will transmit in the slot corresponding to its color. It will sleep when none of its neighbors or itself was allocated to current time slot. OSERENA is a distributed graph coloring algorithm that ensures, for a stable network, a collision-free communication. In this phase, color assignment may be defaulted to the beacon indices in reverse order.
 - When changes of topology (loss of synchronization etc) are reported to the CPAN, a heuristic implemented in the CPAN decides to recolor or keep the existing colors (even if color conflict may happen).
- Period T3-T4 is dedicated to downstream data traffic when the coloring done by OSERENA is available.
 - Period T4-T0 is the MaCARI global sleeping period. All nodes of the network except the CPAN can switch to sleep mode for saving energy.

The OCARI network build-up is initialized by the MaCARI layer of the CPAN. The MaCARI build-up is composed of association, disassociation and address assignment mechanisms. In MaCARI, association of nodes is performed according to IEEE 802.15.4 specification. The CPAN is the first node of the network. After its reset, the CPAN launches an Energy Detection scan in order to pick the more free channel. Then it chooses an PAN Id, a short address (by default 0x0001), and starts sending beacon frames periodically on the chosen channel. Nodes joining this network by associating to the CPAN or to another already associated node. The process results in an acyclic oriented graph of association links, called the association tree. The CPAN is the root of this tree.

The address assignment mechanism is implemented in the CPAN. For each time a new node is associated to the network, the CPAN assigns a unique short address (16 bits).

In order to keep bandwidth usage to a minimum, a relayed association procedure was implemented.

The disassociation procedure depends on the beacon cascade functionality of MaCARI. During the beacon cascade, each received beacon's source is checked against the nodes children list. If the source is a child of the current node, the Missed Beacon field is reset. At each superframe, each of the children's Missed Beacon counter is increased. When the counter on any of the children reaches MACARI MAX MISSED BEACONS, the node is disassociated by removing the device from the Local Associated Devices table and by either sending a Disassociation Relay packet, or, in case of the CPAN, directly removing the node from the address table.

The disassociation mechanism in MaCari is also based on relayed packets. A node considers itself disassociated if it has sent a Disassociation Request packet or if it cannot find its address in the beacon payload coordinator list.

B.2 Format of the Macari frames

As can be seen in Figure B.3, the different fields of a MACARI frame are:

- Frame Control: this field contains information defining the frame type, addressing fields, and other control flags.

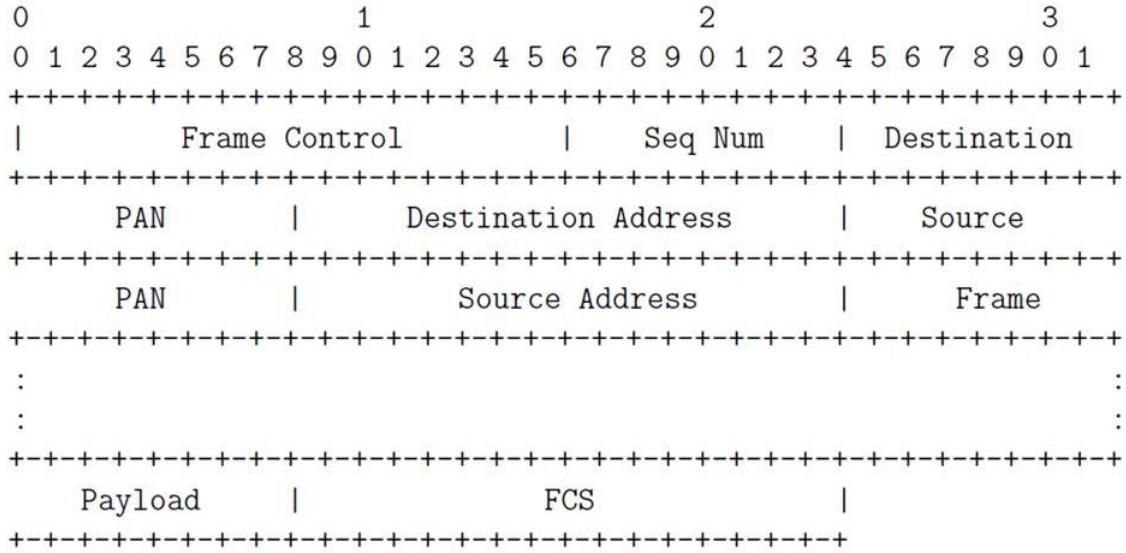


Figure B.3: General format of MACARI frames.

- Bits 0-2 = Frame Type: this field identifies the type of frame between the following types:
 - * (0x00) Beacon frame
 - * (0x01) Data frame
 - * (0x02) Acknowledgment
 - * (0x03) MAC Commands
- Bit 3 = Security Enabled: this field MUST be set to 0 because security aspects are not handled at the MAC layer in MaCARI.
- Bit 4 = Frame Pending: this field MUST be set to 0 because there is no asynchronous transmission in OCARI.
- Bit 5 = Acknowledgment Request: this field shall be set to 1 if the frame has to be acknowledged by the receiver or 0 otherwise. This field is ALWAYS set to 0 for Beacon frames.
- Bit 6 = PAN ID Compression: this field species whether the MAC frame is to be sent containing only one of the PAN identifier fields when both source and destination addresses are present. If this field is set to 1 and both the source and destination addresses are present, the frame shall contain only the Destination PAN Identifier field, and the Source PAN Identifier field shall be assumed equal to that of the destination. If this field is set to 0, then the PAN Identifier field shall be present if and only if the corresponding address is present.
- Bits 7-9 = Reserved (0x00).
- Bits 10-11 = Destination Addressing Mode: this field species whether the Destination Address field is represented in the frame according to the following:
 - * (0x00) No PAN identifier and address fields
 - * (0x02) Address field contains a Short Address of 16 bits
 - * (0x03) Address field contains a unique Long Address of 64 bits

- Bits 12-13 = Frame Version: this field species the version number corresponding to the MaCARI frame identifier (To Be Defined). Implementation: value 0x00
 - Bits 14-15 = Source Addressing Mode: this field species whether the Source Address field is represented in the frame according to the same values than Destination Addressing Mode.
- Seq Num: the Sequence Number field species the sequence identifier for the frame. For a beacon frame, the Sequence Number field shall specify a BSN. For a data, acknowledgment, or MAC command frame, the Sequence Number field shall specify a DSN that is used to match an acknowledgment frame to the data or MAC command frame.
- Destination PAN: this field, present only if the Destination Addressing Mode field is non-zero, species the unique PAN identifier of the intended recipient of the frame. A value of 0x in this field shall represent the broadcast PAN identifier, which shall be accepted as a valid PAN identifier by all devices currently listening to the channel.
- Destination Address: this field, present only if the Destination Addressing Mode field is non-zero, identifies the address of the intended recipient of the frame. A value of 0x in this field shall represent the broadcast short address, which shall be accepted as a valid address by all devices currently listening to the channel.
- Source PAN: this field, present only if the Source Addressing Mode field is non-zero and the PAN ID Compression field is equal to 0, species the unique PAN identifier of the originator of the frame.
- Source Address: this field, present only if the Source Addressing Mode field is non-zero, identifies the address of the originator of the frame.
- Frame Payload: this variable length field contains the payload of the frame.
- FCS: the Frame Check Sequence field contains a 16-bit ITU-T CRC.

B.3 The NWK layer: OPERA

the network layer of the OCARI stack is the main interface between the MAC layer (MaCARI) and the Applicative layer. This layer called OPERA (OPTimized Energy efficient Routing and node Activity scheduling for wireless sensor networks) consists of:

- Data Dispatcher: the dispatcher which deals with the end-to-end routing of data messages
- EOLSR (Energy efficient Optimized Link State Routing): the energy efficient routing protocol. As depicted in Figure B.4, EOLSR includes the neighborhood discovery and the gathering routes establishment.
- OSERENA (Optimized SchEduling of RoutEr Node Activity): the node activity scheduling based on node coloring. OSERENA assigns colors to nodes and each color is associated to a time slot where nodes having the associated color can transmit data. In OCARI, OSERENA is used to optimize energy consumption and data gathering delays. That is why, the tree rooted at the CPAN can be colored. When the data

transmissions are scheduled according to colors, the CPAN receives data from all sensor nodes in a single data gathering cycle. To support colored trees, some additional functionalities are added to EOLSR: EOLSR notifies OSERENA when the tree to be colored by OSERENA is stable.

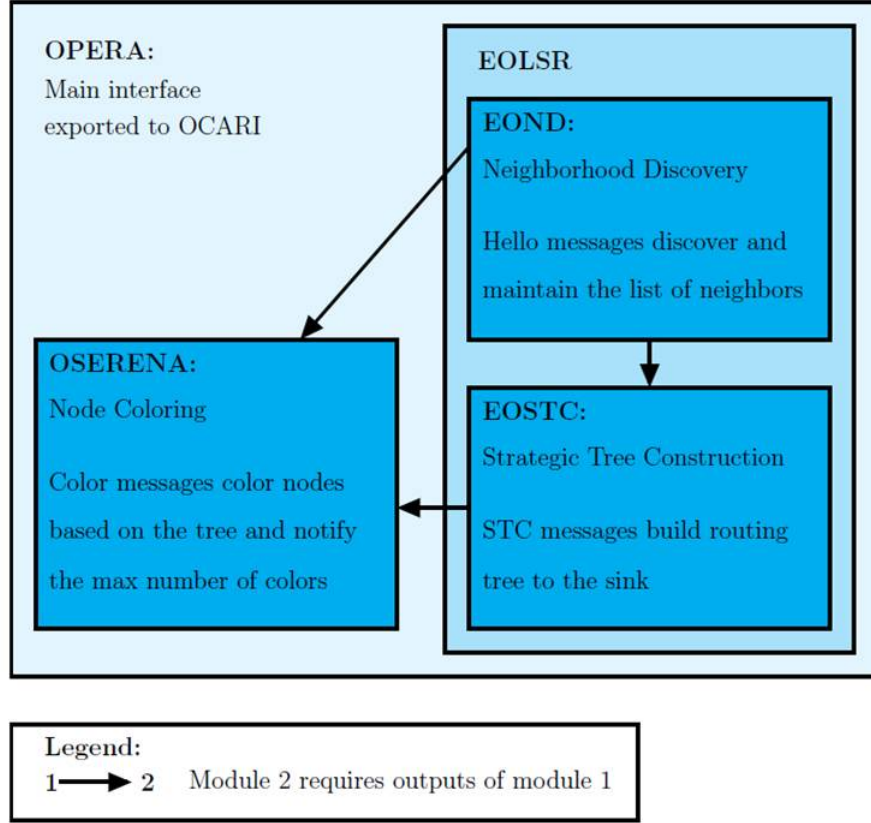


Figure B.4: The OPERA module.

B.3.1 EOLSR

EOLSR is an energy efficient extension of the routing protocol OLSR that is a link state based routing protocol. It is adapted to data gathering applications where sensors generate data and transmit them to a data sink called strategic node. EOLSR has two functionalities:

- EOLSR allows nodes to discover their neighborhood by exchanging Hello messages.
- EOLSR builds routes from any sensor node to the strategic node. Hence, data transmissions are done according to a tree rooted at this strategic node. To reach this strategic node, any sensor transmits its data to its parent in the tree. More than one strategic node may exist. EOLSR builds one routing tree per strategic node. Consequently, any sensor node has a parent per tree, that is per strategic node. Notice that among the strategic nodes, there is the CPAN. The tree rooted at the CPAN is the only tree that is colored by OSERENA in order to minimize the data gathering delays. When the data transmissions are scheduled according to colors, the CPAN receives data from all sensor nodes in a single data gathering cycle. To support this

coloring, EOLSR has a third functionality to detect when the tree to be colored is stable (i.e. no changes in the radio links of the tree). When the tree is stable, OSERENA can start the coloring.

As depicted in Figure B.4, EOLSR is composed of two modules:

- EOND: EOLsr Neighborhood Discovery: The Neighborhood Discovery module is in charge of discovering neighbors, detecting symmetric links and broken links by using the Hello messages.
- EOSTC: EOLsr Strategic Tree Construction: This module is responsible for building trees to strategic nodes through the use of STC messages. In addition, if any tree is designed to be colored by the OSERENA protocol, the EOSTC module should check the stability of this tree by means of TS messages.

EOLSR has the following properties:

1. Given that maintaining and exchanging a route toward each other network node is expensive in terms of data storage, bandwidth and energy, EOLSR maintains on each node only a route per strategic node.
2. The route selected is the route with the minimum energy cost that avoids routers with low residual energy.
3. To enhance scaling, any node does not need to store its 2-hop neighbors. Consequently, routing is no longer based on MPRs (MultiPoint Relays), unlike OLSR.
4. The format of the messages used is kept simple. Merging more than a message is not possible.
5. Compared to OLSR, EOLSR implements mechanisms that allow the CPAN to monitor the radio links stability of the tree if this tree is to be colored. Trees to be colored must be stable before being colored. For instance, the CPAN is alerted each time there is a new node in the network. To implement this functionality, EOLSR uses a new message called Tree Status (TS) message. The TS message is routed to the CPAN. If the links are stable, the root can trigger node coloring by OSERENA. Otherwise, the root will be notified of links instability.

In EOLSR like in OLSR, the neighborhood discovery is done via the exchange of the Hello messages. In EOLSR, the Hello messages are enriched with information relative to the energy level of the node. This information is used to compute the energy cost of routes. Indeed, in OLSR, the role of the Hello messages is:

- Detection of the appearance of new links and its signaling.
- Detection of disappearance of new links and its signaling.
- Check of the symmetry of links.
- Selection of the radio links of good quality.
- Discovery of the 1-hop neighbors.

The module Strategic Tree Construction is responsible for building a tree for every strategic node. More than one strategic node can exist in a single network. Each node maintains a topology table, called Strategic Tree Table (STT) containing the information received in the STC (Strategic Topology Construction) messages broadcast by the strategic nodes. Each entry in this table has a validity time. This table allows any node to have a route to any strategic node. EOSTC uses an energy cost to build the routing tree to the strategic node. This allows to save energy by selecting the minimum energy consuming path provided that router nodes on this path have sufficient residual energy.

B.3.1.1 Role of STC Messages

Routes advertisement and selection is done via the transmission of the messages STC (Strategic Tree Construction). Each strategic node periodically generates a message STC containing its address and a sequence number to allow each node to keep the most recent message.

Any node that receives this message can select a parent, that is the next hop to the strategic node if it does not have any or if the advertised route has a smaller cost. In this case, the node updates the route cost in the message by adding the cost of the transmission of the node itself. Then, it forwards it. Moreover, any node can change its parent if it receives a message with a lower cost. Like that, EOLSR adapts to topology changes.

B.3.1.2 Role of TS Messages

TS messages are only used for trees to be colored. Such a tree is uniquely identified by the pair (strategic address, tree sequence number):

- strategic address is the address of the strategic node. The sequence number of the tree
- tree seqnum is incremented each time the tree is recolored. The role of TS messages is twofold:
 - Determination of the number of descendants of any node in the tree: The number of descendants of any node is exchanged in the TS messages. Each node sums up the number of the descendants of its children and sends it to its parent in the message TS. Notice that the number of descendants is used as a priority in OSERENA. Indeed, each node has a priority that indicates its coloring order among the nodes that should not have the same color as itself. This priority is given by the number of descendants of the node in the tree. Ties are broken by the smallest address of the node: When any two nodes have the same number of descendants, the node with the highest priority is the node with the smallest address.
 - Detection of the stability of the tree: It is the responsibility of EOLSR to inform the strategic node about the stability of the tree using the messages TS. A tree is stable when all the subtrees of this tree are stable. The stability condition of any subtree, checked locally by the root node of this subtree is defined by:
 - * this node has a parent in the routing tree
 - * this node has not recorded a change (appearance or disappearance) of neighbors during a predetermined time interval

B.3.1.3 Periodicity of Hello Messages

There are two types of periodicities of the Hello messages:

- HELLO MAX: which is the period of sending Hello messages in a stable network. This period is equal to the time interval between two successive requests of transmission of a Hello by any node in the network. This is the maximum value and the default value to which the network tends to return.
- HELLO MIN: representing the minimum time interval that has elapsed since the last request of sending a Hello message by a node having detected topology changes. Notice that waiting at least a time HELLO MIN should allow nodes to group the topology changes detected in a single Hello message.

In conclusion, there are two cases:

- In steady periods, each node transmits a Hello each HELLO MAX period
- In case of neighborhood changes, each node that detects the failure of a link with a 1-hop neighbor or the appearance of a new link waits a maximum period of HELLO MIN before broadcasting its Hello message

B.3.1.4 Periodicity of STC Messages

Like for Hello messages, there are two periods for sending the STC messages:

- STC MAX which is the period of transmitting STC messages in a stable network. This period is equal to the time interval between two successive requests of transmission of a message STC by a strategic node. This is the maximum value and the default value to which the network tends to return.
- STC MIN represents the minimum time interval that has elapsed since the last request of sending a message STC initiated by the strategic node having detected a topology change.

B.3.2 OSERENA module

The coloring we propose is called OSERENA: Optimized SchEduling RoutEr Node Activity. OSERENA is implemented in the network layer of the OCARI stack, called OPERA.

The purpose of the coloring algorithm is to assign a color to each sensor node of the wireless network such that:

- Two conflicting nodes do not use the same color.
- The total number of colors used is minimized.

In wireless sensor networks, a typical communication paradigm is the data gathering. Hence, for OCARI, OSERENA is adapted to data gathering applications. In data gathering applications, sensors generate data and transmit them to a specific node called sink or strategic node. This strategic node uses a data gathering tree rooted at itself to collect data from sensors. This data gathering is defined by:

- Strategic Node Address: The address of the data sink which is root of the tree. This sink is called the strategic node. We suppose that this root is the CPAN.

- Tree Sequence Number: The sequence number of this tree.

The primary goal of OSERENA is to schedule nodes activities based on colors. Indeed, OSERENA is used on a slotted medium access based on a cycle composed of time slots. Each color is associated with a time slot.

- During this time slot, all nodes having the associated color can transmit.
- Any node is awake during its time slots to transmit data and during the slots of its neighbors to be able to receive data destined to it.
- The node can sleep (turn-off its radio) the remaining time.
- OSERENA provides MaCARI with the total number of colors (denoted max color in the remaining of this document). This information is necessary to MaCARI as it represents the total number of colored slots in the MAC cycle.

The scheduling of node activity offers the following benefits:

- Gain of bandwidth thanks to spatial reuse of RF: nodes having the same color may transmit simultaneously. Furthermore, no bandwidth is lost due to the retransmissions after collision.
- Gain of energy because a node may sleep in the time slot in which neither itself nor its 1-hop neighbors is transmitting. Also, no energy is lost in collisions and retransmissions after collisions.
- Gain in data gathering delays via a smart coloring adapted to data gathering applications (as explained in this document).

OSERENA is implemented in the network layer (NwCARI) of the OCARI stack. The colors that OSERENA determines are used by the MAC layer MaCARI to build the medium access cycle. Indeed:

- OSERENA provides MaCARI with the total number of colors (denoted max color in the remaining of this document). This information is necessary to MaCARI as it represents the total number of time slots of the MAC cycle.
- MaCARI associates each color with a time slot.
- On each node, OSERENA module provides MaCARI module with the color of the node itself and the colors of its neighbors. Hence, MaCARI on each node is aware about the time slot of the node itself and the time slots of its neighbors.
- Based on this knowledge, MaCARI schedules the activity of nodes: Each node is awake during its slots to transmit data and during the slots of its neighbors to receive data if any. Otherwise, the node turns off its radio to save energy.

Coloring Triggering OSERENA cannot start before the topology links becomes stable. It is the responsibility of EOLSR to notify the stability of the links to the CPAN:

- In each node, EOLSR monitors a local topology stability condition that is defined by: (1) having a parent in the routing tree and (2) absence of neighborhood changes (node disappearance or appearance) for a predefined time interval.

- EOLSR monitors the state of the tree using a message called Tree Status. Each EOLSR node having verified the local stability condition and received a Tree Status(stable) from all its children generates a Tree Status (stable) and transmits it to its parent.
- At the same time, OPERA copies the topology database from EOLSR to OSERENA. This database contains: the 1-hop neighbors, the parent and the children in the routing tree, the number of descendants of the node in the tree and the identification of the tree (address of the root and sequence number of the tree).
- When the CPAN receives a Tree Status(stable) from all its children, it deduces that the topology is stable and can trigger the coloring. Checking the stability of the topology before starting the coloring avoids recoloring in case of late arrival of nodes for instance.

A 3-hop node coloring is chosen. In a 3-hop node coloring, any node has as conflicting neighbors all its neighbors up to 3 hops. This coloring has two benefits:

- Any node can communicate with any 1-hop neighbor not only its parent in the data gathering tree without interferences.
- Any node can change its parent by any of its 1-hop neighbors without creating color conflicts.

To optimize data collection delays, each node should have a color greater than the color of its parent. Indeed, based on these colors, scheduling medium access according to the decreasing order of colors allows each node to aggregate the information received from its children before transmitting them to its parent. Consequently, information from all nodes can reach the sink in a single cycle.

To perform coloring, any node stores information about itself and about its conflicting neighbors. The information relative to its neighbors is collected via the exchange of the Color message.

OSERENA is a distributed 3-hop algorithm based on the exchange of the Color message. OSERENA proceeds by iterations or rounds. In one round, any node receives a Color message from each 1-hop neighbor, generates its own message and transmits it to its 1-hop neighbors.

Appendix C

AT86RF233 Radio state machines

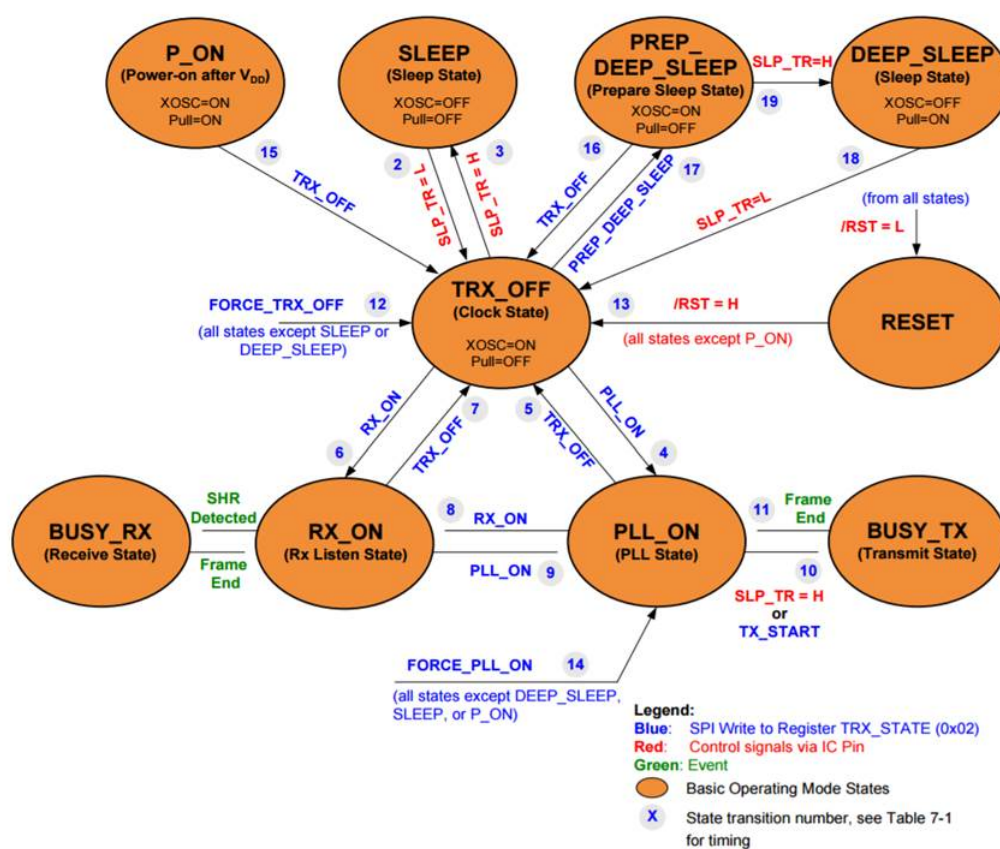


Figure C.1: Basic Operating Mode State Diagram of the AT86RF233 RF transceiver.

