POLITECNICO DI TORINO

Department of Electronics and Telecommunications
Master's Degree in Electronic Engineering (Embedded Systems)

Master's Degree Thesis

# Wireless sensor network for temperature monitoring and cooling optimization in data centers



Supervisor:
prof. Maurizio Rebaudengo

Tutor:
prof. José M. Moya

Candidate:
Andrea Giacobbe

Academic year 2019/2020

# Acknowledgments

There are a number of people without whom my master career and this thesis project would be harder and to whom I am greatly indebted.

First of all, to my family, that has been a source of encouragement and inspiration to me throughout my life, especially during the tough times. A special thank also to Oriana for the support and the help for writing this thesis.

Thanks to prof. Rebaudengo and the team GreenLSI (ETSIT UPM), in particular to prof. José Manuel Moya, Alejandro, and Alvaro for the contribution on this work. To my close friends Marco, Giovanni, Claudia, Lorenzo, Antonio and to all people that I have met during these years, in particular Manuel, Dimitri, Francesco, Cosimo, Marta, Nika, Emanuela, Manuela, Giada, and David.

To my mom and my grandparents who for sure would have been proud of the person who I am.

This work is for, and because of you and all the challenges to come.

# Summary

In recent years the increasing demand for storage and data processing led to the growing complexity and energy density of data centers. Several measures to improve performance and energy efficiency are being studied, not only to reduce operational costs, but also to facilitate a sustainable industry growth. To accomplish this aim, the cooling aspect has a relevant role within the data center infrastructure.
The objective of this thesis has been the development of a sensor node model for the wireless network responsible for monitoring the temperature and the humidity in air-cooled data centers. This network is made with Bluetooth 5.0 with Mesh profile, which allows the interconnection of a multitude of devices. The network consists of a large number of distributed sensor nodes at strategic positions for data sensing, coordinator nodes to control them and let data flow to databases for collection, and then data analysis. These sensor nodes send regular measurements to the gateways that send these data to the analysis platform, which can be used to control the cooling resources in order to dynamically optimize the set-points of the cooling systems in the data centers. The solution of this wireless sensor network takes also in consideration the easy configuration and maintainability by the data center operator to deploy the sensor nodes.
The choice to exploit wireless sensors is due to the fact that they have different advantages, such as not requiring to make changes to the existing infrastructure, being easy to deploy and relocate, and being relatively low cost.
As we are working with critical infrastructures, and cooling control is one of the most important operation decisions, the primary objective is to guarantee safety in operation. For that reason, formal methods and real-time analysis techniques are applied to the design and implementation in order to validate the behaviour of this node in the network.

The initial phase of this thesis project was the definition of the requirements and the system design as finite state machine. The simulation and verification phases have assured the correctness of the system over the set of defined specifications and safety properties due to its real-time purpose even through a complex state space. In order to achieve this, the model checker tool *NuSMV* has been chosen among others available on the web since it is well documented and because of it supports specifications expressed as Linear Temporal Logic formulas.

A C based implementation of this sensor node model has been proposed also thanks to the team *GreenLSI* of Polytechnic University of Madrid. The *Nordic Semiconductor nRF52840* board, which hosts a 32-bit *ARM* Cortex-M4 processor, has been chosen for the purpose because it has protocol support for Bluetooth mesh network that is suited to create large-scale device networks.

Finally, with the results of the C implementation it has been proven the real behaviour of the node through the timing and power analysis. Two possible scheduling for the models have been proposed to guarantee the deadlines by the node, one capable to reduce the dynamic power consumption of this node.
In particular, the topics are divided in this way:

- In **Chapter 1**, an introduction to data centers and motivation.
- In **Chapter 2**, the system modelling using Finite State Machines.
- In **Chapter 3**, model checking and Linear Temporal Logic.
- In **Chapter 4**, the NuSMV checking tool to verify the model.
- In **Chapter 5**, the system requirements and the wireless sensor network definition.
- In **Chapter 6**, the simulation and the verification of the SMV model to control the node.
- In **Chapter 7**, the language containment between the C implemented model and the SMV model.
- In **Chapter 8**, the scheduling techniques and the estimation of total power consumption for the proposed implementation of the node.
- In **Chapter 9**, the conclusion and some possible future developments of this thesis project are suggested.
- In **Appendix A**, the SMV code is shown.

- In **Appendix B**, the C implementation for the node based on the board *Nordic Semiconductor nRF52840*.
- In **Appendix C**, the calculation for the daily total power consumption of the sensor node.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Data centers

## 1.1 Introduction

The trend for Cloud computing is leading to the increasing of data centers due to the necessity to store a big amount of information, which is rising over time. Companies such as *Amazon*, *Google*, $Microsoft$, $Facebook$ and *Apple* have decided to offer information stored in the Internet Cloud, ensuring faster and more efficient services to the users. The magic of the virtual world is created by computer servers, which store website data and share it with other computers and mobile devices, but every search, click, or streamed video can activate servers of different data centers around the world, consuming a lot of real energy resources. The advantages of Cloud computing allow the usage of a technological infrastructure with high degrees of automation, consolidation and virtualization, which means a more efficient management of the resources of a data center. The Cloud model allows, besides a large number of users, the use of concurrent applications that otherwise would require a dedicated computing platform. The total Cloud market had a value of $214.3 billion in 2019, and it is projected to grow up to $331.2 billion in 2022, according to *Gartner*[2].

## 1.2 Energy consumption and environment sustainability

Today, data centers consume about 2% of electricity worldwide, that could rise to 8% of the global total by 2030, according to a study by Anders Andrae, who researches sustainable ICT at $Huawei\ Technologies$[3]. The servers, the storage equipment, and the power cooling infrastructure in the data centers need electricity in order to operate. The total energy consumed by data centers is growing sharply, therefore

becoming a critical element for economic and environment sustainability. The main contributors to the energy consumption in a data center are:

- The Information Technology (IT) resources, which consist of servers and other IT equipment (up to 60% of the overall data center consumption);
- The cooling infrastructure needed to ensure that IT operates within a safe range of temperatures, ensuring reliability (around 40%)[17];
- The remaining percentage comes from lighting, generators, Uninterrupted Power Supply (UPS) systems and Power Distribution Units (PDUs)[6].

The big demand for connectivity means more energy into these data centers, and much of that energy is non-renewable and contributes to carbon emissions. Data centers contribute 0.3% to global carbon emissions, according to *Nature*[4].
Just the United States has 3 million of data centers, and a large number is clustered in Virginia. Only 12% of *Amazon*'s data centers and 4% of *Google*'s are powered by renewable energy in that area, according to *Greenpeace*[3].
In China, data centers get 73% of their power from coal and 23% from renewable sources[3]. As coal-powered sources are relatively cheap and abundant, there is a lack of clean energy infrastructure which is still under development. China's data centers emitted 99 million tons of $CO_2$ in 2018 and will emit two-thirds more by 2023 unless the industry addresses its energy consumption, as stated in a study by *Greenpeace* and *North China Electric Power University* of 2019[3].
The power usage into a server can be: *Dynamic* or *Static*. Dynamic power contribution depends on the switching transistors in electronic devices during workload execution. Static power consumption, around 70% of the power consumption in a server, is strongly related to the temperature due to the leakage currents which increase as technology scales down[5]. The impact of leakage currents on IT resources increase with higher temperatures which can be controlled by a precise management of the cooling system. The temperature in data centers is strongly increasing due to the activity of servers to guarantee the growing demand. The generated heat is evacuated outside as thermal pollution avoiding server failures. With better cooling, the heat coming off the servers can be used to warm other places thus saving electricity demand elsewhere[4].
These energy and thermal aspects stimulate researchers, not only to develop more performing data centers, as well as to minimize their environment impacts.

## 1.3 Motivation

To keep the temperature low is a popular solution to locate data centers in cool environments and blow the outside air into them[4]. For high-density and high-power computing, the most efficient thing to do is to immerse servers in a non-conductive oil or mineral bath[4], thus utilizing big amounts of water (billion litres). Among all available cooling solutions, air cooling dominates the data center industry due to its simpleness. However, the efficiency of air cooling solution is not the best due to the low air density and specific heat.
Controlling the set point temperature of cooling systems in data centers is still to be defined and it represents a key challenge from the energy perspective. Often this choose is conservative as provided by manufacturers of equipment, based on the worst-case scenario resulting on over-cooled facilities. Temperature increasing by 1°C results in savings of 7.08% in cooling consumption[5], so an accurate management ensures both safe temperature range for IT resources and energy saving. Moreover, the accumulated heat can lead to server shutdown and to decrease the lifetime. In case of air cooling system, ASHRAE (American Society of Heating, Refrigerating and Air-Conditioning Engineers) recommends temperatures within 18°C-27°C, and relative humidity within 27%-60%, at constant pressure, as shown in Figure 1.1.

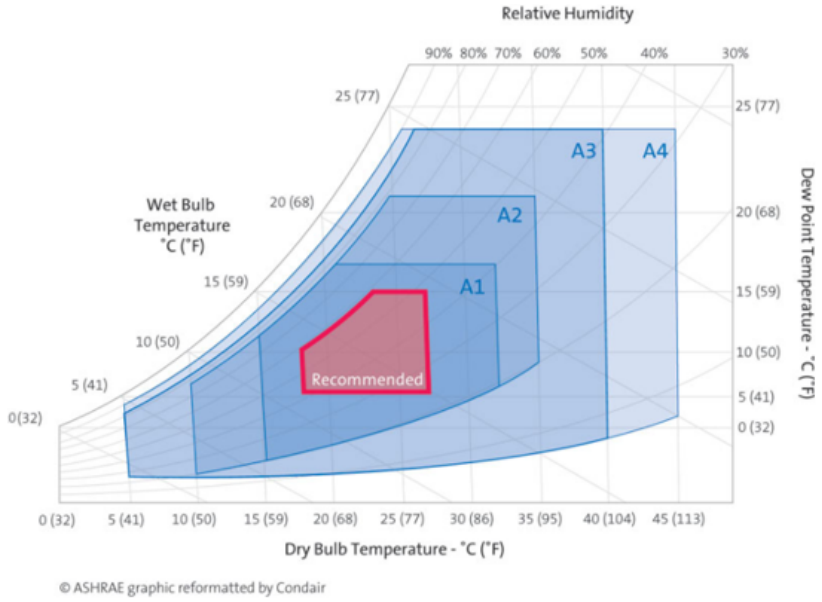



Figure 1.1: ASHRAE's graph which specifies the strict air condition range to ensure the high reliability and performance of IT equipments in data centers[8].

In addition, high air temperature reduces the cooling capability, in particular a study states that an increment of 15°C leads to an increment in circuit delay by 10-15%.

There are two common approaches to evaluate the thermal performance[7]:

- Computational Fluid Dynamics (CFD) simulation, useful for the design and the trouble-shooting phases.
- Real-time measurement, better for the stationary thermal behaviour, server replacement, equipment upgrading and time-varying outdoor condition.

Although a model validation procedure for the the real-time system measurement is still necessary[7], this measurement is more precise and reliable than simulation. These monitored data can be used to dynamically control the cooling resources as response to temperature changes. Moreover, for the real-time measurement, big data analytic techniques and advanced data processing architectures should be employed to manage these data.
Measurement strategies inside the data center can be mainly categorized into three types[7]:

- Traditional method with limited number of sensors installed on fixed locations, openings and hot/cold aisles, etc.
- Advanced method using increased number of sensors (perhaps with some mobility) measuring a wide range of locations.
- 3D thermal map created by built-in sensor readings from various components inside the server, e.g., CPU, HDD, memory, etc.

The advanced method can achieve a better space resolution in comparison with the traditional method. The 3D thermal map provides a big thermal picture at the data center scale without extra cost, but it is often unavailable where servers belong to different owners, thus to make difficult server cooperations. In this case, the advanced method is the best option, and also because of the readings from built-in sensors may suffer from relevant accuracy issues which often lead to overcooling.
Another relevant aspect is the location to deploy the sensors, in particular, they should be in CRAH, hot/cold aisles, and underfloor plenum (see Figure 1.2)[7]. High-density sensor systems are employed to monitor and control the cooling system. For example, the sensor density of HP data center in Bangalore, India, is 1.15 sensors/$m^3$[7]. Therefore, up to 10% of IT equipments is replaced each month and it has been observed that most of the data center operators deploys the sensors in wrong places[7]. In literature, the monitoring intervals range from 0.1 seconds to 30 minutes[7]. Monitoring systems can be wired, wireless, or a combination of both. IoT technology for industrial purpose can be applied to wireless solutions to improve scalability and system flexibility.
Some issues are related to wireless systems such as[7]: limited communication range,

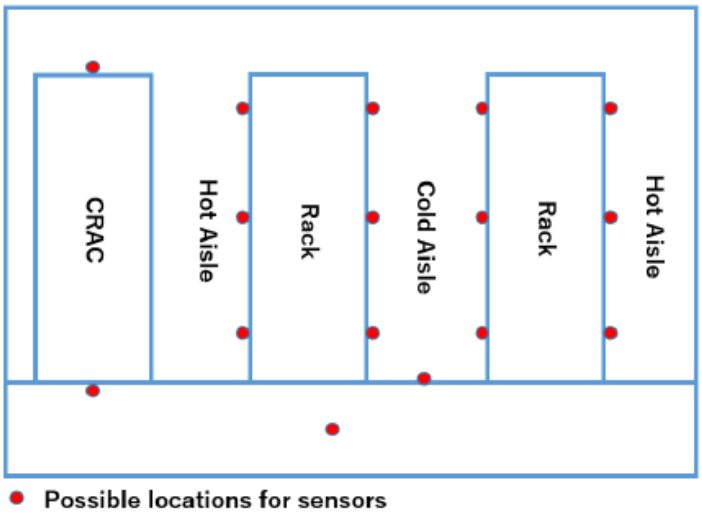



Figure 1.2: Proposal locations for sensors in the data centers: CRAC inlet, CRAC outlet, underfloor plenum, rack inlets and rack outlets[7].

due to obstacles which block the wireless signals; wireless sensing nodes are often powered by battery, hence sleep scheduling techniques are necessary to save energy and to extend system lifetime; node failures and security issues can be addressed by using periodical probing message and encryption techniques.
In Summary, exploring innovative and efficient cooling solutions will become more important in coming years to save electricity costs, guarantee performance connectivity for the user and lessen carbon emission and water utilization.

## 1.4 Objectives

- Analysing the system requirements for this wireless sensor network.
- Defining FSM in order to control the sensor node within the network.
- Coding the model within the NuSMV environment.
- Verifying the model with the model checking approach.
- Analysing the power consumption of the node within the network and the scheduling approaches based on a first implementation.

# Chapter 2

# Finite-State Machines

An introduction to Finite-State machines is shown in this chapter[1], in particular a deep view of the Extended FSM. Real-time embedded systems must be sensitive to environment signals and react to them in a certain amount of time. Hardware and software design methods must be integrated to achieve functional and non-functional requirements. A way to specify real-time systems and their behaviour is through *Finite-State Machines* (FSMs).

## 2.1 Basic elements of Finite-State Machine

During software design and sequential logic circuits, the FSM represents the abstract computation model. A FSM is a system that collects the *state* of an object at a given time and operates on input stimuli (*events*) to change the state and/or cause an action or an output to occur for any given change[9]. Each state is related to the object's history. A change from one state to another is called a *transition*, due to the occurrence of some *triggering events* or *conditions*. A FSM can be defined with a finite number of states. The state in which it is at a given time is called the *current state*. A FSM can be represented by a directed graph called *state diagram* where each state is a node and each transition is an edge; or through a *state transition table*. States can be classified into *initial states*, *final states* and *intermediate states*. Generally, an initial state is graphically identified with an incoming arrow, while a final state is represented with two circles. It describes the system's behaviour considering discrete events and states, where transitions fired on certain events, not specifying the exact time they occur but just knowing the temporal order between them. The complexity of the traditional FSM model is $2^N$, where $N$ is the number of the variables that are used to describe the model. *Deterministic*, *Nondeterministic*, and *Extended* FSM are some of the FSM types.

[1]The information in this chapter comes from [9], [10], [11], [12].

## 2.2 Deterministic Finite State Machine

In Deterministic FSM, the transition from one state to another and the outputs are defined uniquely for each triggering event presented at the input of the machine[9]. Mathematically, it can be denoted as $M = (S, I, \phi, s_0, F)$[9], where:

- $S$ is a finite set of states.
- $I$ is a finite set of possible input symbols.
- $\phi$ is a transition function from each pair (state, input) to a new state, $\phi : S \times I \rightarrow S$
- $s_0$ is the initial state, $s_0 \in S$.
- $F$ is a set of final states, $F \subseteq S$.

Given an input to the machine, it will update its state and, possibly, produce an output to each transition. There are two types of deterministic FSMs that generate outputs:

- *Moore* machines, whose output depends only on the current state.
- *Mealy* machines, whose output depends on the current state and the current input.

Although often a Mealy machine is more compact and practical than a Moore machine since it can perform the same task with a less number of states, any Moore machine can be turned into Mealy, and vice versa.

### 2.2.1 Moore Machine

A Moore machine generates the output as it enters into a new state (the reaction to input events in one cycle is always produced in the next cycle[12]). It is denoted by $M = (S, I, \phi, s_0, O, \lambda)$. The final states set $F$ is not included in the Moore machine definition. In addition, a finite set of output symbols $O$ and the output function $\lambda$ from each state to an output symbol ($\lambda : S \rightarrow O$), indicating that outputs depend just on states, are considered. Figure 2.1 shows a basic state diagram of a Moore machine with two states. The machine starts in state $S_0$ and it produces $X$ to the output. When it receives the input $\alpha$, the machine shifts to $S_1$ and produces $Y$. Similarly, when the machine is in $S_1$, an event $\beta$ changes the machine's state to $S_0$. Again, the machine is in $S_0$ and generates $X$ as output.
Figure 2.2 illustrates how next states and outputs are computed in a Moore machine.

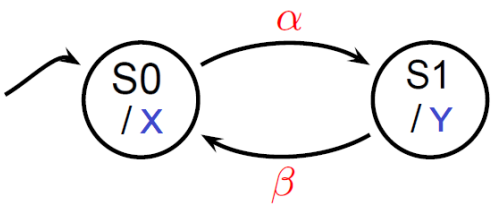



Figure 2.1: State diagram of a simple Moore machine.

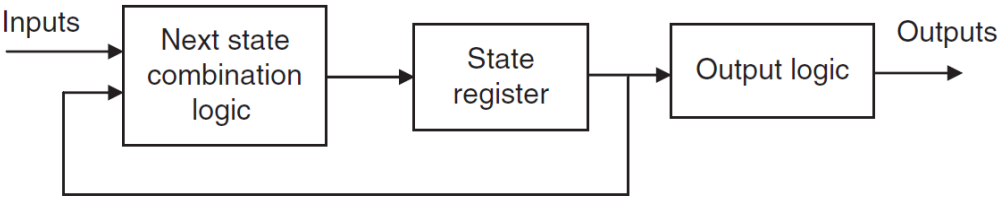



Figure 2.2: State and output generation in Moore machine[9].

### 2.2.2 Mealy Machine

In Mealy machine, the outputs are generated when certain transitions occur, instead of states as in Moore machine. In a Mealy state diagram, transitions are labelled in the format of $i/o$, where $i$ is the input and $o$ is the output generated. It is denoted by $M = (S, I, \phi, s_0, O, \lambda)$ as Moore machine with the only difference that the output function is $\lambda : S \times I \to O$, thus the outputs depend on both current states and inputs, and they are represented on the transitions[9], see Figure 2.3. In the Mealy model, an input produces a change of output in the same cycle[12]. Figure 2.4 illustrates how the following states and outputs are produced in Mealy machines.

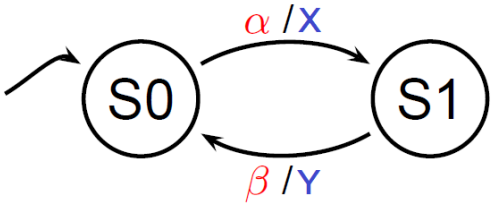



Figure 2.3: State diagram of a simple Mealy machine.

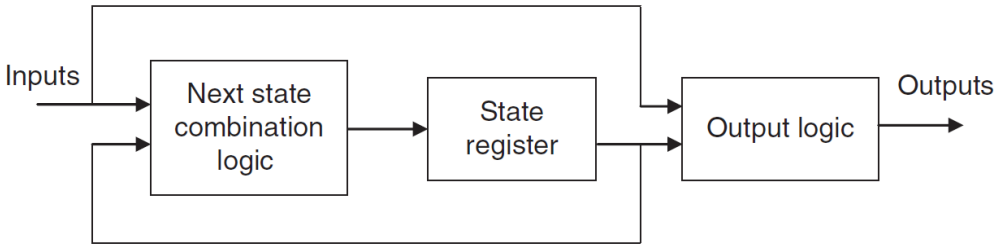



Figure 2.4: State and output generation in Mealy machine[9].

## 2.3 Non-Deterministic Finite State Machine

In a Non-deterministic FSM, given an input and a state, there may be more than one next state, or a transition can link states without any input, or there is no next state at all for some given input. It can be used when there is an unspecified system behaviour. Mathematically, as it can be identified $M = (S, I, \phi, s_0, F)$[10], where $S$, $I$, $s_0$, and $F$ are defined as in deterministic FSM, but with the difference that the transition function is defined as $\phi : S \times I \rightarrow S^*$, where $S^*$ denotes the power set of $S$, which defines the set of all possible subsets of $S$. This kind of model defines different possible behaviours for the machine. Figure 2.5 shows a simple Non-deterministic FSM, where $S_0$, $S_1$, $S_2$ are the states, and $\alpha$, $\beta$, and $\gamma$ are the inputs.

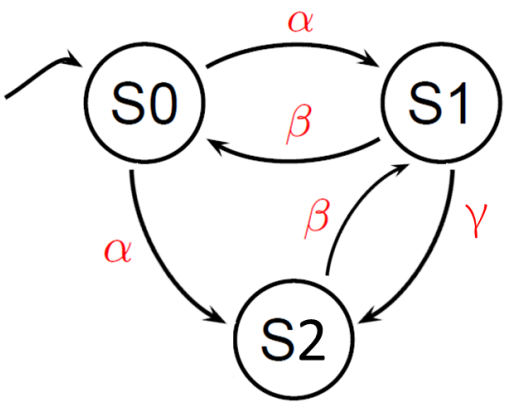



Figure 2.5: A simple Non-deterministic FSM.

## 2.4 Extended Finite State Machine

In a traditional FSM, the transition is associated with a set of output Boolean functions given a set of input Boolean conditions. In an *Extended* FSM model (EFSM), the transition can be expressed by an $If$-*statement* consisting of a trigger condition and a set of data operations: when the trigger condition is satisfied, the transition is executed, thus changing the machine's state and performing the specified data operations[11] (outputs and variables are updated and available in the next cycle). Considering the example shown in Figure 2.6[11], the transition from state $S_1$ to $S_0$ is denoted as:

$$T(S_1 \rightarrow S_0) \ : \ If(counter \ != 6) \ counter++;$$

This transition is only executed when the current state is $S_1$ and the data variable *counter* is not 6. When it is executed, it brings the machine to state $S_0$, and increases the counter value by 1 through the data operation $counter++$. Generally, the trigger conditions and the data operations may depend on the external inputs and data variables.

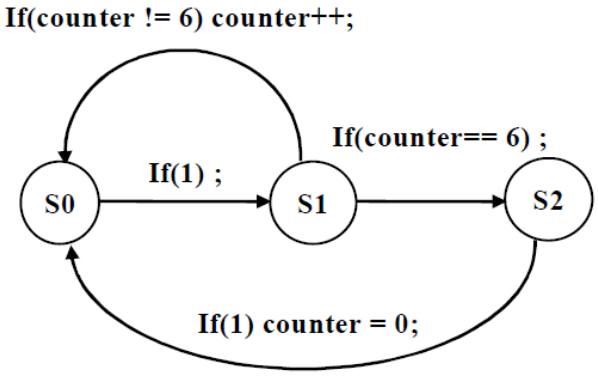



Figure 2.6: An example of Extended FSM[11].

### 2.4.1 Primitive Architecture

To synthesize an EFSM, it can be turned into a structural diagram consisting of three major combinational blocks, see Figure 2.7, and a few registers[11]. These blocks are:

- *FSM-block*, which is a traditional FSM that performs the state transition graphs of the EFSM model;

- *A-block*, the arithmetic block to execute the data variable due to each transition. This block is regulated by the output signal of the *FSM-block*;
- *E-block*, it evaluates the trigger conditions for each transition. The data variables go into this block, while it generates the binary output signals given as inputs to the *FSM-block* for deciding the state transition.

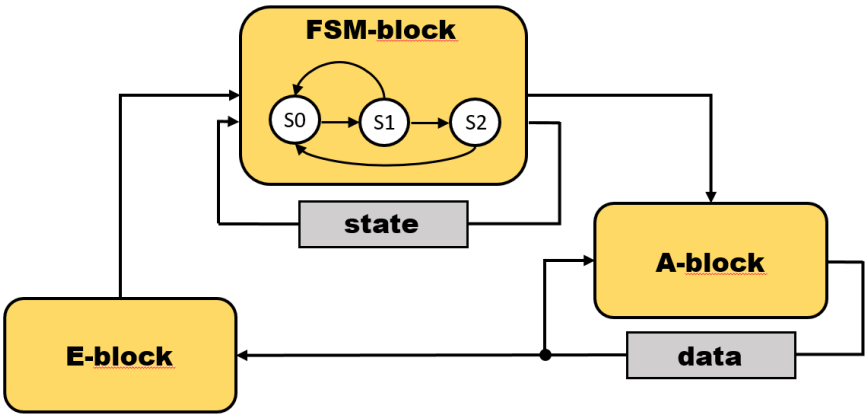



Figure 2.7: A basic architecture for realizing an EFSM[11].

For the example in Figure 2.6, *FSM-block* is the logic for the FSM, *A-block* is responsible for performing three operations on the data variable *counter* (increment, reset, and freeze), and *E-block* realizes the trigger conditions of transitions.

### Timing critical path

The timing critical paths can be revealed by examining the most complicated transition in the EFSM model. For the implementation, the transition that involves the logic in every one of those three blocks is the most timing critical path[11]. Since the transition needs to be executed in a single clock cycle if triggered, its corresponding timing path could start from the output of the *data variable block*, passing through the *E-block*, *FSM-block*, *A-block*, and finally reaches the input of the *data variable block*. In Figure 2.6, the most timing critical path is the transition $T(S_1 \rightarrow S_0)$[11], since it activates all the three blocks. If the delays of the main three blocks along the critical path are denoted as $d(FSM\text{-}block)$, $d(A\text{-}block)$, and $d(E\text{-}block)$, then the clock cycle time is dominated by $\tau_{cp} = d(FSM\text{-}block)+d(A\text{-}block)+d(E\text{-}block)$[11].

# Chapter 3

# Model Checking

In this introductory Chapter we will show how the behaviour of the embedded systems can be verified in a formal way specifying the properties with temporal logic, in particular *Linear Temporal Logic* (LTL). Model checking is a verification technique for the first step of system design, to find out whether system model matches the specifications, or not.

## 3.1 Introduction to Model Checking

*Model Checking*[1] means verifying formally and automatically finite-state concurrent and reactive systems. It comes from the work by E. M. Clarke, E. A. Emerson, by J. P. Queille, and J. Sifakis in the early 1980s. Functional and property specifications are important to ensure reliability and correctness of digital circuits and software designs, in particular for those are highly critical. Two common approaches to ensure them are *software testing* and *simulation*. In software testing, the component, hardware or software, is executed to evaluate desired properties. Instead, in simulation, the system is modelled with a set of mathematical formulas: a program shows an operation to the user through simulation without actually performing that operation. In spite of those techniques being widely used in industrial applications, unfortunately, they are not able to simulate or test all the possible scenarios of a given system (due to the high number of possible cases to be taken into account), thus the failure cases could not be detected, generating potential damages in case of errors in the real environment. For this case, systems are modelled as FSMs and desired properties are specified with formulas based on temporal logic. A formal verification method can be stated: given a System model $M$ and a property specification, expressed as a temporal logic formula $\varphi$, a tool, called *Model Checker* decides if $\varphi$ is satisfied by $M$ from a given state $s$[9]. This checker prints out '$Yes$'

[1]The information in this Chapter come from [9], [13].

whenever the property is satisfied; otherwise, it shows a *counterexample* of execution in which the property is violated. Figure 3.1 illustrates how the checker verifies the property on the system model, that is sensitive to the external environment.

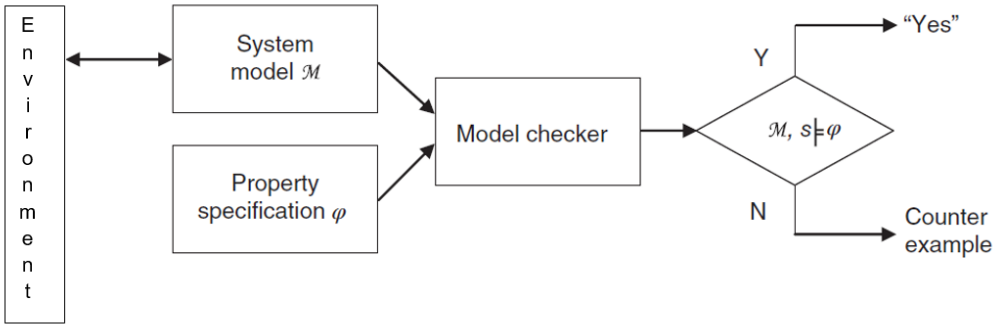



Figure 3.1: Model checking[9].

## 3.2 Temporal Logics

Temporal logic is a set of symbolism and rules which describes the ordering of events in time without explicit notion of time. A temporal logic model contains states that correspond to different time events. There are two models of time:

- *Linear-time model*: Introduced by Pneuli in the 1970s, it defines the time as a single path of time points which are ordered linearly, where each time point has a unique successor and it is easy to say which one is earlier than the other, in a deterministic way. As it is illustrated in Figure 3.2, point $t$ is the $future$ of point $s$. In this case, the logic to model time as sequence of states is called *Linear Temporal Logic* ($LTL$).

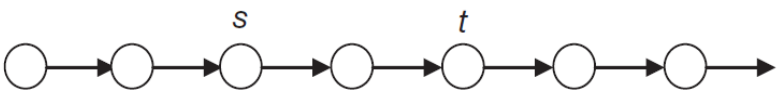



Figure 3.2: LTL model[9].

- *Branch-time model*: a time point may have two or more future points that are not related to each other. For any two of those future points, we cannot say which one occurs before the other point in time. This means that, for each time point, its future is not deterministic. For example, in Figure 3.3 it is not possible to say that $s$ is the future of $t$ (or vice versa), or what is the future

of $r$ deterministically. The other logic to model time as tree-like structure is called *Computational Time Logic* ($CTL$).

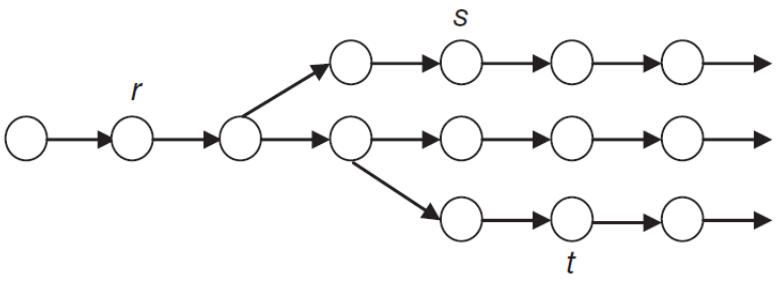



Figure 3.3: Branching-time model[9].

From now on it is considered just LTL model.

## 3.3 Linear Temporal Logic

LTL models time as an infinite sequence of states, called *computation path*, or just *path*. Different paths are present for representing different possible futures. The syntax and the semantics of LTL will be shown onwards.

### 3.3.1 Syntax of LTL

A LTL formula, over a finite set $AP$ of atomic propositions (indivisible formulas), is defined as the following *Backus-Naur* $form$[9]:

$$\varphi ::= \top|\bot|a|\neg\varphi|(\varphi \wedge \psi)|(\varphi \vee \psi)|(\varphi \rightarrow \psi)|(\bigcirc\varphi)|(\Diamond\varphi)|(\Box\varphi)|(\varphi \boldsymbol{U} \psi)|(\varphi \boldsymbol{R} \psi) \quad (3.1)$$

where $a \in AP$; $\top$ and $\bot$ stands for $true$ and $false$, respectively. An atomic proposition is true on a path, if it holds on the first state of a given path. Any atomic proposition, logic constants $true/false$, and any of the operators applied to any LTL formula are still a LTL formula, as shown in Equation 3.1. Any LTL formula is based on the following propositional boolean operators (represented in Table 3.1):

- $\neg$ : Negation ($not$).
- $\wedge$ : Conjunction ($and$).
- $\vee$ : Disjunction ($or$).
- $\rightarrow$ : Implication ($If\text{-}then$).

| $\varphi$ | $\psi$ | $\neg\varphi$ | $\varphi \wedge \psi$ | $\varphi \vee \psi$ | $\varphi \rightarrow \psi$ |
|---|---|---|---|---|---|
| T | T | F | T | T | T |
| T | F | F | F | T | F |
| F | T | T | F | T | T |
| F | F | T | F | F | T |

Table 3.1: Truth table of propositional boolean operators.

In addition, there are operators which express relation among states[9]:

- $\bigcirc$ (***X*** or *Next*) specifies that a formula is held at the next time point.

  $\bigcirc \varphi$ is true if $\varphi$ is true at the next step.

- $\Diamond$ (***F*** or *Future*) means that a formula is eventually held at some point in the future.

  $\Diamond \varphi$ is true if $\varphi$ is true somewhere in the future.

- $\Box$ (***G*** or *Globally*) means that a formula is held on the entire subsequent path.

  $\Box \varphi$ is true if $\varphi$ is always true in all future states.

- ***U*** (*Until*) defines that a formula is held until another one occurs.

  $\varphi \boldsymbol{U} \psi$ is true if $\varphi$ is true at least until $\psi$ becomes true.

- ***R*** (*Release*) defines that a formula is held until and including the time point where another formula becomes true.

  $\varphi \boldsymbol{R} \psi$ is true if $\psi$ is true until the first position in which $\varphi$ is true.

All these operators have different priorities: $\neg$, $\bigcirc$, $\Diamond$, $\Box$ (*unary* operators) are stronger than others (binary operators), and ***U*** takes precedence over $\wedge$, $\vee$, $\rightarrow$. The listed propositions, along with others not presented here, can be combined to express more complex ones.

### 3.3.2 Semantics of LTL

Each LTL formula describes the evolution of events in time. The evaluation of LTL propositions is related to paths, which means a path can satisfy a LTL formula, or not. Assume a fixed set $AP$ of atomic propositions, a set of states $S$, and a *labelling function* $L$ which maps $S$ to the power set of $AP$, denoted by $2^{AP}$, which represents the set of all subsets of $AP$. If $AP = \{p,q,r\}$, then:

$$2^{AP} = \{\varnothing,\ p,\ q,\ r,\ \{p,q\},\ \{q,r\},\ \{p,r\},\ \{p,q,r\}\}$$

For each state $s \in S$, $L(s)$ is a set of all atomic propositions that are evaluated to be true in that state[9]. A generic path can be denoted as:

$$\pi\ =\ s_1\ \rightarrow\ s_2\ \rightarrow\ s_3\ \rightarrow\ ...$$

For any LTL formula, the binary satisfaction relation with respect to a generic path $\pi$ is denoted by '$\models$', in this form:

$$\pi \models LTL\_formula$$

which means that the path $\pi$ satisfies $LTL\_formula$, according LTL syntax. The LTL formulas satisfied by a path $\pi$ are[9]:

- **True/False**, both are always satisfied by the path.

$$\pi \models \top \tag{3.2}$$
$$\pi \models \bot \tag{3.3}$$

- **Atomic proposition**, any atomic proposition $a \in AP$ is true on a path, if and only if (iff) it holds on the first state $s_1$ of the path, so it holds for all that path.

$$\pi \models a\ \textit{iff}\ a \in L(s) \tag{3.4}$$

- **Negation of atomic proposition**, the negation of an atomic proposition $a \in AP$ holds on a path $\pi$ if and only if the atomic proposition does not hold in $s_1$ (so for all states of the path).

$$\pi \models \neg\ a\ \textit{iff}\ a \notin L(s) \tag{3.5}$$

- **Composition**, the composition of LTL formulas $\varphi$ and $\psi$ with propositional boolean operators ($\wedge$, $\vee$, and $\rightarrow$) is evaluated in the first state $s_1$ of the path.

$$\pi \models \quad \varphi \wedge \psi \quad \textit{iff}\ \ \pi \models \varphi\ \textit{and}\ \pi \models \psi \tag{3.6}$$
$$\pi \models \quad \varphi \vee \psi \quad \textit{iff}\ \ \pi \models \varphi\ \textit{or}\ \ \ \pi \models \psi \tag{3.7}$$
$$\pi \models \quad \varphi \rightarrow \psi \quad \textit{iff}\ \ \pi \models \psi\ \textit{as long as}\ \pi \models \varphi \tag{3.8}$$

- **Next** ($\bigcirc$), the LTL formula is evaluated to be true in the next state $s_2$.

$$\pi \models \bigcirc \varphi \textit{ iff } \pi[s_2...] \models \varphi \tag{3.9}$$

- **Future** ($\Diamond$), means that the formula will become true at some state $s_i$ along the path.

$$\pi \models \Diamond \varphi \textit{ iff } \exists\ i \geq 1 \mid \pi[s_i...] \models \varphi \tag{3.10}$$

- **Globally** ($\Box$), means that the formula must hold on the entire subsequent path.

$$\pi \models \Box \varphi \textit{ iff } \forall\ i \geq 1 \mid \pi[s_i...] \models \varphi \tag{3.11}$$

- **Until** (***U***) operator evaluates if $\varphi$ is true in every state along the path until a state in which $\psi$ is true.

$$\pi \models \varphi \mathbf{U} \psi \textit{ iff } \exists\ i \geq 1 \mid \pi[s_i] \models \psi \textit{ and } \forall\ j\text{=}1,2,...i\text{-}1 \mid \pi[s_j] \models \varphi \tag{3.12}$$

- **Release** (***R***) operator evaluates if $\psi$ is true in every state along the path until a state in which both $\psi$ and $\varphi$ are true.

$$\pi \models \varphi \mathbf{R} \psi \textit{ iff } \exists\ i \geq 1 \mid \pi[s_i] \models \varphi \textit{ and } \forall\ j\text{=}1,2,...i \mid \pi[s_j] \models \psi \tag{3.13}$$

Figure 3.4 illustrates the semantics of these temporal operators, where the first bubble in the path represents the first state of the path.

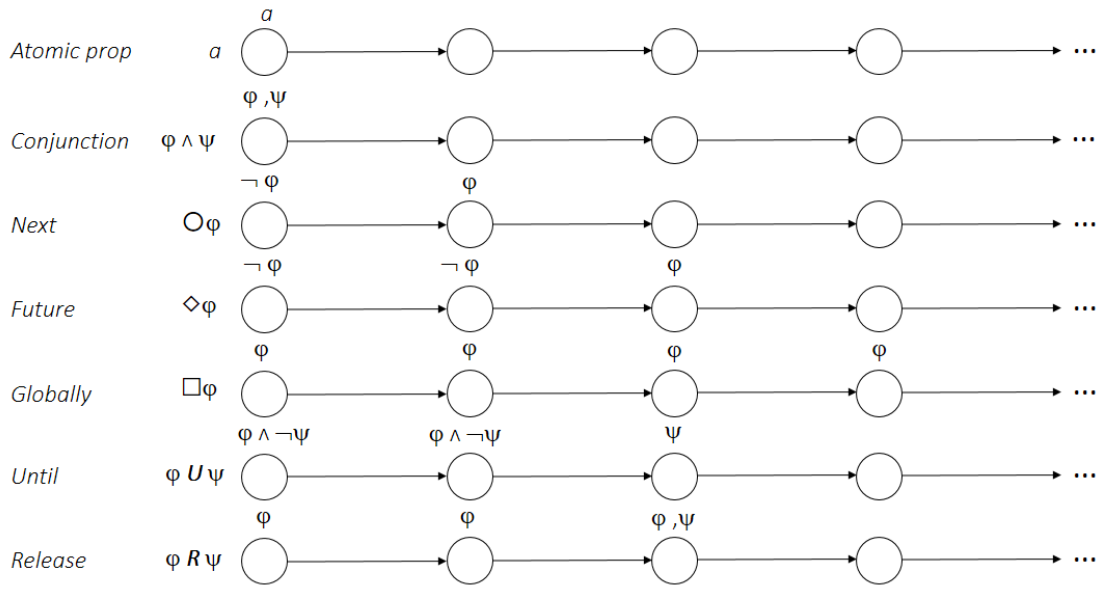



Figure 3.4: Illustration of semantics of LTL temporal operators.

### 3.3.3 Semantics of LTL over Paths and States

Until now we have defined the semantics of the LTL operators over paths. At this step it is necessary to extend this semantics to an interpretation over paths and states of a finite state system. This system can be defined as a *Kripke structure* $M = (S, I, R, L)$[9], where:

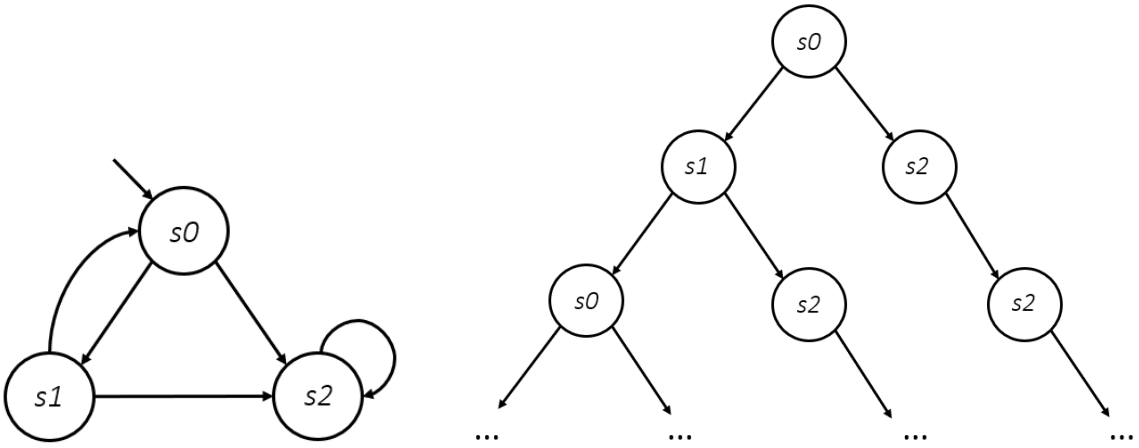
- $S$ is a finite set of states.
- $I \subseteq S$ is a set of initial states.
- $R \subseteq S \times S$ describes state transition relations. For each $s \in S$, there is $s'$ such that $s \rightarrow s'$. The transition is denoted as $(s, s') \in R$.
- $L : S \rightarrow AP$ labels states with the atomic propositions.

It can be illustrated as a transition system as in Figure 3.5a. A path $\pi$ in $M$ is an infinite sequence of states, and since a state can have more than one successor, the *Kripke* structure can be thought as unwinding into an infinite tree, representing all the possible executions of the system starting from the initial states (Figure 3.5b). $M$ satisfies an LTL formula $\varphi$ if and only if it satisfies the formula on all paths starting from any initial state[13]:

$$M \models \varphi \; \textit{iff} \; s \models \varphi \; \forall \; s \in I$$

where, LTL formula $\varphi$ is valid in state $s$ if and only if $\varphi$ is true for all paths starting from $s$:

$$s \models \varphi \; \textit{iff} \; \forall \; \pi \in \textit{Paths(s)} \mid \pi \models \varphi.$$




(a) (b)

Figure 3.5: a) Kripke structure and b) its unwinding.

The assumption of no terminal states in $M$, makes all paths be infinite. This assumption is made for simplicity; it is also possible to define the semantics of LTL for finite paths (the semantics is irrelevant whether or not $M$ is finite).

# Chapter 4

# The NuSMV Model Checking Tool

In this Chapter we will show the model checking tool to support system property verification. Possible model checker can be NuSMV, SPIN, CADP, ProB, FDR2, etc. For our purpose, it is introduced the NuSMV[1] model checker. NuSMV stands for *New Symbolic Model Verifier*, which is an automatic open-source tool developed by FBK-IRST (Italy), Carnegie Mellon University (USA), the University of Genoa (Italy) and the University of Trento (Italy), in 1998. NuSMV, released in 1993, is an extension of SMV, the first symbolic model checker based on *Binary Decision Diagram*[2] (BDD).
BDDs symbolically represent the Kripke structures for boolean formulas, which is standard when the order of the variables has been defined. A state of the system is represented by an assignment of boolean values to the set of state variables. A boolean formula, and thus its BDD, is a compact representation of the set of the states represented by the assignments which make the formula true. In the same way, the transition relation can be expressed as a boolean formula in two sets of variables, one for the current state and the other for the next state[18]. The strength of symbolic model checking is reducing the state space explosion of Kripke structure (exponential dependency on number of components), which would require too much memory to store all its states.
NuSMV has been created to support the verification for industrial designs, the custom verification tools, the formal verification techniques, and the research fields[14]. The main features of NuSMV are the following[14]:

- **Functionalities**. It allows to represent asynchronous and synchronous FSM to analyze specifications expressed as CTL and LTL, using BDD-based and SAT-based model checking techniques, for achieving efficiency and partially controlling the state explosion.

[1]The information in this Chapter comes from [9], [18], [14].
[2]http://nusmv.fbk.eu/NuSMV/papers/sttt_j/html/node4.html.

- **Architecture**. Modules define different components and functionalities of NuSMV. Modules are linked thanks to interfaces.
- **Implementation**. It is written in ANSI C. It combines the state-of-art *BDD*-package, developed at Colorado University, and the *SAT-based* model checking component that includes an *RBC-based* Bounded Model Checker, which can be connected to the *Minisat* SAT Solver and/or to the *ZChaff* SAT solver. This ensures NuSMV2 is robust, portable, efficient and easy to understand by developers.

In this work it is used NuSMV v2.6.0[3] which is distributed with an open source license.

## 4.1 Input Language

In this section it is presented the syntax and semantics of the input language of NuSMV. A vertical bar ('|') is used to separate alternatives in the syntax[14]. Any string starting with two dashes ('−−') and ending with a newline is a comment and it is ignored by the parser[14].

### 4.1.1 Types Overview

This section provides an overview of the types that are supported by NUSMV, which are Boolean, Integer, Enumeration, Word, and Array.

**Boolean** type comprises symbolic values `FALSE` and `TRUE`.

**Integer** type is simply any whole number, positive or negative in the range $-2^{31}+1$ to $2^{31}-1$.

**Enumeration** type is specified by full enumerations of all the values that the type comprises. It does not contain information about the exact values constituting the types, but only the flag whether all values are `integer numbers` (integer enumeration), `symbolic constants` (symbolic enumeration), or both (integers-and-symbolic enumeration). For example, it may be {`-1`, `1`}, {`stopped`, `running`, `waiting`, `finished`}, or {`FAIL`, `1`, `3`, `OK`}.

**Word** type defines `unsigned word[N]` and `signed word[N]`, vectors of bits which allow bitwise logical and arithmetic operations (unsigned and signed, respectively), where N represents the width.

[3] http://nusmv.fbk.eu/

**Array** declares arrays with a lower and upper bound for the index and the type of the elements in the array itself. For example, `array 0..3 of boolean` defines an array of four elements of boolean type.

Since it is intended to describe finite state machines in this work, the only data types used are Boolean and Enumeration.

### 4.1.2 Expressions

In NuSMV all expressions are constraints on the type of operands. If an expression violates the type system, then program will generate error. Expressions can be grouped in *Basic expressions* and *Next expressions*. Basic expressions can represent sets of states. Instead, next expressions link current and next state variables to express transitions as in the Kripke structure. An Identifier is an expression which references an object. Objects are instances of modules, variables, and defined symbols. The syntax of an identifier is as follows:

```
identifier :: symbol
```

#### Basic Expressions

Some of the representative basic expressions are: *Boolean*, *Integers*, *Symbolic* and *Range*.

- **Boolean Constant** is one of the symbolic values `FALSE` or `TRUE`. The syntax is the following:

```
boolean_constant :: FALSE | TRUE
```

- **Integer Constant** is an integer number.

```
integer_constant :: integer_number
```

- **Symbolic Constant** is an `identifier` with a unique value (with symbolic enumeration).

```
symbolic_constant :: identifier
```

- **Range Constant** is a set of consecutive integer numbers.

```
range_constant :: integer_number ..  integer_number
```

For example, a constant `-1..5` indicates the set of numbers `-1, 0, 1, 2, 3, 4` and `5`.

Other common expressions used in NuSMV are: *Arithmetic*, *Logic* and *Bitwise*, *Comparison* and *Conditional*.

- **Arithmetic Operators** are related to integer and word type. They are addition, subtraction, multiplication, division, and mod (remainder of the division).

```
basic_expr ::
              basic_expr + basic_expr
              | basic_expr - basic_expr
              | basic_expr * basic_expr
              | basic_expr / basic_expr
              | basic_expr mod basic_expr
```

- **Logic and Bitwise Operators** require boolean operands, or word type. The main ones are `&` (and), `|` (or), `xor` (exclusive or), `->` (implies), `<->` (if and only if), and `!` (not) (unary operator).

```
basic_expr ::
              basic_expr & basic_expr
              | basic_expr | basic_expr
              | basic_expr xor basic_expr
              | basic_expr -> basic_expr
              | basic_expr <-> basic_expr
              | !basic_expr
```

- **Comparison Operators** are: = (equality), `!=` (inequality), > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to).

```
basic_expr ::
              basic_expr = basic_expr
              | basic_expr != basic_expr
              | basic_expr > basic_expr
              | basic_expr < basic_expr
              | basic_expr >= basic_expr
              | basic_expr <= basic_expr
```

- **Case Expressions** (If-then-else) returns the value of the first expression on the right hand side of '`:`', such that the corresponding condition on the left hand side is `TRUE`.

```
case
      cond1  : expr1;
      cond2  : expr2;
      ...
      TRUE   : exprN; --otherwise
esac
```

If `cond1` then `expr1`, else if `cond2` then `expr2`, else if ... then `exprN`. If all conditions on the left hand side are `FALSE`, the program will return an error.

**Basic Next Expression**

This expression refers to the values of variables in the next state. Given a state variable `a`, `next(a)` refers to that variable `a` in the next time step, thus defining the transition relation in the FSM.

```
next(var) := next_expr;
```

`next_expr` is evaluated in the domain of `var`. If no `next()` assignment is specified for a variable, then the variable can evolve non-deterministically (i.e. it is unconstrained). Usually If-then-else and `next()` expressions are combined to allow variables to have different evolution in time.

## 4.2 Definition of the FSM

The basic building blocks of SMV models are modules. They define the set of state variables, parameters and restrictions on behaviour during the execution of the FSM. The state of the whole model consists of instantiated modules and their internal states. A module includes:

- A declaration section of the variables.
- An assignment section to define the valid initial states.
- An assignment section which defines the transition relations, describing how the variables change at each step.
- A specification section to determine if a such property is verified or not.

### 4.2.1 Variable Declarations

A variable can be *State*, *Frozen*, or *Input* variable. The type of a variable is specified by means of the declaration section, according to the variable's types specified in the paragraph 4.1.1. Each state of the model is an assignment of values to a set of state and frozen variables. A variable can take only the values from the domain of its declared type.

**State variables** can be instanced within a module (which defines the model), and within the specifications. They are declared by the notation `VAR`.

**Input variables**, declared by `IVAR`, are used to label transitions of the Finite State Machine. Input variables cannot be instanced within a module (more limited than state variables).

**Frozen Variables**, `FROZENVAR`, keep their initial value through the whole evolution of the state machine. For these variables, initial state assignments are allowed, but next state assignments are illegal and other statements can lead to some unsafe behaviours of the model. All kinds of specifications involving those variables are allowed.

The following example shows different variable declarations, where `a`, `c`, and `d` are boolean, instead `b` is defined as enumerative type, where all possible values are {`stopped, running, waiting, finished`}.

```
VAR
          a :  boolean;
          b :  {stopped, running, waiting, finished};
IVAR
          c :  boolean;
FROZENVAR
          d :  boolean;
```

### 4.2.2 Modelling Style

The behaviour of the model can be defined through constraints on variable instanced within the modules. Constraints can be `INIT`, `INVAR`, `TRANS`, `ASSIGN`, and `FAIRNESS`.

`INIT` constraint determines the set of boolean expressions that must be true in every initial state.

```
INIT expr
```

INVAR constraint defines the set of boolean expressions that must be true in every state.

```
INVAR expr
```

TRANS constraint defines the set of transition relations of the model as a set of current/next state pairs.

```
TRANS expr
```

In the case where no next statement is defined, TRANS constraint does not hold. For those cases, INVAR constraints must be used.
ASSIGN constraint assigns the current, the initial and the next value to a variable.

```
ASSIGN
        var1         := expr1;
        init(var2)   := expr2;
        next(var2)   := expr3;
```

The normal assignment, init() and next() constraints can be rewritten in terms of INVAR, INIT, and TRANS constraints, respectively, but it is not true the vice versa. FAIRNESS restricts the analysis to paths where the property is true infinitely often.

```
FAIRNESS expr;
```

A path which satisfies this kind of constraint is called *fair path*.
Modelling with ASSIGN constraint assures there are not deadlock states. On the other hand, INIT and TRANS constraints could lead to this problem[14]. To avoid this bad situation, FAIRNESS constraint can be used for each wanted behaviour of the system. Moreover, the set of initial values of the model's variables defines the initial states of the system. Defining the transition relations, means forcing the variables to assume such value in the next time step. If the initial value for a variable is not specified and/or no transition assignment is specified, then that variable initially can assume any value in its domain, and it can evolve non-deterministically.

### Restrictions on the assignments

How the FSM evolves in time is described by the assignments, which are defined by a system of equations. To guarantee that a program is implementable, restrictive syntactic rules are placed on the structure of assignments:

- Each variable may be assigned only once in the program, to avoid conflicting definitions. For example, either assign a value to a variable a (a := value) once, or init(a) and/or next(a) once, but not both at the same time for a.

- The set of equations must not have 'cycles' in its dependency graph not broken by delays, thus there is a fixed order variables computed at each step. For instance, the assignments:

```
x := y;
y := x;
```

  form a loop of dependencies whose total delay is zero. In this case there is no-deterministic way to compute x or y, since at each time instant x depends on the value of y and vice versa. To overcome it, can be introduced a "unit delay dependency" between variables, using `next()` operator.

```
x       := y;
next(y) := x;
```

If a SMV program does not respect these restrictions, an error is reported by NuSMV.

### 4.2.3 Declarations and Instantiations of the Modules

A module is a collection of declarations, constraints and specifications. In each SMV program there must be a `Module main`, which is the top-module. Modules are used in such a way that each instance of a module refers to its data structures. SMV language allows to build structural hierarchy, thus a module can contain instances of other modules. Once defined, a module can be instantiated as many times as necessary, and it is listed inside the `VAR` declaration of the parent module. All the variables defined in a module can be accessed from outside via the `DOT` notation (e.g., `<MODULE_NAME>.<variable>`). The semantics of the module instantiation follows the call-by-reference mechanism, where formal parameters are substituted by actual parameters when the module is instantiated into upper-module. For example in the following code fragment:

```
MODULE main
              ...
              VAR
              a       :  boolean;
              b       :  foo(a);   --foo instance
              ...
MODULE foo(x)
              ASSIGN
              x       := TRUE;
```

the variable a has value TRUE.
The latest versions of NuSMV support only synchronous FSM by the input language, thus there is not any synchronization problem since the program executes the model in a sequential style.

## 4.3 Specifications

Each SMV program can specify the correctness of properties on the FSM, to be sure that the modelled system is designed properly. NuSMV checker supports specifications in LTL, CTL, and others, which can be included within any module. Each specification is evaluated by NuSMV, in order to determine their truth or falsity over the FSM. When they are false, the model checker prints a counterexample, i.e. a trace of the FSM where the property is violated.

### 4.3.1 LTL specifications

LTL specifications are specified by the keyword LTLSPEC. The syntax is:

```
LTLSPEC ltl_expr;
```

where ltl_expr can include any logical expression, future expression, and past expression. The syntax of LTL formulas recognized by NuSMV is as follows:

```
ltl_expr ::
            ! ltl_expr              --logical not
            | ltl_expr & ltl_expr   --logical and
            | ltl_expr | ltl_expr   --logical or
            | ltl_expr xor ltl_expr --logical xor
            | ltl_expr -> ltl_expr  --logical implies
            -- FUTURE
            | X ltl_expr            --next state
            | G ltl_expr            --globally
            | F ltl_expr            --future
            | ltl_expr U ltl_expr   --until
            | ltl_expr V ltl_expr   --release
```

In NuSMV, LTL specifications can be analyzed both by means of BDD-based model, which investigates only infinity paths; or by means of SAT-based model which deals also with bounded paths. Depending on the verification engine used, the NuSMV may return different results for the same LTL specification. The FSM to be checked is said *total* if it returns the same result for both model checker engines, otherwise, it can lead to deadlock states[14].

### 4.3.2 Desired LTL properties

At this point it is important to define the properties in NuSMV that must be satisfied by the model to ensure its correctness. They are: $Safety$, $Liveness$, and $Fairness$ properties[9].

**Safety property** assures that two or more conditions never occur at the same time, thus avoiding bad behaviours. It can be expressed as:

```
LTLSPEC G ! (cond1 & cond2)
```

G-operator is used to state that the formula holds in every state of every path of the model.

**Liveness property** states that whenever a `request` is received, at some subsequent state a `response` is executed:

```
LTLSPEC G (request -> F response)
```

This property prevents the model from starvation condition.

**Fairness property** assures that a condition is verified in every state along every path infinitely often:

```
LTLSPEC G F cond
```

## 4.4 Running NuSMV

In this section are shown the steps to run a SMV program. NuSMV can be used either in batch or in interactive mode. The simulation of a SMV program helps the user to check the model against a set of specifications, and to explore the possible executions (traces) of the SMV model. There are three ways to generate traces:

- Deterministically (automatically generated by NuSMV);
- Randomly (automatically generated by NuSMV);
- Interactively, where the system executes and shows the possible future states step by step. The user can choose the next state and specify further constraints on the next states.

The main interaction mode of NuSMV is through an interactive shell. First of all, to launch the NuSMV shell, the path of '*bin*' directory of NuSMV must be included within the environment variables of the operating system. The model description and the specifications are written in a file with `.smv` extension. The interactive shell is activated from the system prompt as follows:

```
system_prompt> NuSMV -int file_name.smv
NuSMV>
```

The previous command starts the interactive shell (`-int`) loading the file `file_name.smv`. Once in the NuSMV shell, the system is ready to execute user commands step by step. Every command is a sequence of words:

- left-most specifies the command to be executed;
- remaining ones are arguments to the invoked command.

To quit the NuSMV shell:

```
NuSMV> quit
```

### 4.4.1 Simulation and Checking Specifications Commands

In the NuSMV shell, the system must be initialized as follows:

```
NuSMV> go
```

Once the system has been initialized in the NuSMV shell, to perform the simulation, at the beginning it is necessary to pick a state from the possible initial states to start a new trace.

```
NuSMV> pick_state -r -v
```

This command line picks and prints out a random initial state. Subsequent states are computed typing the command:

```
NuSMV> simulate -r -p
```

which randomly simulates 10 steps of a trace from the current selected state and shows variable changes during the simulation (to simulate more steps just add `-k <number_steps>`). To show all the currently simulated traces:

```
NuSMV> show_traces -a
```

To print out the number of all reachable states:

```
NuSMV> print_reachable_states
```

The number of fair transitions (transitions executed infinitely often) is printed out with the command:

```
NuSMV> print_fair_transitions
```

For checking if the transition relation is total to prevent deadlock state:

```
NuSMV> check_fsm
```

For the LTL formulas check over the system can be used:

```
NuSMV> check_ltlspec
```

which prints out if each LTL property is satisfied (`TRUE`) or not by the model. When a specification does not hold, a counterexample (i.e., a witness of the offending behaviour of the system) is produced.

# Chapter 5

# Wireless Sensor Network

In this Chapter[1], the system for the Wireless Sensor Network and its features are defined. The architecture and the different subsystems are explained to accomplish our scope. Moreover, the system is presented as two Extended FSMs. There are different advantages in using wireless sensors, such as[24]:

- Non-intrusive, to deploy them it is not required to make changes to the existing data center infrastructure. Having thousands of sensors wired to the TCP/IP network can lead to the performance degradation and system management risks.
- Suitable to changes, they can be quickly deployed and relocated, which is an important feature for the final operators.
- Low cost, they have a relatively low price in comparison with the worth of the servers, and with the purpose they accomplish.

## 5.1 Introduction to the system

This system is based on plenty of sensors which allow to obtain big amounts of data regarding each server including temperature, humidity, and power consumption. It is thought to work with up to thousands of sensors (limited by the Bluetooth network) in order to keep the data center into its operating range. This sensors can communicate with other parts of the network, sending data, and receiving actions within a network Bluetooth 5.0. The collected data can be processed and will help the final user in real-time to make decisions thus to improve the efficiency of the data center. This system is designed to be flexible and reliable, thanks to the distribution of *Gateways*, which act as joint points for the communication between the sensors

[1]The information in this Chapter comes from [21] and [22].

and the *Base Stations*, the *Core*, and the system *Service* for the user. Figure 5.1, shows the architecture of our system.

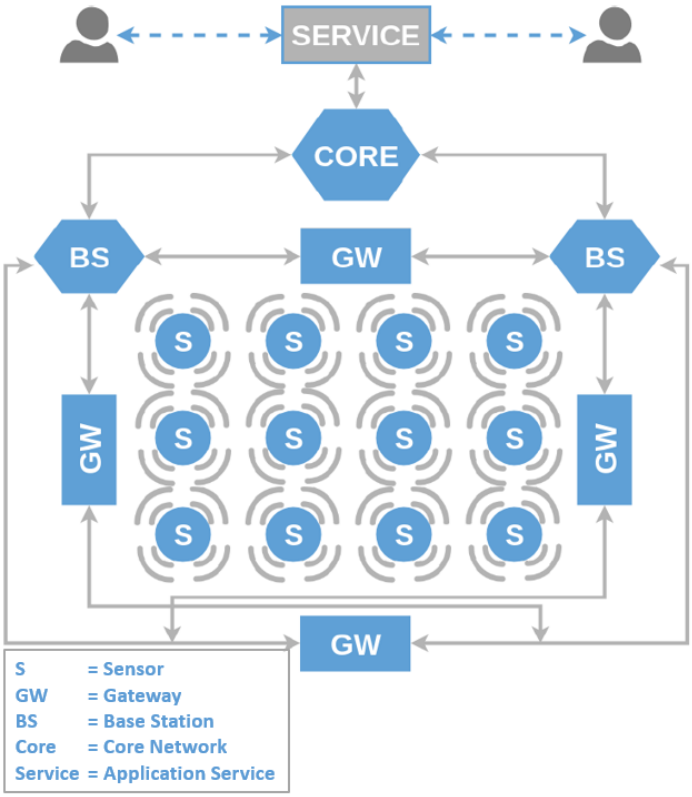



Figure 5.1: Architecture of the system[21].

A board *Nordic Semiconductor nRF*52840 *Development Kit* (nRF52840-DK)[2] and some peripherals are used to validate the operating principle of our scope. This development board implements and manages the gateways, which will communicate with the sensors. It is connected to the PC through a USB cable, which will act as base station, maintaining a direct and stable communication for the core and the service to the final user. The choice of this nRF52840-DK is due to its great communication power. Base Stations will communicate with the Core of the network through *MQTT*[3] bus, which is able to manage huge volume of data and is easily scalable. Sensors and gateways will be linked through a *Wireless Mesh Network*[19], and shown in Figure 5.2. This choice was made in order to have a better communication, security, system stability, and to have an easy reconfiguration for the final user. Mesh network topology has the ability of dynamically self-organizing

[2]https://www.nordicsemi.com/
[3]http://mqtt.org/

and self-configuring the nodes, where every node is connected to each other and everything is efficiently distributed across the network with no central entity. The process of adding a device to a specified mesh network is called *Provisioning*[4]; it turns a sensor device into a node on the network and includes security key distribution and the creation of a unique ID for the sensor being added. During this phase, the gateway executes the auto-configuration. The user who installs the sensors will

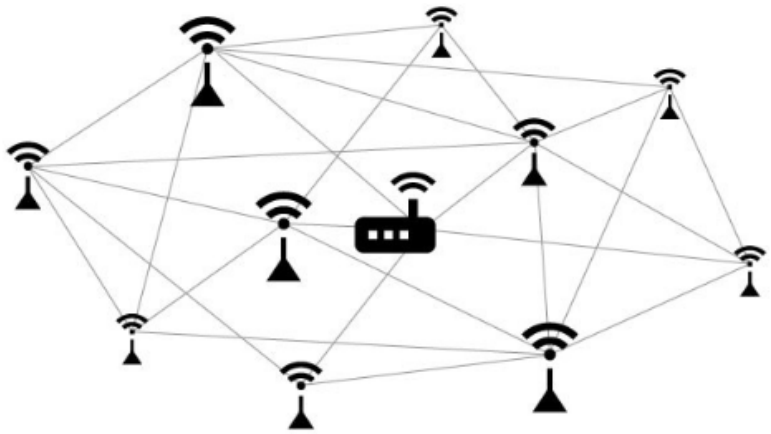


Figure 5.2: Bluetooth Mesh Network[23].

shake one of them, thus the linked gateway will receive a notification that a new sensor has been introduced in the network. The user will assign a name to it for the base station, which will interchange passwords between them, and the gateway will define its address to send data packets and configuration settings. From this point on, the sensor will be part of the system and will send measurements periodically to the base station, and will wait for operations from the gateway. Moreover, the base station will collect messages and information on the available nodes among the network. It's important also to simulate the installation process by the user to ensure the right behaviour of the this system in a real environment. The final user will obtain metrics in real-time and the historical view, to help him making decisions to optimize the cooling resources in the data center. The goal of this project is to verify, from the functional point of view, this wireless sensor network in order to reduce the time and the costs due to the development of this kind of critical real-time systems.

[4]https://www.bluetooth.com/bluetooth-technology/topology-options/mesh/mesh-glossary/

## 5.2 Requirements

This section provides the functional requirements of this system with the format in compliance with the *Easy Approach to Requirements Syntax* (*EARS*)[20]. The functional requirements include Event-driven requirements, State-driven requirements, Unwanted behaviour requirements, and Complex requirements. Event-driven requirements are invoked only when a trigger event takes place. They state what the system shall do when a specific event is detected at the system boundary, and begin with the word **WHEN**. State-driven requirements are triggered when the system enters a specific state, or mode, and they are indicated by the word **WHILE**. Unwanted behaviour requirements handle all situations that are undesirable (such as error conditions, failures, faults, disturbances and other undesired events), and they are stated in **IF/THEN** form. Complex requirements are combinations of all the previous ones. The requirements for this wireless sensor network are:

*RQ*01 - **WHILE** the node is active and **WHEN** it is provisioned in mesh network, the gateway shall maintain the node awake.

*RQ*02 - **WHILE** the node is awake and **WHEN** a notification request timer expires, the node shall send its status to the gateway every 5 seconds.

*RQ*03 - **WHILE** the node is active and **WHEN** it is commanded to remain awake, the gateway shall maintain the node awake.

*RQ*04 - **WHILE** the node is active and **WHEN** it is commanded by the gateway to go into the sleep mode, the gateway shall change the mode of this node.

*RQ*05 - **WHILE** the node is in sleep mode and **WHEN** a specific periodic timer expires, the gateway shall activate the node for 10 seconds.

*RQ*06 - **WHILE** the node is in sleep mode, **WHEN** the user applies a movement on the sensor, the gateway shall activate the node for 10 seconds.

*RQ*07 - **WHILE** the sensor is connected into the network, **WHEN** the user applies a movement on it, the gateway shall detect it and send a notification to the user through the user application.

*RQ*08 - **WHILE** the sensor is detected by the system, **WHEN** the user enters the identifier of the node through the user application, the gateway shall activate the sensor's led.

*RQ*09 - **WHILE** the sensor is detected by the system, **WHEN** the user accepts the terms to operate with it in the user application, the system shall run the autotest and the provisioning for the node into the network.

*RQ*10 - **WHILE** the node has passed the autotest and the configuration in the network, and a specific timer expires, the sensor node shall send data every 2 minutes to the gateway.

*RQ*11 - **WHILE** the node is active, **WHEN** the user performs a predefined sequence of movements on the sensor, the system shall run the autotest on the node.

*RQ*12 - **WHILE** the sensor is connected to the network, **IF** the user enters a wrong identifier through the user application, **THEN** the system shall generate an error condition.

*RQ*13 - **WHILE** the sensor is connected to the network, **IF** the user does not accept the terms to operate with it, **THEN** the system shall generate an error condition.

*RQ*14 - **WHILE** the node is active, **IF** the sensor fails the autotest, **THEN** the system shall generate an error condition.

### 5.2.1 Traceability table

Table 5.1 indicates which LTL specification covers which requirement for this system, as defined in the previous section, where each state and event is described in Table 5.2.

| **RQXX** | **LTL Specifications** |
|---|---|
| RQ01 | `G (active & already_prov -> F (keep & node_on & notify_timer))` |
| RQ02 | `G (keep & notify_req -> F notify_on)` |
| RQ03 | `G (keep & on_req -> F (!keep_timer & notify_timer))` |
| RQ04 | `G (keep & sleep_req -> F (sleep & !keep_timer & !notify_timer & !node_on))` |
| RQ05 | `G (sleep & wake_timer -> F (keep & keep_timer & node_on))` |
| RQ06 | `G (sleep & move -> F (keep & keep_timer & node_on))` |
| RQ07 | `G (active & move -> F notify_pc)` |
| RQ08 | `G (active & id -> F led)` |
| RQ09 | `G (active & accept -> F (auto_req & provis))` |
| RQ10 | `G (keep & timer_expires -> F read_send)` |
| RQ11 | `G (keep & move_seq -> F auto_req)` |
| RQ12 | `G (active & !id -> F err_1)` |
| RQ13 | `G (active & !accept -> F err_2)` |
| RQ14 | `G (active & !autotest -> F err_3)` |

Table 5.1: Requirements and LTL specifications traceability table.

| **State** | **Description** |
|---|---|
| active | Node active but not provisioned in the network |
| keep | Node active and provisioned |
| sleep | Node in sleep mode |
| **Event** | **Description** |
| already_prov | Node successfully provisioned into the network |
| notify_req | Notification request |
| notify_on | Notification sent by the node |
| sleep_req | Sleep request sent by the gateway to the node |
| on_req | Request to keep the node active |
| node_on | Status of the node (1=awake, 0=asleep) |
| wake_timer | Periodic time frame to wake up the node |
| keep_timer | Timer to keep the node awake for 10 seconds |
| notify_timer | Timer to send notification every 5 seconds |
| move | Signal when the user performs a movement on the sensor |
| notify_pc | Sensor detected by the whole system |
| id | Signal related to the ID entered by the user |
| accept | When the user accepts the terms |
| led | Signal to control the led of the sensor |
| auto_req | Autotest function |
| provis | Provisioning function |
| timer_expires | Periodic time frame to read temperature and humidity |
| read_send | Signal to read data from the sensor and send them out every 2 minutes |
| move_seq | Sequence of movements on the sensor performed by the user |
| autotest | Result of autotest function |
| err_1 | Error when the user enters a wrong ID for the node |
| err_2 | Error when the user does not accept the terms to operate with the node |
| err_3 | Error when the sensor fails the autotest |

Table 5.2: Description of the states and events for the LTL specifications.

## 5.3 System Model Definition

For this Wireless Sensor Network, we focus on two main aspects (and an EFSM has been designed for each of them):

- Initialization until the node has been provisioned, sleep mode request, and wake up request sent by the gateway. This aspect is covered by the $Configuration$ model. Moreover, this model assures as well as to keep the node in sleep mode to reduce the power consumption.
- The periodic reading from sensors, the conversion of the values in order to fit in the Bluetooth message payload, the construction and the transmission of data packet forward the base station. This aspect is covered by the *Reading* model.

### 5.3.1 Configuration Model

According to the purpose of this Wireless Sensor Network, the Configuration model has the following specifications:

- At the beginning, the sensor (node) must be provisioned in the network.
- Once it is provisioned, the node remains *on* until it receives the *sleep request* by the gateway. When the node receives a *sleep request*, it goes to the sleep mode, and remains like that until a periodic timer (*wake timer*) expires, or when a movement of the sensor is performed by the user. On both cases a timer starts, and the node remains *on* if it receives an *on request* by the gateway within 10 seconds, otherwise it goes to the sleep mode again.
- When the node is *on*, it is asked to send a $notification$ to the gateway every 5 seconds to confirm that it is active.

Figure 5.3 shows the Configuration model as Extended Finite State Machine, where the system can take one of the following states at any given time:

- **START**, where the node is waiting to be provisioned.
- **KEEP**, the node is in provisioned and awake.
- **PROV**, the node is in sleep mode.

The state transitions are:

- **S1**, when auto-configuration is completed, the node is provisioned into the mesh network (the gateway receives `already_prov`), the sensor is activated (`node_on`) and `notify_timer` starts.

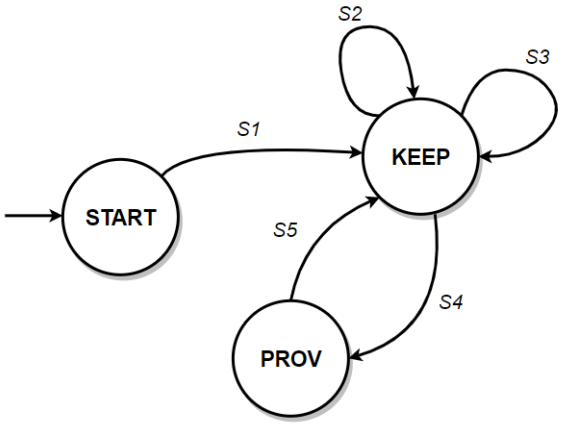



Figure 5.3: EFSM for the Configuration Model.

```
already_prov / node_on := 1, notify_timer := 1;
```

- **S2**, the node receives a request to send a notification (`notify_req`) when `notify_timer` expires. Then, the notification is sent (`notify_on`).

```
notify_req / notify_on := 1;
```

- **S3**, the gateway sends the command to the node to stay active (`on_req`). The node remains active by stopping `keep_timer`, and `notify_timer` starts.

```
on_req / keep_timer := 0, notify_timer := 1;
```

- **S4**, the node receives a command by the getaway to go into the sleep mode (`sleep_req`). Then it is in the sleep mode (`!node_on`), and `keep_timer` and `notify_timer` are stopped.

```
sleep_req /
node_on := 0, keep_timer := 0, notify_timer := 0;
```

- **S5**, the node receives a command to wake up, through either the periodic timer that expires (`wake_timer`) or the movement of the sensor by the user (`move`). The node wakes up and the `keep_timer` starts counting.

```
wake_timer | move / keep_timer := 1, node_on := 1;
```

The current state, the trigger condition, the next state, and the variable updates for the Configuration model are grouped in Table 5.3.

| Transition | Current State | Condition | Next State | Variable |
|---|---|---|---|---|
| S1 | Start | already_prov | Keep | node_on<br>notify_timer |
| S2 | Keep | notify_req | Keep | notify_on |
| S3 | Keep | on_req | Keep | !keep_timer<br>notify_timer |
| S4 | Keep | sleep_req | Prov | !node_on<br>!keep_timer<br>!notify_timer |
| S5 | Prov | wake_timer \| move | Keep | keep_timer<br>node_on |

Table 5.3: State transition table for the Configuration model.

### 5.3.2 Reading Model

For the periodic reading by the sensor and to send data packet to the base station, the Reading Model has been designed according to the following specifications:

- To make this real-time system able to interact with the user, also the possibility of a user application has been considered during the design of this Wireless Sensor Network. Initially, the sensor is placed into the network and the system starts. When the user makes a move on the sensor and it is detected by the system, the user can enter the identifier associated to the sensor into the user application to make it ready to communicate. This is also useful to ensure higher security for the network by the outside world.

- Once the user has entered the identifier and has accepted the terms to operate with it, the network system starts the autotest and the auto-configuration for provisioning the node. The autotest is performed to ensure the hardware integrity of the sensor in the network.

- If the node has passed the autotest and it is configured correctly, then it is able to read and send measurements. Once a specific timer of 2 minutes expires, the network collects the data from the sensor until a sequence of movements is performed on the sensor. When this occurs, the node must perform the autotest again.

- If during the configuration phase, errors are detected, a signal of error is generated and the network must be reset by the user.

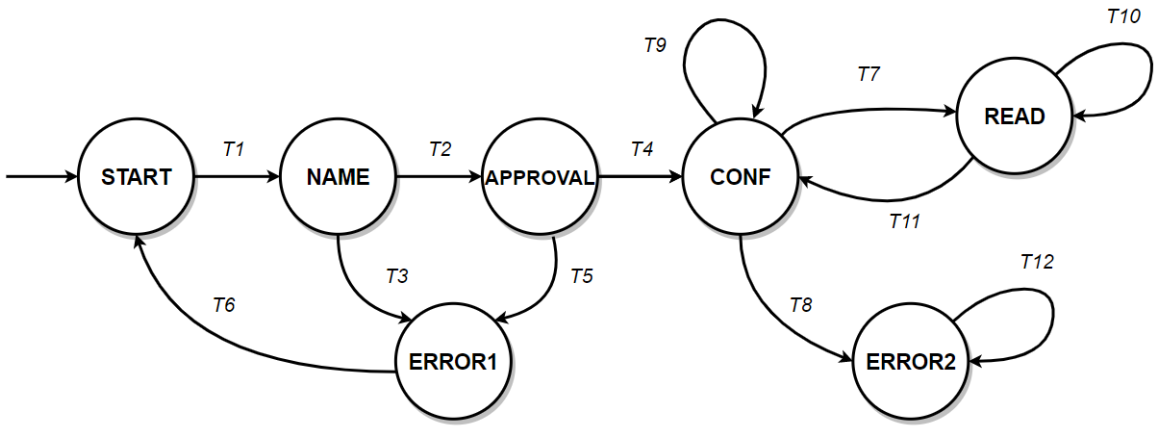



Figure 5.4: EFSM for the Reading model.

Figure 5.4 shows the Reading model as EFSM, where the machine accepts any of the states:

- **START**, the initial state and the sensor is already placed in the network.
- **NAME**, the user can enter the node's identifier into the user application.
- **APPROVAL**, the user can accept or not the terms to operate with it.
- **ERROR1**, in case of the user enters a wrong name or does not accept the terms.
- **CONF**, where the autotest and provisioning are performed.
- **READ**, when the sensor has successfully passed the autotest and the configuration phase, it is able to read and send data periodically.
- **ERROR2**, in case of error during the autotest phase.

The state transitions are:

- **T1**, the node detects the movement (`move`) and notifies the user through the user application (`notify_pc`).

```
move / notify_pc := 1;
```

- **T2**, the user enters the right identifier of the node (`id`). Then the gateway switches on the led of the sensor (`led`).

```
id / led := 1;
```

- **T3**, the user enters a wrong identifier (`!id`). An error notification is raised to the user (`err_1`).

$$\texttt{!id} \,/\, \texttt{err\_1 := 1;}$$

- **T4**, the user accepts the terms for the node (`accept`). The autotest starts (`auto_req`), and the provisioning for the node is performed (`provis`).

$$\texttt{accept} \,/\, \texttt{auto\_req := 1, provis := 1;}$$

- **T5**, the user does not accept the terms (`!accept`). A notification error is raised to the user (`err_2`).

$$\texttt{!accept} \,/\, \texttt{err\_2 := 1;}$$

- **T6**, When the errors conditions (`err_1` and `err_2`) are fixed, an event is generated (`deadline`), and the machine is restarted.

$$\texttt{!err\_1} \wedge \texttt{!err\_2} \,/\, \texttt{deadline := 1;}$$

- **T7**, the node successes the autotest (`autotest`) and it is configured correctly into the network (`config`), and then a timer starts (`init_read`).

$$\texttt{autotest} \wedge \texttt{config} \,/\, \texttt{init\_read := 1;}$$

- **T8**, an error is detected during the autotest (`!autotest`), then a notification is raised (`err_3`).

$$\texttt{!autotest} \,/\, \texttt{err\_3 := 1;}$$

- **T9**, the node is not configured into the network correctly (`!config`), then the system performs the provisioning of the node again.

$$\texttt{!config} \,/\, \texttt{provis := 1;}$$

- **T10**, every time the timer `init_read` expires (`timer_expired`), the sensor performs the reading and it is sent to the gateway (`read_send`).

$$\texttt{timer\_expired} \,/\, \texttt{read\_send := 1;}$$

- **T11**, the specific sequence of movements is performed on the sensor (`move_seq`), then the timer ends (`!init_read`) and the autotest is performed again.

`move_seq` / `init_read := 0`, `auto_req := 1`;

- **T12**, once the sensor fails the autotest, the machine will stay in ERROR2 state until it will be restarted.

`1` / –;

The current state, the trigger condition, the next state, and the variable updates for the Reading model are shown in Table 5.4.

| **Transition** | **Current State** | **Condition** | **Next State** | **Variable** |
|---|---|---|---|---|
| T1 | Start | move | Name | notify_pc |
| T2 | Name | id | Approval | led |
| T3 | Name | !id | Error1 | err_1 |
| T4 | Approval | accept | Conf | auto_req provis |
| T5 | Approval | !accept | Error1 | err_2 |
| T6 | Error1 | !err_1 & !err_2 | Start | deadline |
| T7 | Conf | autotest & config | Read | init_read |
| T8 | Conf | !autotest | Error2 | err_3 |
| T9 | Conf | !config | Conf | provis |
| T10 | Read | timer_expired | Read | read_send |
| T11 | Read | move_seq | Conf | !init_read auto_req |
| T12 | Error2 | 1 | Error2 | - |

Table 5.4: State transition table for the Reading model.

# Chapter 6

# System Simulation and Verification

The system presented in Chapter 5 and modelled in Annex A is simulated and verified within the NuSMV shell in this Chapter. The models have been simulated interactively and randomly, considering 100 steps, according to the NuSMV commands presented in Section 4.4.

## 6.1 Configuration Model Simulation

In this section the simulation result of the Configuration model $CONF\_MOD.smv$, defined in A.1, is presented.

Listing 6.1: Simulation trace of the Configuration model.

```
$ NuSMV -int CONF_MOD.smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:37:22 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson
```

```
NuSMV > go
NuSMV > pick_state -r
NuSMV > simulate -r -p -k 100
********  Simulation Starting From State 1.1   ********
Trace Description: Simulation Trace
Trace Type: Simulation
  -> State: 1.1 <-
    MOVE = FALSE
    ALREADY_PROV = FALSE
    SLEEP_REQ = FALSE
    WAKE_TIMER = FALSE
    NOTIFY_REQ = FALSE
    ON_REQ = FALSE
    sensor1.state = start
    sensor1.NOTIFY_TIMER = FALSE
    sensor1.NODE_ON = FALSE
    sensor1.KEEP_TIMER = FALSE
    sensor1.NOTIFY_ON = FALSE
  -> State: 1.2 <-
  -> State: 1.3 <-
    ALREADY_PROV = TRUE
  -> State: 1.4 <-
    sensor1.state = keep
    sensor1.NOTIFY_TIMER = TRUE
    sensor1.NODE_ON = TRUE
  -> State: 1.5 <-
    SLEEP_REQ = TRUE
  -> State: 1.6 <-
    SLEEP_REQ = FALSE
    sensor1.state = prov
    sensor1.NOTIFY_TIMER = FALSE
    sensor1.NODE_ON = FALSE
  -> State: 1.7 <-
  -> State: 1.8 <-
    MOVE = TRUE
  -> State: 1.9 <-
    MOVE = FALSE
    sensor1.state = keep
    sensor1.NODE_ON = TRUE
    sensor1.KEEP_TIMER = TRUE
  -> State: 1.10 <-
  -> State: 1.11 <-
    SLEEP_REQ = TRUE
  -> State: 1.12 <-
    SLEEP_REQ = FALSE
    sensor1.state = prov
    sensor1.NODE_ON = FALSE
    sensor1.KEEP_TIMER = FALSE
  -> State: 1.13 <-
```

```
  WAKE_TIMER = TRUE
-> State: 1.14 <-
  WAKE_TIMER = FALSE
  ON_REQ = TRUE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.15 <-
  SLEEP_REQ = TRUE
  NOTIFY_REQ = TRUE
  ON_REQ = FALSE
  sensor1.NOTIFY_TIMER = TRUE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.16 <-
  SLEEP_REQ = FALSE
  NOTIFY_REQ = FALSE
  sensor1.state = prov
  sensor1.NOTIFY_TIMER = FALSE
  sensor1.NODE_ON = FALSE
-> State: 1.17 <-
-> State: 1.18 <-
-> State: 1.19 <-
-> State: 1.20 <-
-> State: 1.21 <-
-> State: 1.22 <-
-> State: 1.23 <-
  MOVE = TRUE
-> State: 1.24 <-
  MOVE = FALSE
  WAKE_TIMER = TRUE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.25 <-
  SLEEP_REQ = TRUE
  WAKE_TIMER = FALSE
  ON_REQ = TRUE
-> State: 1.26 <-
  SLEEP_REQ = FALSE
  NOTIFY_REQ = TRUE
  ON_REQ = FALSE
  sensor1.state = prov
  sensor1.NODE_ON = FALSE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.27 <-
  MOVE = TRUE
  NOTIFY_REQ = FALSE
-> State: 1.28 <-
  MOVE = FALSE
```

```
  WAKE_TIMER = TRUE
  ON_REQ = TRUE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.29 <-
  WAKE_TIMER = FALSE
  ON_REQ = FALSE
  sensor1.NOTIFY_TIMER = TRUE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.30 <-
-> State: 1.31 <-
-> State: 1.32 <-
-> State: 1.33 <-
-> State: 1.34 <-
  SLEEP_REQ = TRUE
-> State: 1.35 <-
  SLEEP_REQ = FALSE
  sensor1.state = prov
  sensor1.NOTIFY_TIMER = FALSE
  sensor1.NODE_ON = FALSE
-> State: 1.36 <-
-> State: 1.37 <-
  MOVE = TRUE
-> State: 1.38 <-
  MOVE = FALSE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.39 <-
  SLEEP_REQ = TRUE
-> State: 1.40 <-
  SLEEP_REQ = FALSE
  sensor1.state = prov
  sensor1.NODE_ON = FALSE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.41 <-
-> State: 1.42 <-
  MOVE = TRUE
  WAKE_TIMER = TRUE
-> State: 1.43 <-
  MOVE = FALSE
  WAKE_TIMER = FALSE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.44 <-
  SLEEP_REQ = TRUE
-> State: 1.45 <-
```

```
  SLEEP_REQ = FALSE
  sensor1.state = prov
  sensor1.NODE_ON = FALSE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.46 <-
-> State: 1.47 <-
-> State: 1.48 <-
  WAKE_TIMER = TRUE
-> State: 1.49 <-
  WAKE_TIMER = FALSE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.50 <-
-> State: 1.51 <-
-> State: 1.52 <-
  SLEEP_REQ = TRUE
-> State: 1.53 <-
  SLEEP_REQ = FALSE
  sensor1.state = prov
  sensor1.NODE_ON = FALSE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.54 <-
  WAKE_TIMER = TRUE
-> State: 1.55 <-
  MOVE = TRUE
  WAKE_TIMER = FALSE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.56 <-
  MOVE = FALSE
  ON_REQ = TRUE
-> State: 1.57 <-
  ON_REQ = FALSE
  sensor1.NOTIFY_TIMER = TRUE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.58 <-
  SLEEP_REQ = TRUE
-> State: 1.59 <-
  SLEEP_REQ = FALSE
  sensor1.state = prov
  sensor1.NOTIFY_TIMER = FALSE
  sensor1.NODE_ON = FALSE
-> State: 1.60 <-
  MOVE = TRUE
  WAKE_TIMER = TRUE
-> State: 1.61 <-
  MOVE = FALSE
```

```
  WAKE_TIMER = FALSE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.62 <-
  SLEEP_REQ = TRUE
-> State: 1.63 <-
  SLEEP_REQ = FALSE
  sensor1.state = prov
  sensor1.NODE_ON = FALSE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.64 <-
  MOVE = TRUE
-> State: 1.65 <-
  MOVE = FALSE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.66 <-
  SLEEP_REQ = TRUE
-> State: 1.67 <-
  SLEEP_REQ = FALSE
  sensor1.state = prov
  sensor1.NODE_ON = FALSE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.68 <-
  MOVE = TRUE
-> State: 1.69 <-
  MOVE = FALSE
  ON_REQ = TRUE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.70 <-
  NOTIFY_REQ = TRUE
  ON_REQ = FALSE
  sensor1.NOTIFY_TIMER = TRUE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.71 <-
  NOTIFY_REQ = FALSE
  sensor1.NOTIFY_ON = TRUE
-> State: 1.72 <-
  sensor1.NOTIFY_ON = FALSE
-> State: 1.73 <-
  SLEEP_REQ = TRUE
-> State: 1.74 <-
  SLEEP_REQ = FALSE
  sensor1.state = prov
  sensor1.NOTIFY_TIMER = FALSE
```

```
  sensor1.NODE_ON = FALSE
-> State: 1.75 <-
  MOVE = TRUE
-> State: 1.76 <-
  MOVE = FALSE
  WAKE_TIMER = TRUE
  ON_REQ = TRUE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.77 <-
  WAKE_TIMER = FALSE
  ON_REQ = FALSE
  sensor1.NOTIFY_TIMER = TRUE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.78 <-
  SLEEP_REQ = TRUE
-> State: 1.79 <-
  SLEEP_REQ = FALSE
  sensor1.state = prov
  sensor1.NOTIFY_TIMER = FALSE
  sensor1.NODE_ON = FALSE
-> State: 1.80 <-
-> State: 1.81 <-
  MOVE = TRUE
-> State: 1.82 <-
  MOVE = FALSE
  WAKE_TIMER = TRUE
  ON_REQ = TRUE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.83 <-
  SLEEP_REQ = TRUE
  WAKE_TIMER = FALSE
  NOTIFY_REQ = TRUE
  ON_REQ = FALSE
  sensor1.NOTIFY_TIMER = TRUE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.84 <-
  SLEEP_REQ = FALSE
  NOTIFY_REQ = FALSE
  sensor1.state = prov
  sensor1.NOTIFY_TIMER = FALSE
  sensor1.NODE_ON = FALSE
-> State: 1.85 <-
-> State: 1.86 <-
  MOVE = TRUE
-> State: 1.87 <-
```

```
  MOVE = FALSE
  WAKE_TIMER = TRUE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.88 <-
  WAKE_TIMER = FALSE
  ON_REQ = TRUE
-> State: 1.89 <-
  SLEEP_REQ = TRUE
  ON_REQ = FALSE
  sensor1.NOTIFY_TIMER = TRUE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.90 <-
  SLEEP_REQ = FALSE
  sensor1.state = prov
  sensor1.NOTIFY_TIMER = FALSE
  sensor1.NODE_ON = FALSE
-> State: 1.91 <-
  MOVE = TRUE
  WAKE_TIMER = TRUE
-> State: 1.92 <-
  MOVE = FALSE
  WAKE_TIMER = FALSE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.93 <-
  SLEEP_REQ = TRUE
-> State: 1.94 <-
  SLEEP_REQ = FALSE
  sensor1.state = prov
  sensor1.NODE_ON = FALSE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.95 <-
  WAKE_TIMER = TRUE
-> State: 1.96 <-
  WAKE_TIMER = FALSE
  ON_REQ = TRUE
  sensor1.state = keep
  sensor1.NODE_ON = TRUE
  sensor1.KEEP_TIMER = TRUE
-> State: 1.97 <-
  ON_REQ = FALSE
  sensor1.NOTIFY_TIMER = TRUE
  sensor1.KEEP_TIMER = FALSE
-> State: 1.98 <-
  SLEEP_REQ = TRUE
-> State: 1.99 <-
```

```
    SLEEP_REQ = FALSE
    sensor1.state = prov
    sensor1.NOTIFY_TIMER = FALSE
    sensor1.NODE_ON = FALSE
  -> State: 1.100 <-
    MOVE = TRUE
    WAKE_TIMER = TRUE
  -> State: 1.101 <-
    MOVE = FALSE
    WAKE_TIMER = FALSE
    sensor1.state = keep
    sensor1.NODE_ON = TRUE
    sensor1.KEEP_TIMER = TRUE
NuSMV > quit
```

## 6.2 Reading Model Simulation

In this section is shown the simulation result of the Reading model *READ_MOD.smv*, which has been defined in A.2.

Listing 6.2: Simulation trace of the Reading model.

```
$ NuSMV -int CONF_MOD.smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:37:22 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

NuSMV > go
NuSMV > pick_state -r
NuSMV > simulate -r -p -k 100
********  Simulation Starting From State 1.1   ********
Trace Description: Simulation Trace
Trace Type: Simulation
  -> State: 1.1 <-
    MOVE = FALSE
    MOVE_SEQ = FALSE
```

```
  ID = FALSE
  ACCEPT = FALSE
  TIMER_EXPIRED = FALSE
  AUTOTEST = FALSE
  CONFIG = FALSE
  sensor1.state = start
  sensor1.NOTIFY_PC = FALSE
  sensor1.LED = FALSE
  sensor1.AUTO_REQ = FALSE
  sensor1.PROVIS = FALSE
  sensor1.READ_INIT = FALSE
  sensor1.READ_SEND = FALSE
  sensor1.DEADLINE = FALSE
  sensor1.ERR_1 = FALSE
  sensor1.ERR_2 = FALSE
  sensor1.ERR_3 = FALSE
-> State: 1.2 <-
-> State: 1.3 <-
  MOVE = TRUE
-> State: 1.4 <-
  MOVE = FALSE
  ID = TRUE
  sensor1.state = name
  sensor1.NOTIFY_PC = TRUE
-> State: 1.5 <-
  ID = FALSE
  sensor1.state = approval
  sensor1.NOTIFY_PC = FALSE
  sensor1.LED = TRUE
-> State: 1.6 <-
  sensor1.state = error1
  sensor1.ERR_2 = TRUE
-> State: 1.7 <-
  sensor1.LED = FALSE
  sensor1.ERR_2 = FALSE
-> State: 1.8 <-
  sensor1.state = start
  sensor1.DEADLINE = TRUE
-> State: 1.9 <-
  MOVE = TRUE
  sensor1.DEADLINE = FALSE
-> State: 1.10 <-
  MOVE = FALSE
  sensor1.state = name
  sensor1.NOTIFY_PC = TRUE
-> State: 1.11 <-
  sensor1.state = error1
  sensor1.NOTIFY_PC = FALSE
  sensor1.ERR_1 = TRUE
```

```
-> State: 1.12 <-
  sensor1.ERR_1 = FALSE
-> State: 1.13 <-
  sensor1.state = start
  sensor1.DEADLINE = TRUE
-> State: 1.14 <-
  sensor1.DEADLINE = FALSE
-> State: 1.15 <-
  MOVE = TRUE
-> State: 1.16 <-
  MOVE = FALSE
  ID = TRUE
  sensor1.state = name
  sensor1.NOTIFY_PC = TRUE
-> State: 1.17 <-
  ID = FALSE
  sensor1.state = approval
  sensor1.NOTIFY_PC = FALSE
  sensor1.LED = TRUE
-> State: 1.18 <-
  sensor1.state = error1
  sensor1.ERR_2 = TRUE
-> State: 1.19 <-
  sensor1.LED = FALSE
-> State: 1.20 <-
-> State: 1.21 <-
-> State: 1.22 <-
  sensor1.ERR_2 = FALSE
-> State: 1.23 <-
  sensor1.state = start
  sensor1.DEADLINE = TRUE
-> State: 1.24 <-
  sensor1.DEADLINE = FALSE
-> State: 1.25 <-
-> State: 1.26 <-
-> State: 1.27 <-
-> State: 1.28 <-
-> State: 1.29 <-
-> State: 1.30 <-
-> State: 1.31 <-
-> State: 1.32 <-
-> State: 1.33 <-
-> State: 1.34 <-
-> State: 1.35 <-
  MOVE = TRUE
-> State: 1.36 <-
  MOVE = FALSE
  ID = TRUE
  sensor1.state = name
```

```
  sensor1.NOTIFY_PC = TRUE
-> State: 1.37 <-
  ID = FALSE
  sensor1.state = approval
  sensor1.NOTIFY_PC = FALSE
  sensor1.LED = TRUE
-> State: 1.38 <-
  sensor1.state = error1
  sensor1.ERR_2 = TRUE
-> State: 1.39 <-
  sensor1.LED = FALSE
-> State: 1.40 <-
-> State: 1.41 <-
-> State: 1.42 <-
-> State: 1.43 <-
  sensor1.ERR_2 = FALSE
-> State: 1.44 <-
  sensor1.state = start
  sensor1.DEADLINE = TRUE
-> State: 1.45 <-
  MOVE = TRUE
  sensor1.DEADLINE = FALSE
-> State: 1.46 <-
  sensor1.state = name
  sensor1.NOTIFY_PC = TRUE
-> State: 1.47 <-
  MOVE = FALSE
  sensor1.state = error1
  sensor1.NOTIFY_PC = FALSE
  sensor1.ERR_1 = TRUE
-> State: 1.48 <-
  sensor1.ERR_1 = FALSE
-> State: 1.49 <-
  sensor1.state = start
  sensor1.DEADLINE = TRUE
-> State: 1.50 <-
  sensor1.DEADLINE = FALSE
-> State: 1.51 <-
-> State: 1.52 <-
-> State: 1.53 <-
  MOVE = TRUE
-> State: 1.54 <-
  MOVE = FALSE
  sensor1.state = name
  sensor1.NOTIFY_PC = TRUE
-> State: 1.55 <-
  sensor1.state = error1
  sensor1.NOTIFY_PC = FALSE
  sensor1.ERR_1 = TRUE
```

```
-> State: 1.56 <-
-> State: 1.57 <-
  sensor1.ERR_1 = FALSE
-> State: 1.58 <-
  sensor1.state = start
  sensor1.DEADLINE = TRUE
-> State: 1.59 <-
  sensor1.DEADLINE = FALSE
-> State: 1.60 <-
-> State: 1.61 <-
  MOVE = TRUE
-> State: 1.62 <-
  MOVE = FALSE
  sensor1.state = name
  sensor1.NOTIFY_PC = TRUE
-> State: 1.63 <-
  sensor1.state = error1
  sensor1.NOTIFY_PC = FALSE
  sensor1.ERR_1 = TRUE
-> State: 1.64 <-
-> State: 1.65 <-
-> State: 1.66 <-
  sensor1.ERR_1 = FALSE
-> State: 1.67 <-
  sensor1.state = start
  sensor1.DEADLINE = TRUE
-> State: 1.68 <-
  MOVE = TRUE
  sensor1.DEADLINE = FALSE
-> State: 1.69 <-
  MOVE = FALSE
  sensor1.state = name
  sensor1.NOTIFY_PC = TRUE
-> State: 1.70 <-
  sensor1.state = error1
  sensor1.NOTIFY_PC = FALSE
  sensor1.ERR_1 = TRUE
-> State: 1.71 <-
  sensor1.ERR_1 = FALSE
-> State: 1.72 <-
  sensor1.state = start
  sensor1.DEADLINE = TRUE
-> State: 1.73 <-
  sensor1.DEADLINE = FALSE
-> State: 1.74 <-
-> State: 1.75 <-
  MOVE = TRUE
-> State: 1.76 <-
  MOVE = FALSE
```

```
  sensor1.state = name
  sensor1.NOTIFY_PC = TRUE
-> State: 1.77 <-
  sensor1.state = error1
  sensor1.NOTIFY_PC = FALSE
  sensor1.ERR_1 = TRUE
-> State: 1.78 <-
  sensor1.ERR_1 = FALSE
-> State: 1.79 <-
  sensor1.state = start
  sensor1.DEADLINE = TRUE
-> State: 1.80 <-
  sensor1.DEADLINE = FALSE
-> State: 1.81 <-
  MOVE = TRUE
-> State: 1.82 <-
  MOVE = FALSE
  ID = TRUE
  sensor1.state = name
  sensor1.NOTIFY_PC = TRUE
-> State: 1.83 <-
  ID = FALSE
  ACCEPT = TRUE
  sensor1.state = approval
  sensor1.NOTIFY_PC = FALSE
  sensor1.LED = TRUE
-> State: 1.84 <-
  ACCEPT = FALSE
  AUTOTEST = TRUE
  sensor1.state = conf
  sensor1.AUTO_REQ = TRUE
  sensor1.PROVIS = TRUE
-> State: 1.85 <-
  sensor1.AUTO_REQ = FALSE
-> State: 1.86 <-
  sensor1.PROVIS = FALSE
-> State: 1.87 <-
  CONFIG = TRUE
  sensor1.PROVIS = TRUE
-> State: 1.88 <-
  TIMER_EXPIRED = TRUE
  sensor1.state = read
  sensor1.PROVIS = FALSE
  sensor1.READ_INIT = TRUE
-> State: 1.89 <-
  TIMER_EXPIRED = FALSE
  AUTOTEST = FALSE
  sensor1.READ_SEND = TRUE
-> State: 1.90 <-
```

```
    MOVE_SEQ = TRUE
    sensor1.READ_SEND = FALSE
  -> State: 1.91 <-
    sensor1.state = conf
    sensor1.AUTO_REQ = TRUE
    sensor1.READ_INIT = FALSE
  -> State: 1.92 <-
    MOVE_SEQ = FALSE
    sensor1.state = error2
    sensor1.AUTO_REQ = FALSE
    sensor1.ERR_3 = TRUE
  -> State: 1.93 <-
  -> State: 1.94 <-
  -> State: 1.95 <-
  -> State: 1.96 <-
  -> State: 1.97 <-
  -> State: 1.98 <-
  -> State: 1.99 <-
  -> State: 1.100 <-
  -> State: 1.101 <-
NuSMV > quit
```

## 6.3 Configuration Model Verification

The verification result of the property specifications over the Configuration model *CONF_MOD.smv*, as described in A.1, is shown below.

Listing 6.3: Verification of the property specifications over the Configuration model.

```
$ NuSMV -int CONF_MOD.smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:37:22 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

NuSMV > go
NuSMV > check_fsm

############################################################
The transition relation is total: No deadlock state exists
############################################################
NuSMV > check_ltlspec
-- specification  G ((sensor1.state = start & ALREADY_PROV) ->  F ((
   sensor1.state = keep & sensor1.NOTIFY_TIMER) & sensor1.NODE_ON))
   is true
-- specification  G ((sensor1.state = keep & NOTIFY_REQ) ->  F (sensor1
   .state = keep & sensor1.NOTIFY_ON))  is true
-- specification  G ((sensor1.state = keep & ON_REQ) ->  F ((sensor1.
   state = keep & !sensor1.KEEP_TIMER) & sensor1.NOTIFY_TIMER))  is
   true
-- specification  G ((sensor1.state = keep & SLEEP_REQ) ->  F (((
   sensor1.state = prov & !sensor1.KEEP_TIMER) & !sensor1.NOTIFY_TIMER
   ) & !sensor1.NODE_ON))  is true
-- specification  G ((sensor1.state = prov & WAKE_TIMER) ->  F ((
   sensor1.state = keep & sensor1.KEEP_TIMER) & sensor1.NODE_ON))  is
   true
-- specification  G ((sensor1.state = prov & MOVE) ->  F ((sensor1.
   state = keep & sensor1.KEEP_TIMER) & sensor1.NODE_ON))  is true
-- specification  G ( F sensor1.NOTIFY_TIMER)  is true
-- specification  G ( F sensor1.NOTIFY_ON)  is true
```

```
-- specification  G ( F sensor1.NODE_ON)  is true
-- specification  G ( F sensor1.KEEP_TIMER)  is true
NuSMV > print_reachable_states
############################################################################
system diameter: 9
reachable states: 22 (2^4.45943) out of 3072 (2^11.585)
############################################################################
NuSMV > print_fair_transitions
############################################################################
Fair transitions: 60 (2^5.90689) out of 3072 (2^11.585)
############################################################################
NuSMV > quit
```

## 6.4 Reading Model Verification

The verification result of the property specifications over the Reading model *READ_MOD.smv*, as described in A.2, is presented here.

Listing 6.4: Verification of the property specifications over the Reading model.

```
$ NuSMV -int READ_MOD.smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:37:22 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

NuSMV > go
NuSMV > check_fsm

############################################################
The transition relation is total: No deadlock state exists
############################################################
NuSMV > check_ltlspec
-- specification  G (((sensor1.state = read & TIMER_EXPIRED) & !
   MOVE_SEQ) ->  F (sensor1.state = read & sensor1.READ_SEND))  is
   true
```

```
-- specification  G ((sensor1.state = name & !ID) ->  F (sensor1.state
   = error1 & sensor1.ERR_1))  is true
-- specification  G ((sensor1.state = approval & !ACCEPT) ->  F (
   sensor1.state = error1 & sensor1.ERR_2))  is true
-- specification  G ( F sensor1.READ_SEND)  is true
-- specification  G ((sensor1.state = start & MOVE) ->  F (sensor1.
   state = name & sensor1.NOTIFY_PC))  is true
-- specification  G ((sensor1.state = name & ID) ->  F (sensor1.state =
    approval & sensor1.LED))  is true
-- specification  G ((sensor1.state = conf & !AUTOTEST) ->  F (sensor1.
   state = error2 & sensor1.ERR_3))  is true
-- specification  G (((sensor1.state = conf & AUTOTEST) & CONFIG) ->  F
    (sensor1.state = read & sensor1.READ_INIT))  is true
-- specification  G ((sensor1.state = error2 & !AUTOTEST) ->  G ((
   sensor1.state = error2 & !AUTOTEST) & sensor1.ERR_3))  is true
-- specification  G !(sensor1.ERR_1 & sensor1.LED)  is true
-- specification  G !(sensor1.ERR_3 & sensor1.READ_SEND)  is true
-- specification  G ((sensor1.state = approval & ACCEPT) ->  F ((
   sensor1.state = conf & sensor1.PROVIS) & sensor1.AUTO_REQ))  is
   true
-- specification  G ((sensor1.state = read & MOVE_SEQ) ->  F ((sensor1.
   state = conf & sensor1.AUTO_REQ) & !sensor1.READ_INIT))  is true
-- specification  G !(sensor1.ERR_2 & (sensor1.PROVIS & sensor1.
   AUTO_REQ))  is true
NuSMV > print_reachable_states
############################################################################
system diameter: 10
reachable states: 95 (2^6.56986) out of 917504 (2^19.8074)
############################################################################
NuSMV > print_fair_transitions
############################################################################
Fair transitions: 210 (2^7.71425) out of 917504 (2^19.8074)
############################################################################
NuSMV > quit
```

## 6.5 Verification Results

During the simulation and verification phase, the inputs to the system have been considered random, thus to keep the FSM's execution unpredictable. As shown by the *Simulation Trace*, each Model has the expected behaviour, thus to match the requirements. Both FSMs are free from deadlock states and possible logical contradiction during the design phase, thus ensuring the implementation of both of them. The system properties, as well as the Liveness properties, defined in Chapter 5 are all verified as shown executing the command `check_ltlspec`. Moreover, Safety properties have been satisfied to assure the absence of incorrect behaviours during the execution, and Fairness properties are guaranteed in case of a task makes infinitely often requests to produce a specific output. Another information provided is the total number of *Reachable states* and *Fair transitions* that each model can actually reach out of the complete state space.

# Chapter 7

# Language Containment

In this Chapter[1] it is proven that the C implemented models (Annex B) are equivalent to the SMV models (Annex A) for verification purposes. In particular, to ensure that the C implementations and the SMV models produce the same output given a particular input sequence. Given a state machine $M$, it is called language $L(M)$ the set of all behaviours for that machine. A behaviour of a state machine is an assignment of such a signal to each port $p$ of that machine. A signal is a sequence of values, one value at each reaction, according to its type set $V_p \cup \{absent\}$, represented as a function of the form[25]:

$$s_p : \mathbb{N} \rightarrow V_p \cup \{absent\}$$

A signal can be received on that port, if it is an input, or produced on that port, it is an output. Two machines are said *Language equivalent* if they have the same language, thus the same behaviour. Therefore, given two deterministic state machines $A$ and $B$, with the same inputs and outputs ports, same input signal types accepted and output signal types produced, and $L(A) \subseteq L(B)$, therefore $B$ has I/O behaviours that $A$ does not have. This case is called *Language containment* and $A$ is a *Language refinement* of $B$. Moreover, if every behaviour of $B$ is acceptable to an environment, then every behaviour of $A$ will also be accepted to that environment, and $A$ can replace $B$ for that environment. Language containment assures that any LTL formula about inputs and outputs verified on $B$ are also verified on $A$[25].

[1]The information in this chapter comes from [25].

SMV model and the implementation are deterministic state machines, so the replacement and correctness in terms of property specification are assured when the output is fired given the same input signals, as summarized in Table 7.1 and Table 7.2. Figure 7.1 and Figure 7.2 show the C implemented and SMV finite state machines for the Configuration and Reading models, respectively.

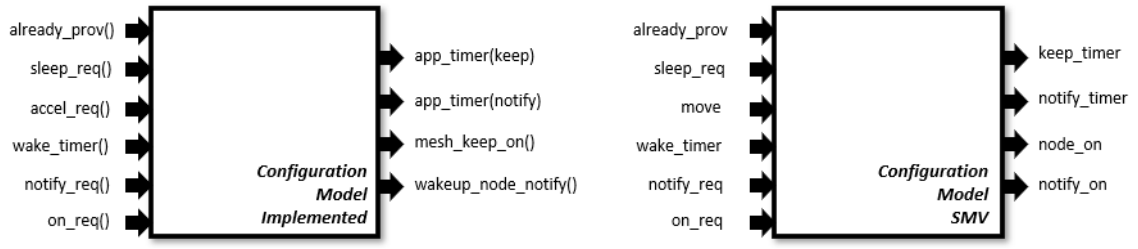



Figure 7.1: C Implementation vs SMV state machine for the Configuration model.

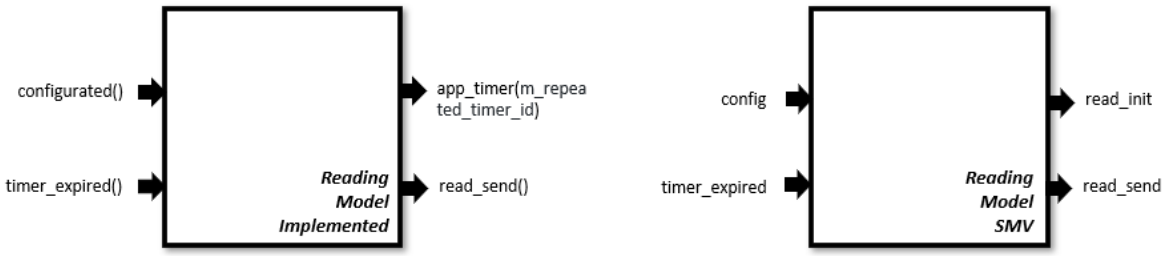



Figure 7.2: C Implementation vs SMV state machine for the Reading model.

| Configuration FSM | | | |
|---|---|---|---|
| **C Implementation** | | **SMV** | |
| **Input** | **Output** | **Input** | **Output** |
| already_prov() | app_timer_start(notify_timer)<br>mesh_keep_on(true) | already_prov | notify_timer<br>node_on |
| sleep_req() | app_timer_stop(notify_timer)<br>app_timer_stop(keep_timer)<br>mesh_keep_on(false) | sleep_req | !notify_timer<br>!keep_timer<br>!node_on |
| accel_req() | mesh_keep_on(true)<br>app_timer_start(keep_timer) | move | node_on<br>keep_timer |
| wake_timer() | mesh_keep_on(true)<br>app_timer_start(keep_timer) | wake_timer | node_on<br>keep_timer |
| notify_req() | wakeup_node_notify() | notify_req | notify_on |
| on_req() | app_timer_stop(keep_timer)<br>app_timer_start(notify_timer) | on_req | !keep_timer<br>notify_timer |

Table 7.1: I/O behaviours of C Implementation vs SMV model for the Configuration FSM.

| Reading FSM | | | |
|---|---|---|---|
| **C Implementation** | | **SMV** | |
| **Input** | **Output** | **Input** | **Output** |
| configured() | app_timer_start(m_repeated_timer_id) | config | init_read |
| timer_expired() | read_send() | timer_expired | read_send |

Table 7.2: I/O behaviours of C Implementation vs SMV model for the Reading FSM.

# Chapter 8

# Experimental Results

## 8.1 Schedulability Analysis

Task management and scheduling are important topics for real-time systems. The scheduler in Real-Time Operating System kernel is able to allocate and schedule tasks on the processor to ensure that deadlines are met. This chapter presents the techniques used for the scheduling of this system.
A *task* is a unit of work scheduled that the CPU can execute. There are three types of tasks[9]:

- *Periodic tasks*, they are cyclically executed at specific rates, which can be derived from the application requirements. Each instance of a task must complete its execution before next instance starts.

- *Aperiodic tasks*, these are single-tasks and do not have deadlines or at least soft deadlines.

- *Sporadic taks*, these have arrival times not known a priori, but with hard deadlines since they need to response on a minimum interarrival time.

In any real-time system, a task can be specified by these temporal parameters[9] (shown in Figure 8.1):

- *Release time* ($R$), the time when a task becomes available for the execution. A task can be scheduled at or after the release time.

- *Deadline* ($D$), it is the instant of time by which its execution must be completed. The amount of time between the release time and the deadline is called *Relative deadline*.

- *Execution time* ($C$), the time required to complete the task, when it is executed alone and when it has all the resources available. This parameter depends on the task's complexity and on the speed of processor.
- *Response time*, the elapsed time between the release time and the time when the task is completed.
- *Period* ($T$), the time between the release times of two consecutive instances of the task.

Release time and deadline are typically used to impose real-time constraints. Given $n$ periodic tasks, the least common multiple of the periods of the tasks is called *hyperperiod* $H$ and the secondary periods are defined as $H{=}k \cdot S$. When the schedule is defined for the first hyperperiod, then the same schedule for the following hyperperiods is repeated.

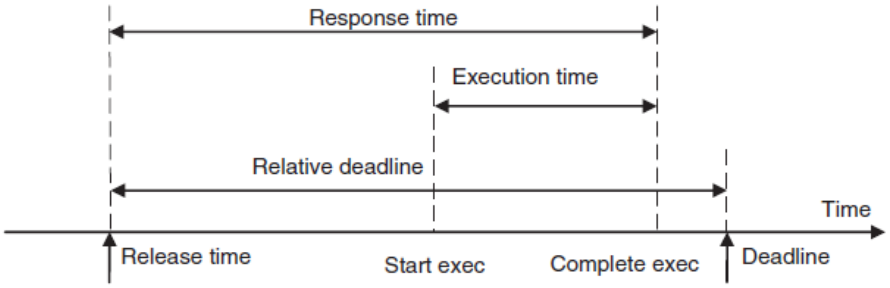



Figure 8.1: Task parameters[9].

For our system, the tasks are periodic and will run on one thread, so a *Cyclic Scheduling* can be exploited. For this purpose there are two tasks, one for each FSM. The parameters for the tasks are grouped in Table 8.1 and shown in Figure 8.2. In particular, the execution times $C$ have been approximately evaluated using the power profiler of Nordic when case the node wakes up and sends out the data (see Figure 8.14), including 10% of uncertainty, and considering that $conf\_fsm$ and $read\_fsm$ are executed by 30% and 70% respectively out of the total time. In this case $H{=}2.486s$ and $S{=}1.243s$. Figure 8.3 shows the cyclic scheduling which exploits the maximum CPU frequency.

| Task | C | T | D |
|---|---|---|---|
| $conf\_fsm$ | 339ms | 1243ms | 1243ms |
| $read\_fsm$ | 791ms | 2486ms | 2486ms |

Table 8.1: Temporal parameters for the two tasks.

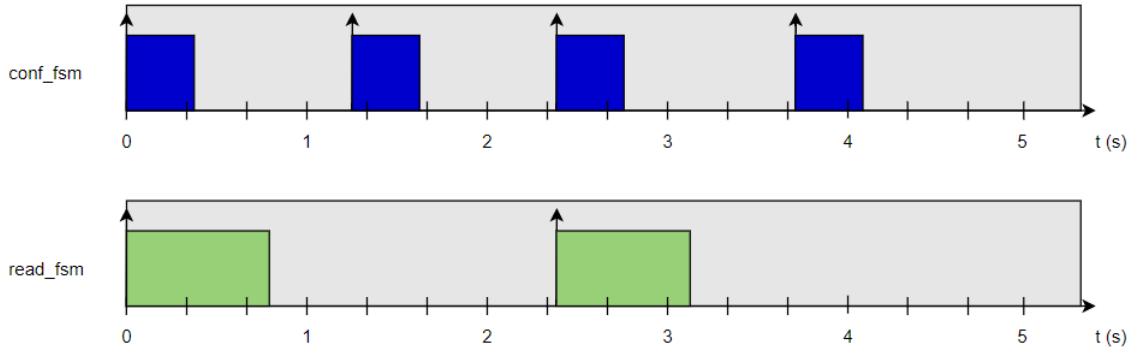



Figure 8.2: Tasks.

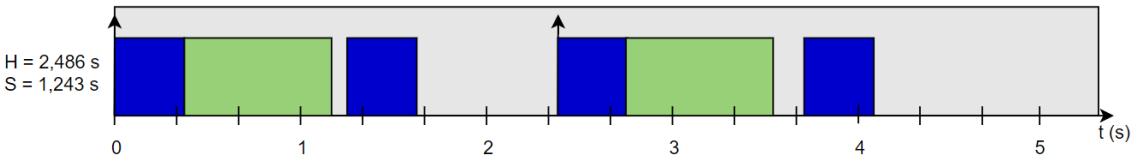



Figure 8.3: Cyclic scheduling.

Another possible scheduling is presented to reduce the dynamic power consumption of this system. Since the dynamic power depends on the frequency, we can use the *YDS Scheduling*[26]. Using the Table 8.1, YDS defines the *Intensity* in a generic time interval $(a,b)$ as

$$G[a,b] = \frac{\sum_{i=1}^{N} C_i}{(b-a)} \tag{8.1}$$

where the term $\sum_{i=1}^{N} C_i$ is the sum of the execution times of the tasks included in that interval. The highest intensity, in each interval, defines the frequency rate of the tasks to be executed in that time interval for the final scheduling. In our case:

$$G[0,1243] = 0.91$$
$$G[1243,2486] = 0.27$$

The CPU frequency of the microcontroller can operate at frequency $0.91 \cdot f_{max}$ in the first secondary period to perform the tasks, and at $0.27 \cdot f_{max}$ in the second one

to execute $conf\_fsm$, as well as assuring the deadlines. Figure 8.4 shows the YDS scheduling to reduce the dynamic power.

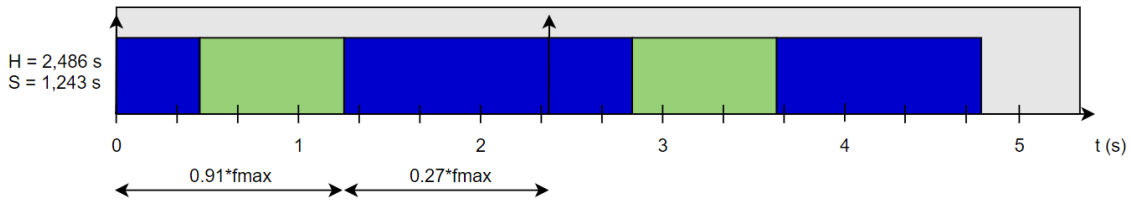



Figure 8.4: YDS scheduling.

Although YDS scheduling is able to reduce the dynamic power consumption, we cannot say a priori it is better than the classic cyclic scheduling to minimize the total power consumption. In fact, the first scheduling could minimize the total power consumption, having a lower static power consumption than the other one, thus to lead a better optimization of the overall power consumption.

## 8.2 Power consumption Analysis

In this section will be shown the power consumption results for the different phases of the node, as well as the daily average power consumption analysis, since the power consumption is one the most relevant aspects for this kind of system.

### 8.2.1 Hardware configuration

For this purpose, *Nordic* supplies the *Power Profiler Kit*[1] for power measurement and optimization of embedded systems. This tool is mounted directly on the nRF52840-DK (shown in Figure 8.6), and allows to obtain the current consumption of the sensor in real-time. The sensor used has been manufactured by the team *GreenLSI* of Polytechnic University of Madrid, and it is shown in Figure 8.5. The sensor has been connected to the *Power Profiler Kit* through the *External DUT* connector, and nRF52840-DK has been connected to the PC trough an USB cable. The current measurements are displayed through a desktop application, called *nRF Connect*, provided by *Nordic*. The whole hardware configuration to calculate the power consumption is shown in Figure 8.7.

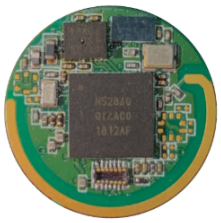


Figure 8.5: Customized sensor used to measure temperature and humidity.

[1]https://www.nordicsemi.com/Software-and-tools/Development-Kits/Power-Profiler-Kit

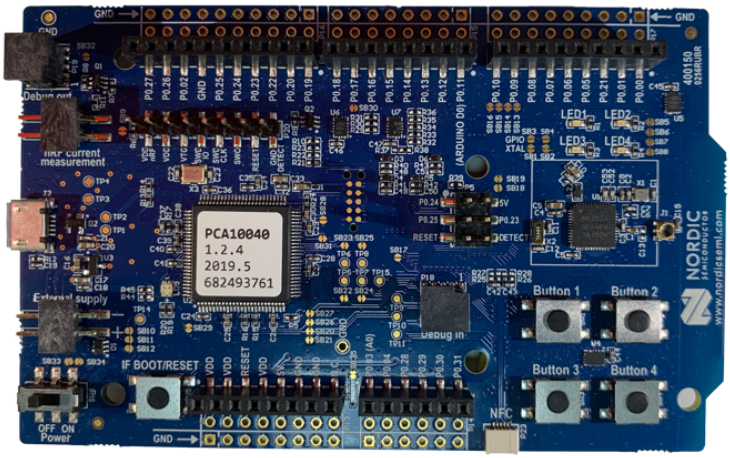



Figure 8.6: Nordic nRF52840-DK Board.

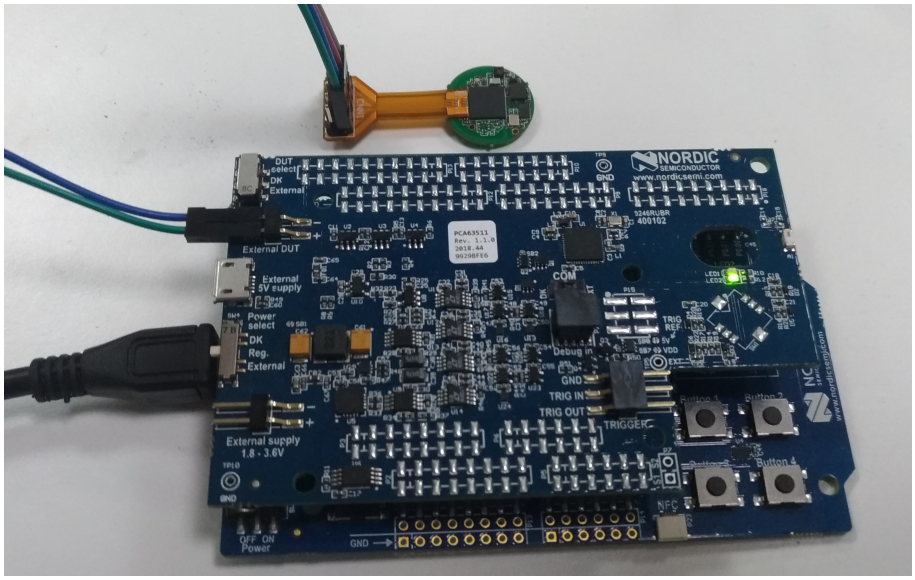



Figure 8.7: Hardware configuration.

### 8.2.2 Power consumption at different states

#### Node not provisioned and awake

When the node is awake and not provisioned in the network, it consumes power rapidly unless when there are some troughs due to periodical messages issued by it, as in Figure 8.8 through the application *nRF Connect*. This state occurs also when the node is reset or when there is no gateway present to manage the sleep mode of the node.

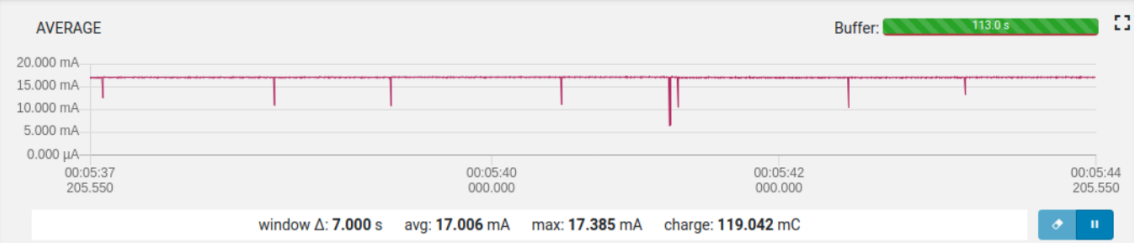



Figure 8.8: Node not provisioned in the network and awake.

#### Node provisioned

The provisioning phase is shown in Figure 8.9. The node, during the configuration, needs to be awake to exchange messages with the gateway. Again, troughs are shown due to messages exchanged.

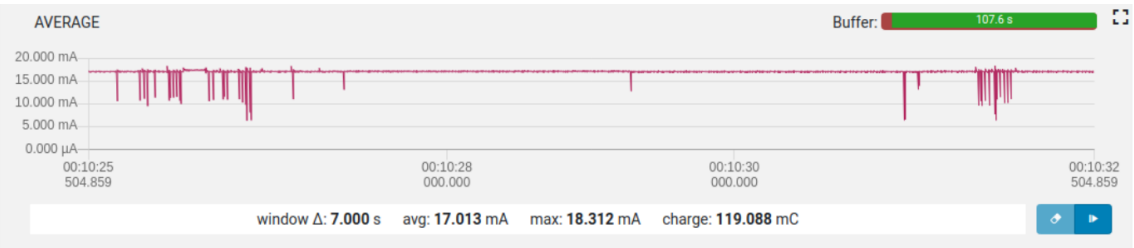



Figure 8.9: Node provisioning phase.

#### Initial sleep mode configuration

Figure 8.10 shows the initial sleep configuration. In this image, the node is awake and receives the command by the gateway to go into the sleep mode.

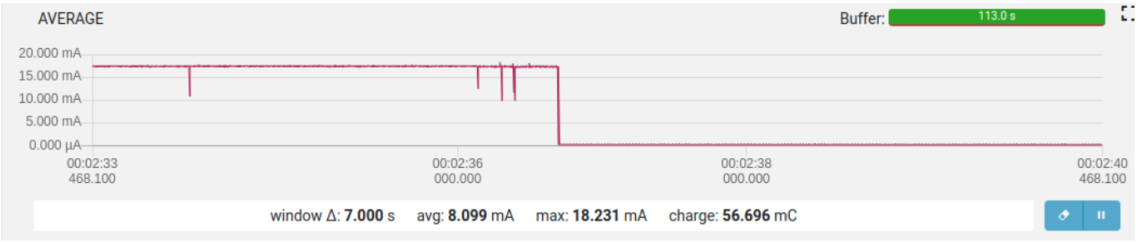



Figure 8.10: Initial sleeping phase.

### Node in sleep mode

In sleep mode (Figure 8.11), as expected, the node consumes less power, thus saving battery's charge when it is not required to operate.

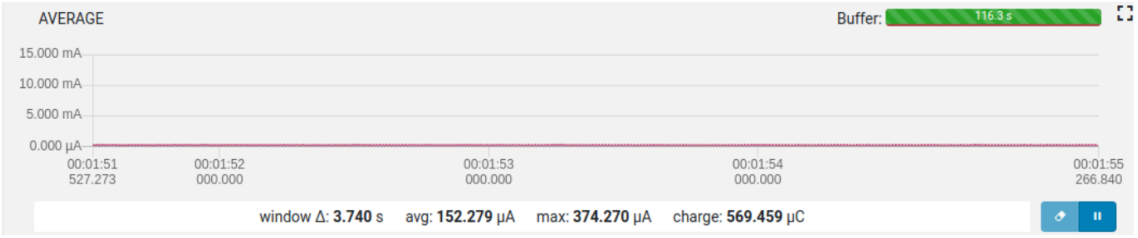



Figure 8.11: Node provisioned and in sleep mode.

### Node wake-up and not sending

Figure 8.12 shows the node provisioned in the network, that wakes up and sends a message to the gateway to declare it is on. Once the message is sent, the node is commanded by the getaway to go to sleep after almost 1 second (anyway it would go automatically after 10 seconds). This time, when the node is awake, it has a significant impact on the total power consumption.

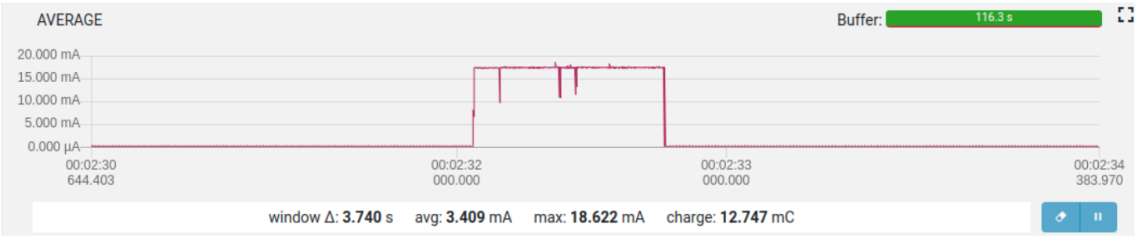



Figure 8.12: Node provisioned, wakes up and then goes into the sleep mode again.

### Node sends out the measurement

Figure 8.13 shows the measurement by sensors and the sending phase. During the first stage, the node is sleeping and then it starts to measure. After the second step, it wakes up and sends out the data packet as a BLE message using the radio channel. In this case, the node wakes up without the specific wake-up event. Therefore, once it has sent out the data, it goes to sleep automatically (unless it receives a wake-up event when it is high). A long delay ensures that the message is sent.

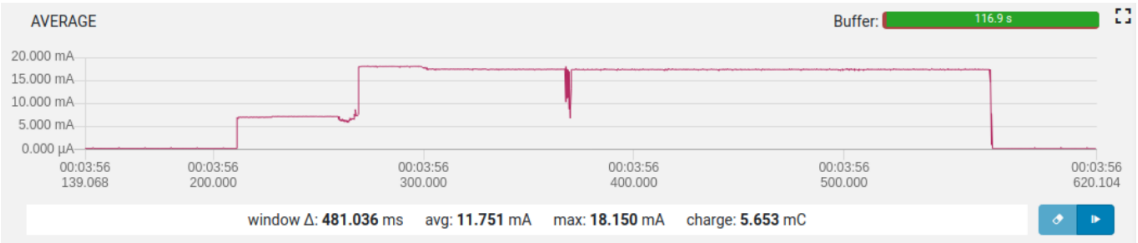



Figure 8.13: Node provisioned and sends out the measurement without the wake-up event.

The state in Figure 8.14 occurs when the node is provisioned, but the wake-up and the send events are different. Again, the first step is for measuring, then the node wakes up and sends out the data. The node sends a message to the gateway to declare it is awake and ready to execute another command. This case is more reliable than the previous one, but it requires more time to stay on, and thus consumes more power.

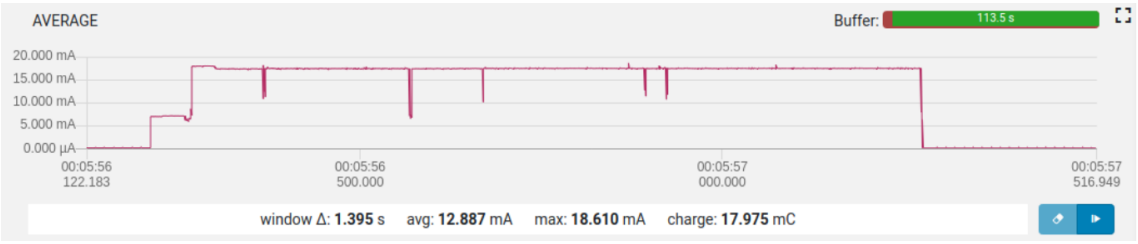



Figure 8.14: Node provisioned, wakes up and sends out the measurement.

### 8.2.3 Daily power consumption analysis

The purpose of this analysis is to quantify the daily average power consumption of this implementation (see Appendix C for more details), which is:

$$\Delta \mathrm{P}_{day} = \Delta \mathrm{P}_{send_{day}} + \Delta \mathrm{P}_{wake_{day}} + \Delta \mathrm{P}_{sleep_{day}}$$

$$\Delta \mathrm{P}_{day} = 4.78\mathrm{mAh}$$

where:

- $\Delta \mathrm{P}_{send_{day}} = 1.13\mathrm{mAh}$, the average power consumption when the node is sending.
- $\Delta \mathrm{P}_{wake_{day}} = 0.021\mathrm{mAh}$, the average power consumption when the node is awake.
- $\Delta \mathrm{P}_{sleep_{day}} = 3.63\mathrm{mAh}$, the average power consumption when the node is sleeping.

This case study does not include the power consumption during the provisioning phase, since it is only done once for each node. For an average provisioning time of 15 seconds, a waiting time frame of 3 minutes before a gateway's command and a drain of current of 17mA (as in Figure 8.9), the node consumes 0.9208mAh.

# Chapter 9

# Conclusion

The work carried out during this thesis project has been a great advance in the deployment of a wireless network of sensors in an environment that requires a very accurate and long-lasting monitoring like in data centers. The main goal has been the design and verification of the sensor node model for the wireless sensor network to detect the temperature and humidity in order to control the cooling resources in air-cooled data centers.
The system requirements have been defined in order to allow better performance and usage. This thesis project has been a first attempt to study a modern approach trying to set up a system based on wireless sensors using formal verification methodologies to monitor the servers in real-time. The sensor node has been defined with two finite state machines: one to read the temperature and humidity, and the other to manage the sleep mode of the node when it is not required to operate. The FSMs and properties specifications as LTL formulas have been coded within the NuSMV environment. Moreover, thanks to this tool, a detailed study of the behaviour of the model of the sensor node has been done. The analysis of the simulation traces provided by the tool and the continuous improvement of the node model have been critical to achieve the results obtained. In fact, model checking allows to investigate a large number of possible executions of the system in comparison with the classical testing to prevent damages for the whole infrastructure. Additionally, the verification phase has been important to support the development and the correctness of the C coded implementation of the node model on the *Nordic* board by the team *GreenLSI*. Finally, this implementation has been necessary to define the different scenarios of the power consumption of the node which have been analysed in a case of study of the daily power consumption in order to estimate the real behaviour of this node. Two possible scheduling have been presented to accomplish the tasks within a certain given time. Therefore, this work has been a great enhance in the model verification field for the wireless sensor network, based on Bluetooth mesh 5.0.

This work can be used as a base to define new functionalities of the model, for example: to define a proper user interface and the interaction with other nodes in the network to prevent collisions between them and the gateways. In the future, the sensor node can be fed by a battery, thus the optimization of the power consumption of the single node would be required in order to develop a complete wireless sensor network with several nodes that maximize the battery's life.

# Appendix A

# SMV code

In this Annex are presented the SMV codes for the Configuration Model and for Reading Model, as defined in the Chapter 5. Each program include the FSM with random inputs, and the LTL properties must be verified.

## A.1 *CONF_MOD.smv*

Listing A.1: *SMV* code for the Configuration Model.

```
1  --WIRELESS SENSOR NETWORK
2  --Configuration_FSM and LTL PROPERTIES
3
4  ---------------------------------------------------------------------
5  MODULE main
6  VAR
7      MOVE         : boolean;
8      ALREADY_PROV : boolean;
9      SLEEP_REQ    : boolean;
10     WAKE_TIMER   : boolean;
11     NOTIFY_REQ   : boolean;
12     ON_REQ       : boolean;
13
14     sensor1 : system(MOVE,ALREADY_PROV,SLEEP_REQ,WAKE_TIMER,ON_REQ,
           NOTIFY_REQ);
15
16 ASSIGN
17     init(MOVE)         := FALSE;
18     init(ALREADY_PROV) := FALSE;
19     init(SLEEP_REQ)    := FALSE;
20     init(WAKE_TIMER)   := FALSE;
21     init(NOTIFY_REQ)   := FALSE;
22     init(ON_REQ)       := FALSE;
23
```

```
24 next(ALREADY_PROV) :=
25     case
26         !ALREADY_PROV : {TRUE,FALSE};
27         TRUE          : ALREADY_PROV;
28     esac;
29
30 next(NOTIFY_REQ) :=
31     case
32        NOTIFY_REQ                    : FALSE;
33        sensor1.state=keep & ON_REQ   : {TRUE,FALSE};
34        TRUE                          : NOTIFY_REQ;
35     esac;
36
37 next(ON_REQ) :=
38     case
39        ON_REQ                                  : FALSE;
40        (sensor1.state=start & ALREADY_PROV) | WAKE_TIMER | MOVE : {TRUE
             ,FALSE};
41        TRUE                                    : ON_REQ;
42     esac;
43
44 next(SLEEP_REQ) :=
45     case
46         SLEEP_REQ          : FALSE;
47         sensor1.state=keep : {TRUE,FALSE};
48         TRUE               : SLEEP_REQ;
49     esac;
50
51
52 next(WAKE_TIMER) :=
53     case
54         WAKE_TIMER         : FALSE;
55         sensor1.state=prov : {TRUE,FALSE};
56         TRUE               : WAKE_TIMER;
57     esac;
58
59 next(MOVE) :=
60     case
61         MOVE               : FALSE;
62         sensor1.state=prov : {TRUE,FALSE};
63         TRUE               : MOVE;
64     esac;
65
66 -----------------------------------------------------------------------
67 --LTL SPECIFICATIONS
68 --LIVENESS
69 --RQ01
70 LTLSPEC G ((sensor1.state=start & ALREADY_PROV) -> F (sensor1.state =
       keep & sensor1.NOTIFY_TIMER & sensor1.NODE_ON))
```

```
71 --RQ02
72 LTLSPEC G ((sensor1.state=keep  & NOTIFY_REQ)   -> F (sensor1.state =
       keep & sensor1.NOTIFY_ON))
73 --RQ03
74 LTLSPEC G ((sensor1.state=keep  & ON_REQ)       -> F (sensor1.state =
       keep & !sensor1.KEEP_TIMER & sensor1.NOTIFY_TIMER))
75 --RQ04
76 LTLSPEC G ((sensor1.state=keep  & SLEEP_REQ)    -> F (sensor1.state =
       prov & !sensor1.KEEP_TIMER & !sensor1.NOTIFY_TIMER & !sensor1.
       NODE_ON))
77 --RQ05
78 LTLSPEC G ((sensor1.state=prov  & WAKE_TIMER)   -> F (sensor1.state =
       keep & sensor1.KEEP_TIMER & sensor1.NODE_ON))
79 --RQ06
80 LTLSPEC G ((sensor1.state=prov  & MOVE)         -> F (sensor1.state =
       keep & sensor1.KEEP_TIMER & sensor1.NODE_ON))
81
82 --FAIRNESS
83 LTLSPEC G F (sensor1.NOTIFY_TIMER)
84 LTLSPEC G F (sensor1.NOTIFY_ON)
85 LTLSPEC G F (sensor1.NODE_ON)
86 LTLSPEC G F (sensor1.KEEP_TIMER)
87
88 ------------------------------------------------------------------------
89 --SYSTEM MODULE
90 MODULE system(a,b,c,d,e,f)      --a=M1.MOVE_TEMP,b=ALREADY_PROV,c=
       SLEEP_REQ,d=WAKE_TIMER,e=ON_REQ,f=NOTIFY_REQ
91
92 VAR
93     state        : {start,keep,prov}; --set of states
94     NOTIFY_TIMER : boolean;
95     NODE_ON      : boolean;
96     KEEP_TIMER   : boolean;
97     NOTIFY_ON    : boolean;
98
99 ASSIGN
100    init(state)        := start;
101    init(NOTIFY_TIMER) := FALSE;
102    init(NODE_ON)      := FALSE;
103    init(KEEP_TIMER)   := FALSE;
104    init(NOTIFY_ON)    := FALSE;
105
106 next(state) :=
107    case
108        (state = start & b)       : keep;
109        (state = keep  & c)       : prov;
110        (state = prov  & (d | a)) : keep;
111        (state = keep  & e)       : keep;
112        (state = keep  & f)       : keep;
```

```
113         TRUE                        : state;
114     esac;
115
116 next(NOTIFY_TIMER) :=
117     case
118         c                         : FALSE;
119         state = keep  & e & !c    : TRUE;
120         state = start & b         : TRUE;
121         TRUE                      : NOTIFY_TIMER;
122     esac;
123
124 next(NOTIFY_ON) :=
125     case
126         c                               : FALSE;
127         NOTIFY_ON                       : FALSE;
128         (state = keep & f & !c & !e)    : TRUE;
129         TRUE                            : NOTIFY_ON;
130     esac;
131
132 next(NODE_ON) :=
133     case
134         (state=start & b) | d | a   : TRUE;
135         state = keep & c            : FALSE;
136         TRUE                        : NODE_ON;
137     esac;
138
139 next(KEEP_TIMER) :=
140     case
141         c | e                     : FALSE;
142         (state = prov & (d | a))  : TRUE;
143         TRUE                      : KEEP_TIMER;
144     esac;
145
146 FAIRNESS NOTIFY_TIMER;
147 FAIRNESS NOTIFY_ON;
148 FAIRNESS NODE_ON;
149 FAIRNESS KEEP_TIMER;
```

## A.2 *READ_MOD.smv*

Listing A.2: *SMV* code for the Reading Model.

```
1  --WIRELESS SENSOR NETWORK
2  --Reading_FSM and LTL PROPERTIES
3
4  ------------------------------------------------------------------------

5  MODULE main
6  VAR
7      MOVE          : boolean;   --single movement on the sensor
8      MOVE_SEQ      : boolean;   --sequence of movements on the sensor
9      ID            : boolean;
10     ACCEPT        : boolean;
11     TIMER_EXPIRED : boolean;
12     AUTOTEST      : boolean;
13     CONFIG        : boolean;
14
15     sensor1 : system(MOVE,MOVE_SEQ,CONFIG,TIMER_EXPIRED,AUTOTEST,ID,
           ACCEPT);
16
17 ASSIGN
18     init(MOVE)          := FALSE;
19     init(MOVE_SEQ)      := FALSE;
20     init(ID)            := FALSE;
21     init(ACCEPT)        := FALSE;
22     init(CONFIG)        := FALSE;
23     init(TIMER_EXPIRED) := FALSE;
24     init(AUTOTEST)      := FALSE;
25
26 next(MOVE) :=
27     case
28         sensor1.state=start  : {TRUE,FALSE};
29         TRUE                 : FALSE;
30     esac;
31
32 next(MOVE_SEQ) :=
33     case
34         sensor1.state=read   : {TRUE,FALSE};
35         TRUE                 : FALSE;
36     esac;
37
38 next(ID) :=
39     case
40         MOVE  : {TRUE,FALSE};
41         TRUE  : FALSE;
42     esac;
43
```

```
44 next(ACCEPT) :=
45     case
46         ID   : {TRUE,FALSE};
47         TRUE : FALSE;
48     esac;
49
50 next(TIMER_EXPIRED) :=
51     case
52         TIMER_EXPIRED      : FALSE;
53         AUTOTEST & CONFIG  : {TRUE,FALSE};
54         TRUE               : FALSE;
55     esac;
56
57 next(AUTOTEST) :=
58     case
59         sensor1.state=conf & AUTOTEST : AUTOTEST;
60         ACCEPT                        : TRUE;
61         MOVE_SEQ                      : {TRUE,FALSE};
62         TRUE                          : FALSE;
63     esac;
64
65 next(CONFIG) :=
66     case
67         sensor1.state=conf & !CONFIG : {TRUE,FALSE};
68         ACCEPT                       : {TRUE,FALSE};
69         TRUE                         : CONFIG;
70     esac;
71
72 ---------------------------------------------------------------------

73 --LTL PROPERTIES
74 --LIVENESS PROPERTIES
75 --RQ07
76 LTLSPEC G ((sensor1.state=start & MOVE) -> F (sensor1.state=name &
          sensor1.NOTIFY_PC))
77 --RQ08
78 LTLSPEC G ((sensor1.state=name & ID ) -> F (sensor1.state=approval &
    sensor1.LED))
79 --RQ09
80 LTLSPEC G ((sensor1.state=approval & ACCEPT) -> F (sensor1.state=conf &
     sensor1.PROVIS & sensor1.AUTO_REQ))
81 --RQ10
82 LTLSPEC G ((sensor1.state=read & TIMER_EXPIRED & !MOVE_SEQ) -> F (
    sensor1.state=read & sensor1.READ_SEND))
83 --RQ11
84 LTLSPEC G ((sensor1.state=read & MOVE_SEQ) -> F (sensor1.state=conf &
    sensor1.AUTO_REQ & !sensor1.READ_INIT))
85 --RQ12
```

```
86 LTLSPEC G ((sensor1.state=name & !ID) -> F (sensor1.state=error1 &
       sensor1.ERR_1))
87 --RQ13
88 LTLSPEC G ((sensor1.state=approval & !ACCEPT) -> F (sensor1.state=
       error1 & sensor1.ERR_2))
89 --RQ14
90 LTLSPEC G ((sensor1.state=conf & !AUTOTEST) -> F (sensor1.state=error2
       & sensor1.ERR_3))
91
92 LTLSPEC G ((sensor1.state=conf & AUTOTEST & CONFIG) -> F (sensor1.state
       =read & sensor1.READ_INIT))
93 LTLSPEC G ((sensor1.state=error2 & !AUTOTEST) -> G (sensor1.state=
       error2 & !AUTOTEST & sensor1.ERR_3))
94
95 --SAFETY
96 LTLSPEC G ! (sensor1.ERR_1 & sensor1.LED)
97 LTLSPEC G ! (sensor1.ERR_2 & (sensor1.PROVIS & sensor1.AUTO_REQ))
98 LTLSPEC G ! (sensor1.ERR_3 & sensor1.READ_SEND)
99
100 --FAIRNESS CONSTRAINT
101 LTLSPEC G F (sensor1.READ_SEND)
102
103 ------------------------------------------------------------------------
104 --SYSTEM MODULE
105 MODULE system(a1,a2,b,c,d,f,g)  --a1=M1.MOVE_TEMP, a2=M2.MOVE_TEMP, b=
       CONFIG, c=TIMER_EXPIRED, d=AUTOTEST, f=ID, g=ACCEPT, h=ERR_ID, i=
       ERR_ACCEPT, j=ERR_AUTO
106 VAR
107     state     : {start,name,approval,error1,conf,read,error2}; --states
108     NOTIFY_PC : boolean;
109     LED       : boolean;
110     AUTO_REQ  : boolean;
111     PROVIS    : boolean;
112     READ_INIT : boolean;
113     READ_SEND : boolean;
114     DEADLINE  : boolean;
115     ERR_1     : boolean;
116     ERR_2     : boolean;
117     ERR_3     : boolean;
118
119 ASSIGN
120     init(state)     := start;
121     init(NOTIFY_PC) := FALSE;
122     init(LED)       := FALSE;
123     init(AUTO_REQ)  := FALSE;
124     init(PROVIS)    := FALSE;
125     init(READ_INIT) := FALSE;
126     init(READ_SEND) := FALSE;
127     init(DEADLINE)  := FALSE;
```

```
128     init(ERR_1)     := FALSE;
129     init(ERR_2)     := FALSE;
130     init(ERR_3)     := FALSE;
131
132 next(state) :=
133     case
134         (state=name & !f) | (state=approval & !g) : error1;
135         (state=conf & !d) | state=error2          : error2;
136         (state=start & a1)                        : name;
137         (state=name  & f)                         : approval;
138         (state=approval & g) | (state=read & a2)  : conf;
139         (state=conf & d & b) | (state=read & c)   : read;
140         (state=error1 & !ERR_1 & !ERR_2)          : start;
141         TRUE                                      : state;
142     esac;
143
144 --NOTIFY_PC
145 next(NOTIFY_PC) :=
146     case
147         (state=start & a1) : TRUE;
148         TRUE               : FALSE;
149     esac;
150
151 --LED
152 next(LED) :=
153     case
154         state=error1     : FALSE;
155         (state=name & f) : TRUE;
156         TRUE             : LED;
157     esac;
158
159 --Autotest request
160 next(AUTO_REQ) :=
161     case
162         (state=approval & g) | (state=read & a2) : TRUE;
163         TRUE                                     : FALSE;
164     esac;
165
166 --Provisioning request
167 next(PROVIS) :=
168     case
169         state=conf & !b      : {TRUE,FALSE};
170         (state=approval & g) : TRUE;
171         TRUE                 : FALSE;
172     esac;
173
174 --TIMER for reading
175 next(READ_INIT) :=
176     case
```

```
177         (state=conf & d & b)  : TRUE;
178         (state=read & a2)     : FALSE;
179         TRUE                  : READ_INIT;
180     esac;
181
182 --READ AND SEND DATA
183 next(READ_SEND) :=
184     case
185         state=read & c : TRUE;
186         TRUE           : FALSE;
187     esac;
188
189 --DEADLINE
190 next(DEADLINE) :=
191     case
192         (state=error1 & (!ERR_1 & !ERR_2))  : TRUE;
193         TRUE                                : FALSE;
194     esac;
195
196 --Error ID entered
197 next(ERR_1) :=
198     case
199         state=error1 & ERR_1 : {TRUE,FALSE};
200         (state=name & !f)    : TRUE;
201         TRUE                 : FALSE;
202     esac;
203
204 --Error terms not accepted
205 next(ERR_2) :=
206     case
207         state=error1 & ERR_2  : {TRUE,FALSE};
208         (state=approval & !g) : TRUE;
209         TRUE                  : FALSE;
210     esac;
211
212 --Error autotest
213 next(ERR_3) :=
214     case
215         (state=conf & !d) : TRUE;
216         TRUE              : ERR_3;
217     esac;
218
219 FAIRNESS READ_SEND;
```

# Appendix B

# System Implementation

In this Annex it is presented an implementation[1] of this Wireless Sensor System modelled in Annex A and formally verified in Chapter 6. This Implementation has been developed in code C to exploit the tools and libraries of board *Nordic*.

## B.1 *conf_fsm.c*

The following implementation has been developed according the Configuration Model defined in A.1.

Listing B.1: C Implementation of the Configuration FSM.

```
#include <stdint.h>
#include <string.h>
#include <stdbool.h>

#include "main.h"

#define NRF_LOG_MODULE_NAME conffms
#include "nrf_log.h"
NRF_LOG_MODULE_REGISTER();

#include "app_utils.h"
#include "app_timer.h"
#include "nrfx_gpiote.h"
#include "nrfx_saadc.h"

#include "mesh_helper.h"
#include "wakeup_common.h"
#include "wakeup_node.h"
```

[1]The Implementation has been developed with the team *GreenLSI* of the Polytechnic University of Madrid.

```
19 #include "nrf_strerror.h"
20 #include "boards.h"
21
22 #include "fxos.h"
23
24 #define DEFAULT_KEEP_TIME 10000UL
25 #define DEFAULT_NOTIFY_TIME 5000UL
26
27 #define INT_PIN PIN_INT1_FXOS8700CQ
28
29 /** TIMER */
30 APP_TIMER_DEF(conf_timer);
31 APP_TIMER_DEF(keep_timer);
32 APP_TIMER_DEF(notify_timer);
33
34 /** FSM */
35 enum {
36   START,
37   KEEP,
38   PROV,
39 };
40
41 /* Global variables ************************************************/
42 static volatile bool wake_flag   = false;
43 static volatile bool accel_flag  = false;
44 static volatile bool sleep_flag  = false;
45 static volatile bool notify_flag = false;
46 static volatile bool on_flag = false;
47
48 volatile int16_t voltage = 0;
49
50 /* Function prototypes *********************************************/
51 static void timers_init(void);
52
53 static void set_wake_flag(void* p_context);
54 static void set_keep_flag(void* p_context);
55 static void set_notify_flag(void* p_context);
56
57 static int already_prov(mealy_t* self);
58 static int sleep_req(mealy_t* self);
59 static int on_req(mealy_t* self);
60 static int accel_req(mealy_t* self);
61 static int wake_timer(mealy_t* self);
62 static int notify_req(mealy_t* self);
63 static void mesh_off(mealy_t* self);
64 static void start_keep(mealy_t* self);
65 static void start_config(mealy_t* self);
66 static void start_on(mealy_t* self);
67 static void notify_on(mealy_t* self);
```

```
68
69 /* Transition Table ***************************************************/
70 static mealy_trans_t mealy_trans[] = {
71   {START, already_prov, KEEP, start_config},
72   {PROV, accel_req, KEEP, start_keep},
73   {PROV, wake_timer, KEEP, start_keep},
74   {KEEP, sleep_req, PROV, mesh_off},
75   {KEEP, notify_req, KEEP, notify_on},
76   {KEEP, on_req, KEEP, start_on},
77   {-1, NULL, -1, NULL}
78 };
79
80 /* Callbacks *********************************************************/
81 static void set_wake_flag(void* p_context)
82 {
83   wake_flag = true;
84 }
85
86 static void set_notify_flag(void* p_context)
87 {
88   notify_flag = true;
89 }
90
91 static void set_keep_flag(void* p_context)
92 {
93   sleep_flag = true;
94 }
95
96 static int already_prov(mealy_t* self)
97 {
98   return mesh_is_prov();
99 }
100
101 static int sleep_req(mealy_t* self)
102 {
103   return sleep_flag;
104 }
105
106 static int notify_req(mealy_t* self)
107 {
108   return notify_flag;
109 }
110
111 static int accel_req(mealy_t* self)
112 {
113   return accel_flag;
114 }
115
116 static int on_req(mealy_t* self)
```

```
117 {
118   return on_flag;
119 }
120
121 static int wake_timer(mealy_t* self)
122 {
123   return wake_flag;
124 }
125
126 static void mesh_off(mealy_t* self)
127 {
128   NRF_LOG_DEBUG("%s", __func__);
129   conf_fsm_t* conf = (conf_fsm_t*)self;
130   sleep_flag = false;
131   accel_flag = false;
132   notify_flag = false;
133   ERROR_CHECK(app_timer_stop(notify_timer));
134   ERROR_CHECK(app_timer_stop(keep_timer));
135   wakeup_node_sleep(&conf->wakeup_node);
136   mesh_keep_on(false);
137   DEBUG_LED_OFF(LED_2);
138 }
139
140 static void start_keep(mealy_t* self)
141 {
142   NRF_LOG_DEBUG("%s", __func__);
143   accel_flag = false;
144   wake_flag = false;
145   mesh_keep_on(true);
146   DEBUG_LED_ON(LED_2);
147   notify_flag = true;
148   nrf_saadc_value_t  vbat = 0;
149   ERROR_CHECK(nrfx_saadc_sample_convert(0, &vbat));
150   voltage = (uint16_t) (vbat * 3600.0 / 4095.0);
151   NRF_LOG_DEBUG("battery: %d mV, %x", (int)(voltage), voltage);
152   ERROR_CHECK(app_timer_start(keep_timer, APP_TIMER_TICKS(
      DEFAULT_KEEP_TIME),
153       NULL));
154 }
155
156 static void start_config(mealy_t* self)
157 {
158   NRF_LOG_DEBUG("%s", __func__);
159   accel_flag = false;
160   wake_flag = false;
161     mesh_keep_on(true);
162   DEBUG_LED_ON(LED_2);
163   notify_flag = true;
164   ERROR_CHECK(app_timer_start(notify_timer,
```

```
165       APP_TIMER_TICKS(DEFAULT_NOTIFY_TIME),
166       NULL));
167 }
168
169 static void start_on(mealy_t* self)
170 {
171   NRF_LOG_DEBUG("%s", __func__);
172   on_flag = false;
173   DEBUG_LED_ON(LED_2);
174   ERROR_CHECK(app_timer_stop(conf_timer));
175   ERROR_CHECK(app_timer_stop(keep_timer));
176   ERROR_CHECK(app_timer_start(notify_timer,
177       APP_TIMER_TICKS(DEFAULT_NOTIFY_TIME),
178       NULL));
179   notify_flag = true;
180 }
181
182 static void notify_on(mealy_t* self)
183 {
184   NRF_LOG_DEBUG("%s", __func__);
185   conf_fsm_t* conf = (conf_fsm_t*)self;
186   notify_flag = false;
187   uint32_t status = wakeup_node_notify(&conf->wakeup_node);
188   NRF_LOG_DEBUG("Notify status: %d, %s", status, nrf_strerror_get(
      status));
189 }
190
191 /* Function definitions ***********************************************/
192 static void timers_init()
193 {
194   NRF_LOG_DEBUG("Init Timers");
195
196   ERROR_CHECK(app_timer_create(
197       &conf_timer,
198       APP_TIMER_MODE_REPEATED,
199       set_wake_flag));
200   ERROR_CHECK(app_timer_create(
201       &notify_timer,
202       APP_TIMER_MODE_REPEATED,
203       set_notify_flag));
204   ERROR_CHECK(app_timer_create(
205       &keep_timer,
206       APP_TIMER_MODE_SINGLE_SHOT,
207       set_keep_flag));
208
209 }
210
211 /* Public functions ***************************************************/
212 void conf_fsm_init(conf_fsm_t* fsm)
```

```
213 {
214   timers_init();
215   mealy_init((mealy_t*)fsm, mealy_trans);
216   DEBUG_LED_ON(LED_2);
217 }
218
219 void conf_sleep_cb(uint32_t time)
220 {
221   NRF_LOG_DEBUG("Sleep cb, %d", time);
222
223   if(time == 0) {
224     on_flag = true;
225
226   } else {
227     ERROR_CHECK(app_timer_stop(conf_timer));
228     ERROR_CHECK(app_timer_start(conf_timer, APP_TIMER_TICKS(time),
229         NULL));
230     sleep_flag = true;
231   }
232
233 }
```

## B.2 *read_fsm.c*

For the Reading model, it has been implemented just the configuration, the reading, and the sending phases of the node within the network, according to A.2.

Listing B.2: C Implementation of the Reading FSM.

```
1  #include <stdint.h>
2  #include <string.h>
3  #include <stdbool.h>
4
5  #include "main.h"
6  #include "app_utils.h"
7  #include "app_timer.h"
8  #include "hts221.h"       // Temperature
9  #include "lps25hbtr.h"    // Barometer
10 #include "mesh_helper.h"
11 #include "nrf_delay.h"
12 #include "nrf_strerror.h"
13
14 /** TIMER */
15 APP_TIMER_DEF(m_repeated_timer_id);
16
17 /** FSM */
18 enum {
19   CONF,
```

```
20   READ,
21 };
22
23 /* Global variables ************************************************/
24 static volatile bool timer_flag = false;
25 static hts221_t m_hts;
26 static lps25hbtr_t m_lps;
27 extern volatile int16_t voltage;
28
29 /* Function prototypes *********************************************/
30 static void timer_init(void);
31 static void setTimerFlag(void* p_context);
32 static void hts_init(void);
33 static void lps_init(void);
34
35 static int configured(mealy_t* self);
36 static int timer_expired(mealy_t* self);
37 static void init_read(mealy_t* self);
38 static void read_send(mealy_t* self);
39
40 /* Transition Table ************************************************/
41 static mealy_trans_t mealy_trans[] = {
42   {CONF, configured, READ, init_read},
43   {READ, timer_expired, READ, read_send},
44   {-1, NULL, -1, NULL}
45 };
46
47 /* Callbacks *******************************************************/
48 static void setTimerFlag(void* p_context)
49 {
50   timer_flag = true;
51 }
52
53 static int configured(mealy_t* self)
54 {
55   return mesh_is_prov();
56 }
57
58 static int timer_expired(mealy_t* self)
59 {
60   return timer_flag;
61 }
62
63 static void init_read(mealy_t* self)
64 {
65   ERROR_CHECK(app_timer_start(
66       m_repeated_timer_id, APP_TIMER_TICKS(600000), NULL));
67 }
68
```

```
69 static void read_send(mealy_t* self)
70 {
71   temp_fsm_t* self_temp = (temp_fsm_t*)self;
72   timer_flag = false;
73   hts221_value_t temp;
74   hts221_value_t humd;
75   lps25hbtr_value_t press;
76   lps25hbtr_value_t temp2;
77   ERROR_CHECK(hts221_read(&m_hts, &temp, &humd));
78   ERROR_CHECK(lps25hbtr_read(&m_lps, &press, &temp2));
79
80   // Convert data from float to int16
81   int16_t temp_aux = (int16_t) (temp * 100.0); // ?C x100
82   uint8_t humd_aux = (uint8_t) (humd); //%rh
83   int32_t press_aux = (int32_t) (press * 100.0);
84   uint8_t voltage_aux = (uint8_t) ((voltage-1000)/10); //V x100
85
86   NRF_LOG_DEBUG("Send temp(x100): %d humd(x10.000): %d press(hPa x100):
       %d", temp_aux, humd_aux, press_aux);
87
88   DEBUG_LED_ON(LED_3);
89   mesh_enable();
90   uint32_t status = nrftemp_node_send_unreliable(
91     &self_temp->nrftemp_node, voltage_aux, temp_aux, humd_aux,
     press_aux, 1);
92   NRF_LOG_DEBUG("Send status: %d, %s", status, nrf_strerror_get(status)
     );
93   DEBUG_LED_OFF(LED_3);
94 }
95
96 /* Function definitions ***********************************************/
97 static void hts_init(void)
98 {
99   hts221_conf_t hts_conf = {
100     .frec     = HTS221_FREC_ONE_SHOT,
101     .avtemp   = HTS221_AVTEMP_128,
102     .avhumd   = HTS221_AVHUMD_128,
103     .cb       = NULL,
104     .drdy_pin = PIN_DRDY_HTS221,
105   };
106
107   ERROR_CHECK(hts221_init(&m_hts, &m_twi, &hts_conf));
108   NRF_LOG_DEBUG("HTS init");
109 }
110
111 static void lps_init(void)
112 {
113   lps25hbtr_conf_t lps_conf = {
114     .frec     = LPS25HBTR_ONE_SHOT,
```

```
115     .cb       = NULL,
116     .drdy_pin = PIN_INT_DRD_LPS25HBTR,
117     .timeout  = 0,
118   };
119   ERROR_CHECK(lps25hbtr_init(&m_lps, &m_twi, &lps_conf));
120   NRF_LOG_DEBUG("LPS init");
121 }
122
123 static void timer_init()
124 {
125   NRF_LOG_DEBUG("Init Timer");
126
127   // Create timers
128   ERROR_CHECK(app_timer_create(&m_repeated_timer_id,
129       APP_TIMER_MODE_REPEATED,
130       setTimerFlag));
131 }
132
133 /* Public functions ***************************************************/
134 void temp_fsm_init(temp_fsm_t* fsm)
135 {
136   timer_init();
137   hts_init();
138   lps_init();
139   mealy_init((mealy_t*)fsm, mealy_trans);
140   NRF_LOG_DEBUG("%s: Init temp fsm", __func__);
141 }
```

# Appendix C

# Total power consumption calculation

In this Appendix will be shown the steps used to calculate the daily power consumption. Considering the power consumption for the different phases:

**Sending phase**

The required time, the average current flowing in the node according to the Figure 8.13, and the average power consumption in order to send the measurement are:

$$\begin{aligned} T_{send} &= 0.481s \\ \Delta I_{send} &= 11.751mA \\ \Delta P_{send} &= \Delta I_{send} \cdot (T_{send})_h = \\ &= 0.00157mAh \end{aligned}$$

The total number of times the node sends the measurements in a day, sending every 2 minutes:

$$N_{send} = \frac{24h}{Period_{send}(h)} = \frac{24h}{0.0333h} = 720 \tag{C.1}$$

Then, the daily average power consumption when the node sends is:

$$\Delta P_{send_{day}} = \Delta P_{send} \cdot N_{send} = 1.13mAh \tag{C.2}$$

### Awake phase

According Figure 8.12, the required time, the average current flowing, and the average power consumption when the node is awake are:

$$\begin{aligned} T_{wake} &= 3.74s \\ \Delta I_{wake} &= 3.409mA \\ \Delta P_{wake} &= \Delta I_{wake} \cdot (T_{wake})_h = \\ &= 0.00354mAh \end{aligned}$$

The total number of times the node is awake, waking up every 4 hours, is:

$$N_{wake} = \frac{24h}{Period_{awake}(h)} = \frac{24h}{4h} = 6 \tag{C.3}$$

The daily average power consumption when the node is awake:

$$\Delta P_{wake_{day}} = \Delta P_{wake} \cdot N_{wake} = 0.021mAh \tag{C.4}$$

### Sleep mode

The time, the average current flowing, and the average power consumption during the sleep mode, according Figure 8.11, are:

$$\begin{gathered} T_{sleep} = (24h - (T_{send})_h \cdot N_{send} - (T_{wake})_h \cdot N_{wake}) = 23.89h \\ \Delta I_{sleep} = 0.152mA \end{gathered}$$

The daily average power consumption when the node is in sleep mode:

$$\Delta P_{sleep_{day}} = \Delta I_{sleep} \cdot T_{sleep} = 3.63mAh \tag{C.5}$$

Finally, the daily average power consumption results in:

$$\Delta P_{day} = \Delta P_{send_{day}} + \Delta P_{wake_{day}} + \Delta P_{sleep_{day}} = 4.78mAh \tag{C.6}$$

# Bibliography


[1] A. Capozzoli, G. Primiceri. *Cooling systems in data centers: state of art and emerging technologies*. Energy Procedia, Volume 83, 2015, Pages 484-493, ISSN 1876-6102, https://doi.org/10.1016/j.egypro.2015.12.168. (http://www.sciencedirect.com/science/article/pii/S1876610215028337).

[2] Katie Costello. *Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.5 Percent in 2019*. Gartner, STAMFORD, Conn. April 2, 2019.

[3] Naomi Xu Elegant. *The Internet Cloud Has a Dirty Secret*. https://fortune.com/2019/09/18/internet-cloud-server-data-center-energy-consumption-renewable-coal/, Fortune, September 18, 2019.

[4] Nicola Jones. *How to stop data centres from gobbling up the world's electricity*. https://www.nature.com/articles/d41586-018-06610-y, Sep. 18, 2019.

[5] Patricia Arroba Garcìa. *Proactive Power and Thermal Aware Optimizations for Energy-Efficient Cloud Computing*. Universidad Politécnica De Madrid, PhD Thesis, 2017.

[6] J. G. Koomey. *Growth in data center electricity use 2005 to 2010*. Analytics Press, Oakland, CA, Tech. Rep., Aug, 2011, p. 24.

[7] J. Wan, X. Gui, S. Kasahara, Y. Zhang and R. Zhang. *Air Flow Measurement and Management for Improving Cooling and Energy Efficiency in Raised-Floor Data Centers: A Survey*. in IEEE Access, vol. 6, pp. 48867-48901, 2018, doi: 10.1109/ACCESS.2018.2866840.

[8] Condair. *Data Centre Humidification*. https://www.condair.sg/knowledge-hub/data-centre-humidification.

[9] J. Wang. *Real-time embedded systems*. John Wiley & Sons, Inc, First Edition, 2017.

[10] S. Anderson, M. Cole, P. Jackson, M. Grohe, D. Sannella, and M. Cryan. *Inf1A: Non-deterministic Finite State Machines*. http://www.inf.ed.ac.uk/teaching/courses/inf1/cl/notes/Comp3.pdf, The University of Edinburgh, School of Informatics. 2004.

[11] Shi-Yu Huang. *On Speeding Up Extended Finite State Machines Using Catalyst Circuitry*. Conference: Design Automation Conference, 2001, Proceedings of the ASP-DAC 2001, Asia and South Pacific, Feb. 2001.

[12] M. Tranchero, E. Bellocchia, D. Boyang, G. Causapruno, A. Moré, N. Ostadabbasi, J.C. Wang, L. Lavagno. *Modeling and Optimization of Embedded Systems*. Politecnico di Torino, Department of Electronics and Telecommunication Engineering, Jan. 2019.

[13] C. Baier, J.- P. Katoen. *Principles of model checking*. MIT, 2008.

[14] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, A. Tchaltsev. *NuSMV 2.6 User Manual*. FBK-irst, 2010.

[15] N. Engbers, E. Taen. *Green Data Net. Report to IT Room INFRA, European Commision*. FP7 ICT 2013.6.2, Nov. 2014.

[16] A. Donoghue, P. Inglesant, and A. Lawrence. *The EU dreams of renewable-powered datacenters with smart-city addresses, "https://451research.com"*. Oct. 2013.

[17] J. Hamilton. *Cooperative expendable micro-slice servers (cems): low cost, low power servers for internet-scale services*. In Proceedings of the 4th Biennial Conf. Innovative Data Systems Research, ser. CIDR '09, Asilomar, CA, USA, 2009.

[18] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri *NuSMV: a new symbolic model checker*. International Journal on Software Tools for Technology Transfer 2(4):410-425. March 2000. DOI 10.1007/s100090050046.

[19] Yu Liu, Kin-Fai Tong, Xiangdong Qiu, Ying Liu, and Xuyang Ding. *Wireless Mesh Networks in IoT Networks*. 2017 International Workshop on Electromagnetics: Applications and Student Innovation Competition, London, 2017, pp. 183-185.

[20] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak. *EARS (Easy Approach to Requirements Syntax)*. IEEE, October 2009.

[21] A. E. Vilar. *Análisis y diseño de herramientas de verificación y testeo automatizado para aplicaciones de internet de las cosas*. Universidad Politécnica De Madrid, Bachelor's Thesis, June 2019.

[22] A. V. Pérez. *Desarollo de un firmware optimizado en consumo para una red mallada de dispositivos bluetooth*. Universidad Politécnica De Madrid, Bachelor's Thesis, July 2019.

[23] *Rete Mesh, come funziona la tecnologia Wi-Fi del futuro?*. https://www.ilsoftware.it/articoli.asp?tag=Rete-mesh-cos-e-e-come-funziona_18101. Dec 2018.

[24] Jie Liu, Bodhi Priyantha, Feng Zhao, Chieh-Jan mike Liang, Qiang Wang, Sean Jame. *Towards discovering data center genome using sensor nets*. HotemNets' 08, June 2-3, 2008, Charlottesville, VA.

[25] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. Second Edition, MIT Press, 2017.

[26] Lothar Thiele and Jian-Jia. *Energy-Efficienct Real-Time Task Scheduling*. Embedded Systems and Wireless Networking Laboratory.

https://www.csie.ntu.edu.tw/ r95093/files/slides/Energy%20Efficiency%20Real-Time%20Scheduling.pdf