# FRAMEWORK FOR WIRELESS SENSOR NETWORKS CODE GENERATION FROM FORMAL SPECIFICATION

Sara Houhou[1], Laid Kahloul[1], Saber Benharzallah[2] and Roufaida Bettira[1]

[1]LINFI laboratory, Computer Science Department, Biskra University, Algeria
[2]LINFI laboratory, Department of Computer Science, Batna 2 University, Batna, Algeria

***ABSTRACT***

*The development of embedded applications (such as Wireless Sensor Network protocols) often requires a shift to formal specifications. To insure the reliability and the performance of the WSNs, such protocols must be designed following some methods reducing error rate. Formal methods (as Automata, Petri nets, algebra, logics, etc.) were largely used in the specification of these protocols, their analysis and their verification. After that, their implementation is an important phase to deploy, test and use those protocols in real environments. The main objective of the current paper is to formalize the transformation from high-level specification (in Timed Automata) to low-level implementation (in NesC language and TinyOs system) and to automate such transformation. The proposed transformation approach defines a set of rules that allow the passage between these two levels. We implemented our solution and we illustrated the proposed approach on a protocol case study for the "humidity" and "temperature" sensing in WSNs applications.*

***KEYWORDS***

*Formal Methods, Timed Automata, Wireless Sensor Network, Code Generation, Automatic Transformation*

## 1. INTRODUCTION

Embedded systems are used in several domains: robotics, smart cars, wireless sensor networks (WSNs), etc. WSNs [1] are composed of tiny embedded devices. They can be defined as a distributed sensors system with an auto-configured infrastructure. To guarantee the reliability and performance in the development of these systems, the use of formal methods represents an ambitious issue. Formal methods allow high-level specification, avoid ambiguity and provide verification techniques. However, the passage from high-level description to the concrete implementation remains an informal step and error prone. The formalisation of this passage and its automation is an attractive research field.

In this paper, we propose a code generator tool that takes as input a formal specification of WSN protocols and generates implementation files in NesC language [2], which is a C dialect language, intended for programming structured component based applications for the TinyOs [2] operating system. admittedly, the approach takes as input a specification written in Timed Automata (TA)

[3] provided and verified by UPPAAL [4] model-checker tool and generates a source code written in the NesC [2] language which is the most used in WSNs development. The choice of TA relapsed to time aspect, which is an important characteristic of WSNs. In fact, this paper makes the following contributions: (1) Proposes an extension for timed automata model, (2) Defines a set of transformation rules from TA elements to NesC code, and (3) Implements in Python the generator code tool that takes the TA model as input and generates the corresponding source code.

The reminder of this paper is organised as follows: Section 2 presents related work. Section 3 presents the proposed approach. Section 4 presents the proposed rules and restrictions for the transformation. Section 5 presents the algorithm of the code generation tool. Section 6 presents the application of the proposed approach on a WSN protocol, and finally Section 7 concludes the paper.

## 2. RELATED WORK

Different approaches to WSN application development can be found in the literature. The work [5] presents a model driven development (MDD) approach for the of WSN applications. The authors define three meta-model at different level of abstraction, with automatic model transformations between them. Moreover, the input of their approach is a domain specific language (DSL) [6]. At first, they define two model-to-model transformation successively which are: DSL model to a subset of UML [7] meta-model for describing a structure of the system and model resulting to NesC meta-model, then, they define model-to-code transformation from NesC meta-model to NesC code. This approach is based on semi-formal specification instead of formal language as proposed in our approach. In [8], the authors present another MDD process to convert a high-level abstract model to a concrete model. The input of their approach is Domain Specific Modelling Language (DSML) [9], and then it is transformed into an executable model TinyDB [10] by using the model transformation rule. The key limitation in their research is that the DSMLs have the capability to describe static applications, meanwhile not to describe mobility and adaptive behaviour of WSNs. In addition, the authors do not conserve the consistency between the models. In [11]–[13], the authors present an approach for developing the DMAMAC protocol proposed in [11]. In [12], the authors design the protocol with an abstract formal model (TA), and then they implement it in NesC language. This approach is based on a manual transformation, thus it may induce lot of errors, and consumes time and efforts.

Recently, several authors [14]–[17] have proposed model driven software engineering (MDSE) approach for WSNs code generation from Coloured Petri Nets (CPNs). Inge *et.al.* [14] describe an automatic code generation from a restricted class of CPNs, annotated with code generation pragmatics, called Pragmatics annotated Coloured Petri nets (PA-CPN) [15] models, to a set of target language (groovy [18], Java [19], Clojure [20], and Python [21]). The tool PetriCode described in [16] shows the implementation of this approach. The authors in [17] used this approach on an industrial sized protocol, which is IETF WebSocket, and they generated its implementation for the Groovy platform. However, the use of pragmatics approach to generate code for any target platform restrict the class of CPN models and make it not suitable for TinyOs. In [22], the authors expanded the PetriCode tool [16] to specific platform model to generate the implementation of sensor network protocols. The input of this approach is Coloured Petri Nets models of protocols, then, the resulting CPN model is given to PetriCode tool, which transformed it automatically to code for (MiXiM [23] simulator, and TinyOs platforms, C++, and NesC respectively. In [24], the authors completed the code generation for the two specific platforms and they tested it in real-world deployment Zolertia Z1 motes. However, this last approach ignores the stochastic nature and real time constraints of WSNs. In [25], another approach based on pragmatics is presented. The approach describes a transformation from CPNs to NesC code running in TinyOs platform. The construction of CPNs models follows five stages of manual refinement. After the fifth stage, the source code of protocol is generated automatically for the TinyOs platform using a prototype software. The major drawback of this approach is that the refinement steps are specified informally. In addition, the approach is based on manual refinement of models, which may induce errors.

## 3. APPROACH DESCRIPTION

The main purpose of the paper is the automatic code generation to facilitate the development of WSN applications. The figure 1 represents the global architecture of the proposed approach, illustrating the use of two tools: UPPAAL and the "Code Generator tool". Firstly, the UPPAAL tool is used to model the application and to verify the WSNs model protocols. Secondly, the "Code Generator tool" which represents the implementation of our transformation approach. The input of the approach is a formal protocol description (i.e., a network of Timed Automata) and a set of modelling restrictions. The timed automata network is specified using UPPAAL tool and the designer of the system must respect the modelling restrictions (introduced in this paper) to assist the generation of a complete code. These restrictions oblige the designer to add specific keywords on timed automata (as procedures and synchronization variables). These later represent functional information linked with patterns of the target language. After the modelling, we check the properties to be verified, if they are satisfied, the "Code Generator tool" takes as input the resulted and verified model and transforms it into an implementation by applying the proposed mapping rules. The result of the mapping phase is an instance of the implementation. The generated implementation represents a large part of the real code of the protocol but it needs some additions to be emulated in real hardware or simulated in a simulator tools as TOSSIM [26].

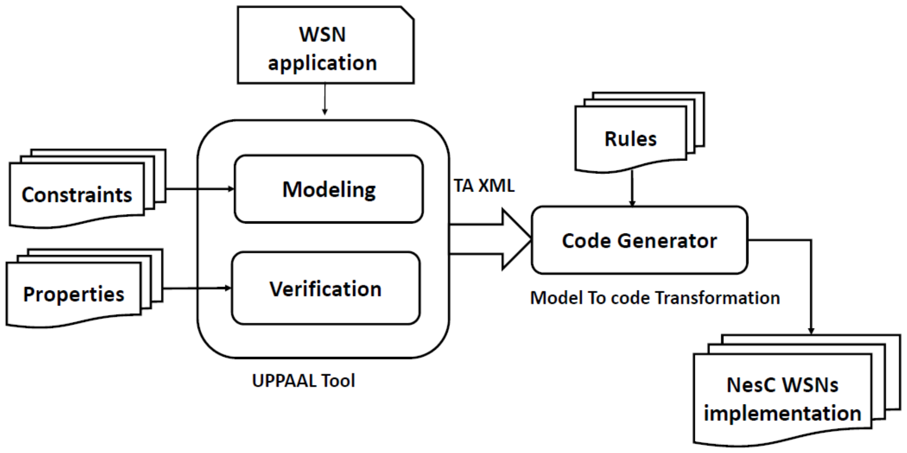


Figure 1: Approach Overview.

## 4. CODE DERIVATION

In this section, we present the basic step of our approach, which transforms high-level specifications written in TA to a NesC source code of a WSN protocol. In order to generate a source code using our tool, we apply the following informal restrictions to transform a given timed automata to a NesC code. These restrictions refer to the generic patterns that TinyOs defines (the use of hardware as Radio and LEDs, the software as the clock components, the sensing components, and the message communication). We quote these restrictions in the table 1:

Table 1: Restrictions.

| Number | Restrictions |
|---|---|
| 1 | the user must define explicitly the timed automata model used to model the environment; |
| 2 | the user must use the procedure PreparePacket() to define the packet structure; |
| 3 | the user must use the procedure LediToggle to turn on and off LEDs; |
| 4 | the user must use the procedure LediOn to turn on LEDs; |
| 5 | the user must use the procedure LediOff to turn off LEDs; |
| 6 | the user must use the procedure Ack() to enforce acknowledgements for the transmitted messages in the protocol; |
| 7 | the user must use the procedure TempRead() to sense temperature measure from the environment; |
| 8 | the user must use the procedure HumidRead() to sense humidity measure from the environment; |
| 9 | the user must use the procedure LightRead() to sense light measure from the environment; |
| 10 | the user must use the procedure VoltRead() to sense the voltage measure from the environment; |
| 11 | the user must use the variable isSucceed to check whether the previous sensing procedures are executed successfully; |
| 12 | The user must use the synchronisation channel Comm to model the sending and receiving radio messages. |

The generation of protocol implementation uses a set of rules that we have defined to map each TA element to a portion of source code. These rules are presented in the following items:

- **R1**. Each set of templates in the TA model will be used to generate a WSN application.
- **R2**. Each template is mapped to a ***module*** component and a ***configuration*** component unless if this TA is added for only simulation purposes (i.e., humans, environment, pre-exiting systems...etc.).
- **R3.** Each module component of a template uses the ***Boot*** interface and implements the initial point of each NesC program to be executed automatically by the processor, which is the "***Booted"*** event. The Boot event is signalled when the system has booted successfully.

- **R4.** Clocks in the TA trigger the use of an instance of the ***TimerC*** component. The interface ***Timer*** provided by this later is used in the module file. The application starts firstly the Timer by calling the command ***startPeriodic*** for many times or ***startOneShot*** for one single time. Such commands are often implemented in the booted event that we have specified in the previous rule. The clock variables values represent the Timer period. As a consequence of using the interface Timer, we should define the fired event, which is signalled when the Timer expires. In the configuration file, a component must be added for wiring the Timer interface to an instance of the ***TimerMilliC*** component.

- **R5.** The use of keywords: (***LediToggle (), LediOn (), and LediOff ()*** where i = 0...2 according to the manipulated LED) in TA. These keywords are mapped to: the use of the ***Leds*** interface in the module file, a call of ***the LediToggle***() (resp. ***LediOn()***, and ***LediOff()***) command, finally, the declaration of the component ***LedC*** in the configuration file, which provides the ***Leds*** interface and wires it to the module which uses this interface.

- **R6**. The use of the keyword ***Ac***k () in the TA is mapped to: the use of the ***PacketAcknowledgements*** interface in the module file. To call the ***RequestAck*** command defined in the ***PacketAcknowledgements*** interface before sending messages, to call the ***WasAcked*** command defined in the ***PacketAcknowledgements*** interface to ensure the arrival of the ***acknowledgement***, and to declare the component ***ActiveMessageC*** in configuration file, which provides the interface ***PacketAcknowledgements*** and wires it to the module which uses this interface.

- **R7.** The use of the keyword ***preparePacket(arg)*** in the TA is mapped to the: use the ***Packet*** interface in the module file, to define a structure of message in a header file, to import this later in the module, and to declare a function ***preparePacket( typedef arg)*** in the concerned module.

- **R8**. The use of the keyword ***comm*** refers to a communication, which involves : the manipulation of the radio and the use of the ***SplitContro***l interface in the module, the call to the start command for starting the radio in the booted event, and the implementation of the ***startDone()*** (resp. ***stopDone()***) events, which are defined in the ***SplitControl*** interface. In addition, the declaration of the component ***ActiveMessageC*** in the configuration file which provides the interface ***SplitControl*** and wires it to the module, which uses this interface.

- **R9**. As the communication contains two parts, the transformation will be as follows:

  - The sender part ***comm!*** is mapped to the: use of the ***AMsend*** interface, to call the function ***preparePacket(arg)***, to call the ***send()*** command in the module file, to implement the ***sendDone()*** event in the module, and to declare the component ***AMSenderC(AM RADIO)*** in the configuration file, which provides the interface ***AMSend*** and wires it to the module using this interface.

  - The receiver part ***comm?*** is mapped to the: use of the ***Receive*** interface, to implement the ***receive*** event. Then, to declare the component ***ReceiveC*** in the configuration file, which provides the interface ***Receive*** and wires it to the module, which uses this interface.

- **R10**. The use of the keyword ***TempRead ()*** in the TA are mapped to the : use the ***Read ()*** interface as ***TempRead***, to call the ***Read*** command in the module file, to implement the ***ReadDone ()*** event in the module, and to declare an instance of the component

***SensirionSht11C ()*** in configuration file. This component provides the interface temperature and wires it to the module, which uses the ***TempRead*** interface.

- **R11.** The use of the keyword ***HumidRead ()*** in the TA are mapped to: the use the ***Read ()*** interface as ***HumidRead***, to call the ***Read*** command in the module file, to implement the ***ReadDone ()*** event in the module, and to declare an instance of the component ***SensirionSht11C ()*** in configuration file. This component provides the ***humidity*** interface and wires it to the module, which uses the ***HumidRead*** interface.

- **R12**. The use of the keyword ***LightRead ()*** in the TA is mapped to: the use of the ***Read ()*** interface as ***LightRead,*** to call the ***Read*** command in the module file, to implement the ***ReadDone ()*** event in the module, and to declare an instance of the component ***HamamatsuS10871TsrC ()*** in configuration file, then wires it to the module which uses the ***LightRead*** interface.

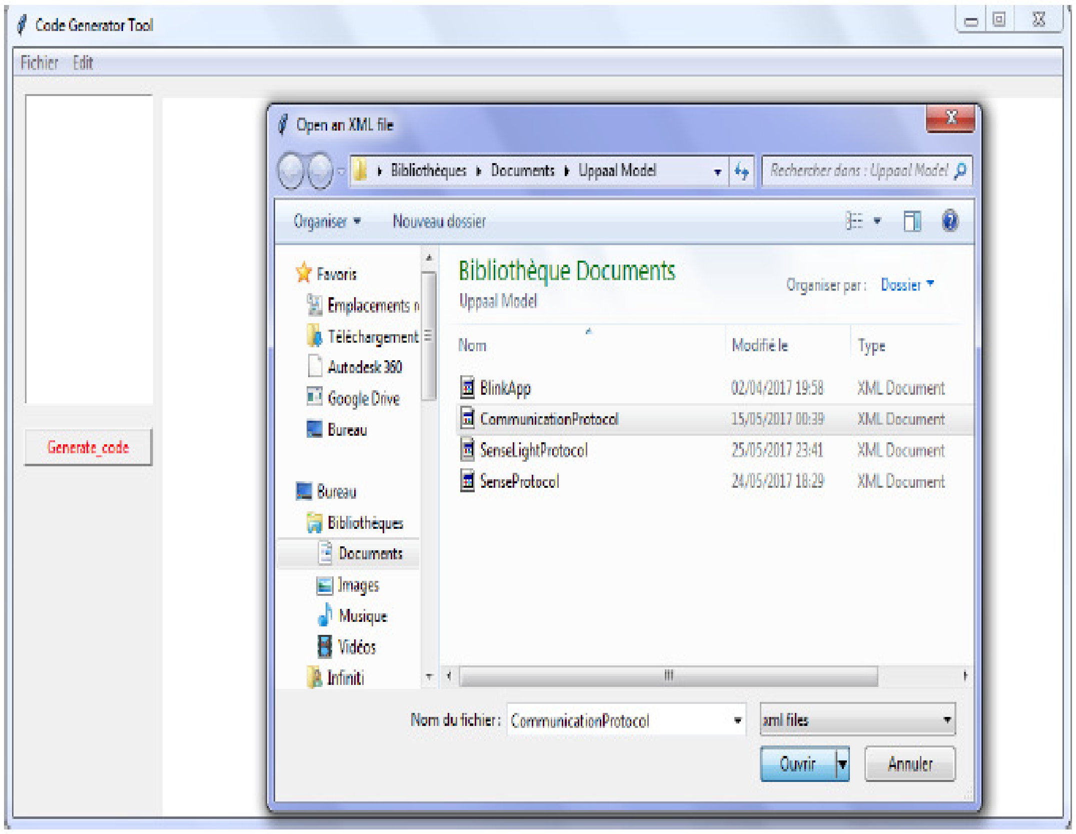
- **R13**. The use of the keyword ***VoltRead ()*** in the TA is mapped to: the use of the ***Read ()*** interface as ***VoltRead***, to call the ***Read*** command in the module file, to implement the ***ReadDone ()*** event in the module, and to declare an instance of the component ***VoltageC ()*** in configuration file, then wires it to the module which uses the ***VoltRead*** interface.

- **R14.** Variables declared in a particular template are mapped to a declaration of variables in the module, which implements this template.

- **R15.** If the guard is built upon non-clock variables then it is mapped to a standard conditional branching (an If or If-Else block) in the implementation of the module file, according to the state source of the transition. If the guard is upon clock variables then it is mapped to arguments used in the starting commands of the timer. When using the variable isSucceeded, a particular if-else block will be generated.

## 5. CODE GENERATOR TOOL

In this subsection, we present the implemented tool that enables the use of formal model for a semi-automated code generation approach. As presented in Figure.2, the transformation can be applied on a given formal specification in timed automata to nesC language. In that case, the designer can design the TA using the UPPAAL tool [4]. UPPAAL is a model checker supporting the timed automata modelling. UPPAAL is based on timed automata concepts, which are a set of clocks, channels for the synchronized systems (automata), variables and additional elements. Each automaton has an initial state. The synchronization between the different automata can take place using channels. A channel may be an output channel (denoted by channel-name!), or an input channel (denoted by channel-name?). After the design of the model, the designer can use UPPAAL also to check proprieties, verify and validate the model. When the model is validated, the designer can import it to the implemented Code Generator tool in-order to generate the corresponding code implementation (See the Figure.3).



| Nom | Modifié le | Type |
|---|---|---|
| BlinkApp | 02/04/2017 19:58 | XML Document |
| CommunicationProtocol | 15/05/2017 00:39 | XML Document |
| SenseLightProtocol | 25/05/2017 23:41 | XML Document |
| SenseProtocol | 24/05/2017 18:29 | XML Document |

Nom du fichier : CommunicationProtocol

xml files

Figure 2: Generator tool interface (Importing XML Files from UPPAAL).

The implementation of the code generator framework and the translation library are written in Python language [21]. Python is an intelligent untyped programming language that lets you work more quickly and integrate your systems more effectively. It is an open source language, and it is available for several operating systems. Firstly, we use the PyXB Python library for parsing XML files of timed automata. XML [27] is used as the standard format to describe models, formats, and data types. The modelled specification in UPPAAL is XML files that can be opened and visualized graphically as template models. Secondly, we have developed a translation from the information resulting from the parsing step to NesC source code by implementing the transformation rules. The following algorithms present the main procedures used in the implementation of our generator tool. The Algorithm 1 introduces the procedures used to map the formal model specification to NesC code. In this algorithm, we follow the same logic of steps presented in Fig. 2. First, each element in the XML file (representing the formal specification) is statically parsed to identify the set of operations and variables declaration. The functional behaviour (operations) can be identified from global declaration part of the XML file and from the template part. Then, as represented in the procedure 1 and the procedure 2, for each checked operation, we apply the corresponding rule. At the same time, the generated files are filled.

**Algorithm 1** Code Generator algorithm.

```
Data: XML file
Result:  NesC implementation
Var : balise : String
while not at end of the XML file do
    Get(file_Content)
    Read content
    if balise=="Declaration" then
        ParsingDecalaration()
    end
    if balise=="Template" then
        Aply_rules(R2&R3)
        ParsingTemplate()
    end
    Apply_rule(02)
end
```

**Procedure 1** ParsingDeclaration.

```
Data: Declaration content
Result:  source Code
Var : chan_nam,variable_type, text : String ;
while text not null do
    Read content
    if variable_type= int then
        Add( file.h , "uint8_t " , var_name )
    end
    if variable_type= String then
        Add( file.h , "String " , var_name )
    end
    if variable_type= chan or variable_type= broadcast chan then
        if chan_name= comm then
            Apply_rule(R8&R9)
        end
        if chan_name= comm? then
            Apply_rule(R8&R9)
        end
    end
end
```

**Procedure2** ParsingTemplate.

```
Var : balise_child, text : String ;
while not at end of the template do
    text== Get(Template_Content)
    if (text not null) then
        Apply_rule(R13)  if ( balise_child == Declaration ) then
            if (( variable_type = int ) and ( variable_name= isSucceed )) then
                Apply_rule(R14) ;
            end
            if ( variable_type= clock ) then
                Apply_rule(R4) ;
            end
            if (declared_procedure ) then
                if ( declared_procedure= preparePacket( )) then
                    Apply_rule(R7);
                end
                if ( declared_procedure = Toggle( ) )or (declared procedure=
                 lediOn()) or (declared procedure= lediOff()) then
                    Apply_rule(R5) ;
                end
                if  (declared_procedure=  TempRead  (  ))  or  (
                declared_procedure= HumidRead ( )) then
                    Apply_rule(R10) ;
                end
                if (declared_procedure= LightRead( )) then
                    Apply_rule(R11) ;
                end
                if (declared_procedure= VoltRead( )) then
                    Apply_rule(R12) ;
                end
            end
        end
        if (balise_child == Transition) then
            Kind==Get_Label_Kind()  if (Kind=="Guard") then
                Check_Source_transition()
                Apply_rule(R14)
            end
        end
    end
end
```

## 6. Application On A Case Study

Our example is an illustrative system that offers to the user the opportunity to measure the humidity and the temperature from the environment. This system presents an Air temperature and Humidity monitoring used in temperature-controlled rooms. This system contains all the elements of sensor network application such as communication (sending and receiving), sensing, and processing packets. The system describes the communication between two Telosb nodes in a sensor network. The system uses two clocks. An alarm system should be linked to the monitoring system, which presented in our case by the use of two LEDs. The system manages the sending and receiving cycle using two clocks. The two nodes are specified as Sense_Sender node, which has two types of sensing (temperature and humidity), and the receiver node. The communication between these two nodes is based on the temperature, and humidity measurements. Each time c1 (2000 milliseconds), the Sense_Sender node detects the humidity and the temperature values from the environment and uses these values to calculate the humidity (resp. the temperature). If any of these values are upper a certain threshold, the corresponding LED turns on. After 4000 milliseconds for the second clock, the receiver board will light up.

## 6.1. Timed Automata Models of the System

We modelled the example with three timed automata. The first and second automaton represent the behaviour of the temperature (resp. humidity) sensing operations as shown in the Figure 4 (resp. 5). The third TA depicted in Figure 6 represents the receiver.

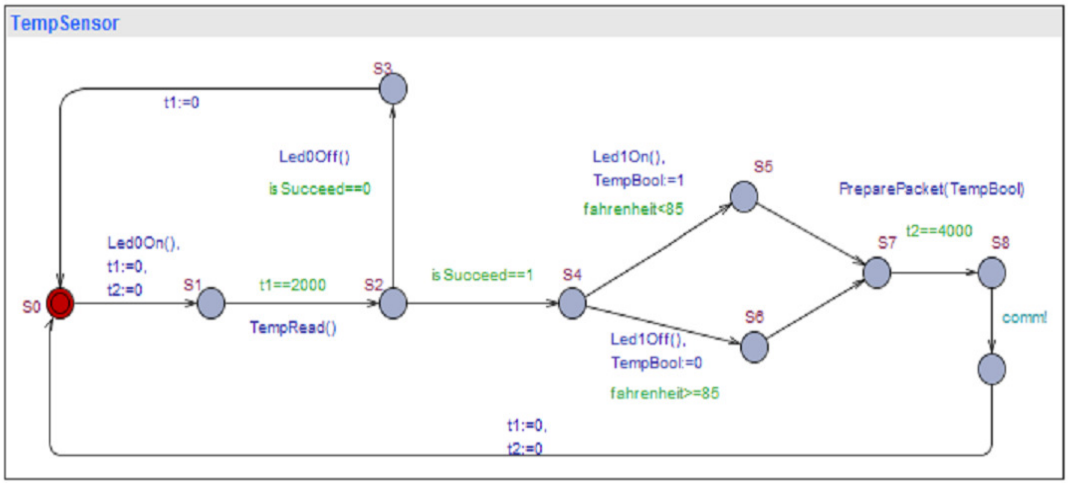


Figure 3: TA Model for Temperature Sensing.

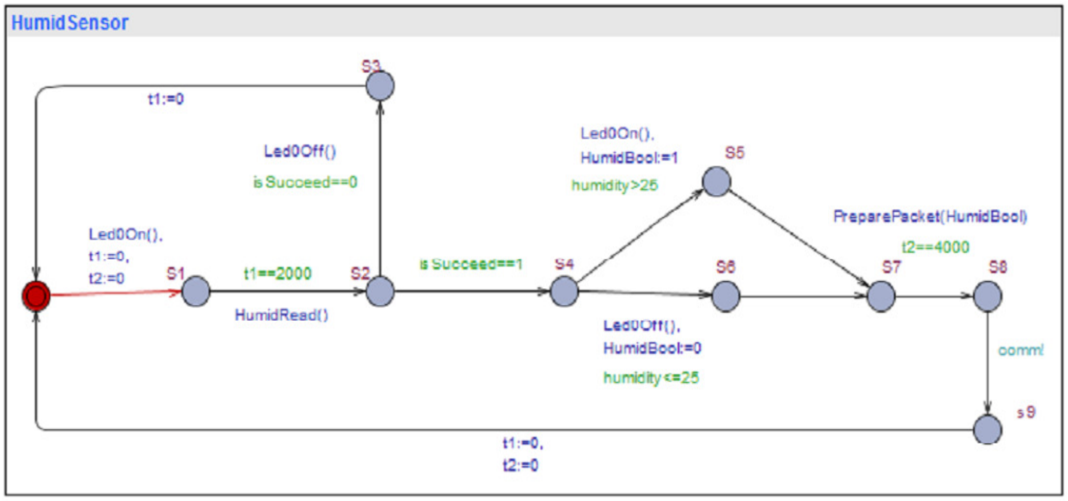


Figure 4: TA Model for Humidity Sensing.

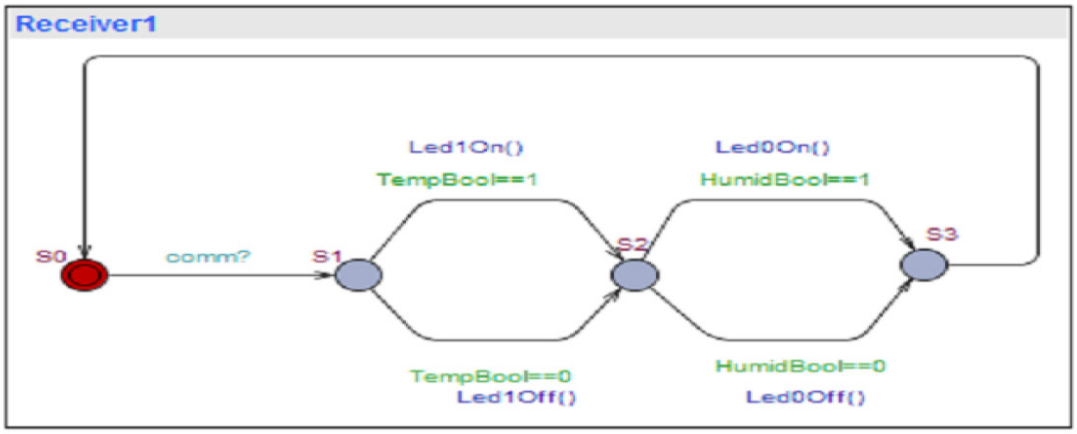


Figure 5: TA Model for the Receiver part.

### 6.2. Timed Automata Models of the System

In this subsection, we present the principal files, which contain the implementation of the given model. The code presented in the following listings (see listings 1, 2) show the generated code by our code generator. Due to space reason, we have explained a part of the generated code from the model presented in 6.1. The Figure.4 presents the timed automata of the temperature sensing. Firstly, the code generator applies the rules R1 and R2 to create a module and a configuration files. Then, it applies the rule R3 to implement the first event to start the program execution. The figure.3 shows a started transition (S0, S1) labelled by two resets (t1=:0, t2:=0) of the clocks t1, t2 and one procedure "Led0On()". During the parsing of the model, the appearance of a clock in a guard or a reset on a transition triggers the application of the rule R4 that consists to a call for the Timer component. The procedure call "Led0On()" on the transition (S0,S1) is mapped to the use of the LEDs component in the code, using the rule R5. The listings 1, 2 present the module and the configuration files. The transition (S1, S2) is labelled by the guard (t1==2000) and the procedure call (TempRead()). During the parsing of the model, the guard upon the clock t1 is used to define the argument value used in a starting time command as mentioned in the rule R14. The call of the procedure TempRead() is mapped to the use of the "temperature sensing component" as described in the rule R10. In addition, if the procedure TempRead defines and uses variables, the code generator applies the rule R13. The following listings 1 and 2 show the generated code corresponding to the part of the model linking state S0 to state S2. We put the complete generated code in the appendix A.

```
configuration SenseC{
}
implementation{
       components MainC, LedsC;
       components TempM;
       components new TimerMilliC() as T1;
       components new TimerMilliC() as T2;
       TempM.Boot -> MainC;
       TempM.Leds -> LedsC;
       TempM.T1 -> T1;
       TempM.T2 -> T2;
}
```

Listing 1: Skeleton of the configuration file.

```
#include "packets.h"
// application of the rule R1
module TempM{
        uses {
                // application of the rule R3
                interface Boot;
                // application of the rule R5
                interface Leds;
                // application of the rule R4
                interface Timer<TMilli> as T1;
                interface Timer<TMilli> as T2;
        }
}
implementation{
        // application of the rule R3
        event void Boot.booted(){
                // application of the rule R4
                call T1.startPeriodic(...);
                call T2.startPeriodic(...);
                // application of the rule R5
                Leds.led0On;
        }
   // application of the rule R4
        event void T1.fired(){
                              }

        event void T2.fired(){

                              }
}
```

Listing 2: Skeleton of the module file.

## 7. CONCLUSION AND PERSPECTIVES

Wireless sensor networks are of considerable interest and a new stage in the evolution of information and communication technologies. This new technology attracts increasing interest because of the diversity of its applications: health, environment, industry and even sports. In the current paper, an automated transformation approach was presented which intended to generate source code of specific target language (NesC programming language) from high-level formal specification (Timed Automata model). This approach is divided into two parts: the first part is dedicated to the modelling of the WSNs system respecting specific constraints model, and a second part dedicated to code generation. In fact, the proposed approach is intuitive, it defines a set of transformation rules that a generator code takes as input as well as the XML description of a TA specification, and then it generates the source code. In order to automate this transformation, we have implemented these rules in a framework. In addition, we have demonstrated the application of the approach on a case study of air monitoring system. During the realisation of the current work, we have faced several difficulties in the definition of rules. This is due to the high level abstraction of TA whereas NesC is closer to hardware devices. Therefore, it is not evident to extract enough information from TA, that help in the implementation of a WSN system. The current work has defined a set of not complete rules. These rules can be used to generate code for several case studies. The work will be improved by completing the set of rules that can be used to generate code for more complex formal models and to consider other aspect of the formal model (i.e., probabilistic parameters).

## REFERENCES

[1] Yick, J., Mukherjee, B., & Ghosal, D. (2008). Wireless sensor network survey. *Computer networks*, *52*(12), 2292-2330.

[2] Levis, P., & Gay, D. (2009). *TinyOS programming*. Cambridge University Press.

[3] Bengtsson, J., & Yi, W. (2004). Timed automata: Semantics, algorithms and tools. *Lecture Notes in Computer Science*, *3098*, 87-124.

[4] J. B. Gerd Behrmann, Tobias Amnell, (2015). "UPPAAL", URL: http://www.uppaal.org/. [accessed: 2017-05-08].

[5] Losilla, F., Vicente-Chicote, C., Álvarez, B., Iborra, A., & Sánchez, P. (2007, September). Wireless sensor network application development: An architecture-centric mde approach. In European Conference on Software Architecture (pp. 179-194). Springer, Berlin, Heidelberg.

[6] D. Demange, (2006), "Domain specific language", URL: http://igm.univmlv.fr/~dr/XPOSE2006 /LOPDEMANGE/dsl.html [Accessed: 2017-07-14].

[7] O. M. Group, (2017), "Unified modeling language", URL: http://www.uml.org/ [Accessed: 2017-03-15].

[8] Shimizu, R., Tei, K., Fukazawa, Y., & Honiden, S. (2011, May). Model driven development for rapid prototyping and optimization of wireless sensor network applications. In Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications (pp. 31-36). ACM.

[9] S. I. J. R. B. S. Grady Booch, Alan Brown, (2004), "Domain-specific modelling", URL: http:// www.dsmforum.org/ [accessed: 2017-05-15].

[10] Madden, S. R., Franklin, M. J., Heller stein, J. M., & Hong, W. (2005). TinyDB: an acquisitional query processing system for sensor networks. ACM Transactions on database systems (TODS), 30(1), 122-173.

[11] Somappa, A. A. K., Øvsthus, K., & Kristensen, L. M. (2014). Towards a dual-mode adaptive MAC protocol (DMA-MAC) for feedback-based networked control systems. Procedia Computer Science, 34, 505-510.

[12] Somappa, A. A. K., Prinz, A., & Kristensen, L. M. (2015). Model-Based Verification of the DMAMAC Protocol for Real-time Process Control. In VECoS (pp. 81-96).

[13] Somappa, A. A. K., Øvsthus, K., & Kristensen, L. M. (2016). Implementation and Deployment Evaluation of the DMAMAC Protocol for Wireless Sensor Actuator Networks. Procedia Computer Science, 83, 329-336.

[14] Simonsen, K. I. F., Kristensen, L. M., & Kindler, E. (2013, September). Generating protocol software from cpn models annotated with pragmatics. In Brazilian Symposium on Formal Methods (pp. 227-242). Springer, Berlin, Heidelberg.

[15] Simonsen, K. I. F., Kristensen, L. M., & Kindler, E. (2016). Pragmatics annotated coloured petri nets for protocol software generation and verification. In Transactions on Petri Nets and Other Models of Concurrency XI (pp. 1-27). Springer Berlin Heidelberg.

[16] Simonsen, K. I. F. (2013, September). PetriCode: a tool for template-based code generation from CPN models. In International Conference on Software Engineering and Formal Methods (pp. 151-163). Springer, Cham.

[17] Simonsen, K. I. F., & Kristensen, L. M. (2014, June). Implementing the websocket protocol based on formal modelling and automated code generation. In IFIP International Conference on Distributed Applications and Interoperable Systems (pp. 104-118). Springer, Berlin, Heidelberg.

[18] The Apache Groovy project, (2003-2017), "Apache groovy". [Online]. Available: http: //www. groovy-lang.org/

[19] Oracle, (2003-2017), "Java", URL : https://www.java.com/fr/ [Accessed : 2017-07-20].

[20] R. Hickey, (2017), “The clojure programming language”, URL: https://clojure.org/ [accessed: 2017-02-20].

[21] Python, J. (2009). Python (programming language). Python (programming Language) 1 CPython 13 Python Software Foundation 15, 1.

[22] Kumar, S. A., & Simonsen, K. I. (2014, April). Towards a model-based development approach for wireless sensor-actuator network protocols. In Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems (pp. 35-39). ACM.

[23] A. Viklund, (2011-2017), “Mixim”, URL: http://mixim.sourceforge.net/ [accessed: 2017-07-20].

[24] Somappa, A. A. K., & Simonsen, K. I. F. (2016). Model-based Development for MAC Protocols in Industrial Wireless Sensor Networks. In PNSE@ Petri Nets (pp. 193-212).

[25] Kristensen, L. M., & Veiset, V. (2016, June). Transforming CPN Models into Code for TinyOS: A Case Study of the RPL Protocol. In International Conference on Applications and Theory of Petri Nets and Concurrency (pp. 135-154). Springer International Publishing.

[26] Levis, P., Lee, N., Welsh, M., & Culler, D. (2003, November). TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In Proceedings of the 1st international conference on Embedded networked sensor systems (pp. 126-137). ACM.

[27] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). World Wide Web Journal, 2(4):27–66, 1997.

## APPENDIX A

```
#ifndef STRUCT_MSG_H
#define STRUCT_MSG_H
   typedef nx_struct my_msg_t {
       nx_uint16_t msg_type;
       nx_uint16_t msg_id;
       typedef msg_value;
       } my_msg_t;
#define REQ 1
#define RESP 2
   enum { AM_RADIO =6 };
#endif
```

Listing 3: Skeleton of the Packet Structure.

```
configuration SenseC{
}
implementation{
        components MainC, LedsC;
        components TempM;
        components ActiveMessageC;
        components new AMSenderC(AM_RADIO);
        components new TimerMilliC() as TempTimer;
        components new TimerMilliC() as NetworkTimer;
        components SerialPrintfC;
        components new SensirionSht11C() as TempRead;
        TempM.Boot -> MainC;
        TempM.Leds -> LedsC;
        TempM.T1 -> TempTimer;
        TempM.T2 -> NetworkTimer;
        TempM.TempRead -> TempRead.Temperature;
        TempM.Packet -> AMSenderC;
        TempM.AMSend -> AMSenderC;
        TempM.AMControl -> ActiveMessageC;
}
```

Listing 4: Skeleton of the Temperature Configuration

```
#include "packets.h"
module TempM{
        uses {
                interface Boot;
                interface Leds;
                interface Timer<TMilli> as T1;
                interface Timer<TMilli> as T2;
                interface Read<uint16_t> as TempRead;
                interface Packet;
                interface AMSend;
                interface SplitControl;
        }
}
implementation{
        uint16_t centigrade, fahrenheit;
        uint8_t tempBool;
        message_t _packet;

        event void Boot.booted(){
                call T1.startPeriodic(2000);
                call T2.startPeriodic(4000);
                call SplitControl.start();
                Leds.led0On;
        }
        event void SplitControl.startDone(error_t error) {
                if (error == SUCCESS) {
                        call Leds.led0On();
                }
                else {
                        call SplitControl.start();
                }
        }
        event void SplitControl.stopDone(error_t error) {
                               }
        event void T1.fired(){
                if (!(call TempRead.read() == SUCCESS))
                   call Leds.led0Off();
        }
        //construct the packet:
        void PreparePacket(uint16_t val){
        //obtain the packet pointer
            my_msg_t* msg = call Packet.getPayload(& _packet, sizeof(my_msg_t));
        // Specify the sequence number of the message
                msg->msg_id= TOS_NODE_ID;
        //affect the sensed information to the message value
                msg->msg_value=tempBool;
        }
        event void T2.fired(){
           PreparePacket(tempBool);
           call AMSend.send(AM_BROADCAST_ADDR, &_packet, sizeof(my_msg_t));
                                       }
        event void AMSend.sendDone(message_t *msg, error_t error) {
        ...
        }
        event void TempRead.readDone(error_t result, uint16_t val){
                        centigrade = -39.6 + .01*val;
                        fahrenheit = (9/5)*centigrade + 32;
                        if (fahrenheit > 85){
                                call Leds.led1On();
                                tempBool = 1;
                        } else {
                                call Leds.led1Off();
                                tempBool = 0;
                        }}}
```

Listing 5: Skeleton of Temperature Sensing Module

```
#include "MsgStruct.h"
module ReceiveC{
  uses {
       interface Boot;
       interface Leds;
       interface Packet;
       interface AMPacket;
       interface SplitControl as AMControl;
       interface Receive;
       }
}
implementation{
  uint8_t tempBool;
  uint8_t humidBool;
  event void Boot.booted(){
       call SplitControl.start();
       }
  event void SplitControl.startDone(error_t error) {
       if (!(error == SUCCESS))
              call SplitControl.start();
    }

  event void SplitControl.stopDone(error_t error) {
       }
  event message_t * Receive.receive(message_t *msg,void *payload,uint8_t
len) {
              if (len == sizeof(my_msg_t)) {
                     my_msg_t * incomingpacket = (my_msg_t*) payload;
                     tempBool = incomingpacket->tempBool;
                     humidBool = incomingpacket->humidBool;
                     if (humidBool == 1) {
                            call Leds.led0On();
                            }
                     else {
                            call Leds.led0Off();
                            }
                     if (tempBool == 1) {
                            call Leds.led1On();
                     }
                     else {
                            call Leds.led1Off();
                            }
              }
              return msg;
       }
}
```

Listing 6: Skeleton of Receiver Module.

```
#include "packets.h"

module HumidityM{
        uses {
                interface Boot;
                interface Leds;
                interface Timer<TMilli> as T1;
                interface Timer<TMilli> as T2;
                interface Read<uint16_t> as HumidRead;
                interface Packet;
                interface AMSend;
                interface SplitControl;
        }           }
implementation{
        //variables declaration
        uint16_t humidity;
    uint8_t HumidBool;
        message_t _packet;
        event void Boot.booted(){
                call T1.startPeriodic(2000);
                call T2.startPeriodic(4000);
                call SplitControl.start();
                Leds.led0On;
        }
        event void SplitControl.startDone(error_t error) {
                if (error == SUCCESS) {
                        call Leds.led0On();
                }
                else {
                        call SplitControl.start();
                }}
        event void SplitControl.stopDone(error_t error) {
        }
        //start the system, radio..
        event void TempTimer.fired(){
                if (!(call HumidRead.read() == SUCCESS))
                        call Leds.led0Off();
        }
         //construct the packet:
        void PreparePacket(uint16_t val){
        //obtain the packet pointer
            my_msg_t* msg = call Packet.getPayload(& _packet, sizeof(my_msg_t));
        // Specify the sequence number of the message
                msg->msg_id= TOS_NODE_ID;
        //affect the sensed information to the message value
                msg->msg_value=HumidBool;
        }
        event void T2.fired(){
        //prepare packet
                PreparePacket(HumidBool);
        //send the packet
       call AMSend.send(AM_BROADCAST_ADDR, &_packet, sizeof(my_msg_t));
        }
        ......
        event void HumidRead.readDone(error_t result, uint16_t val){
           humidity = -4.0 + 0.0405*val + (-2.8 * pow(10.0,-6))*(pow(val,2));
                        if (humidity > 25){
                                call Leds.led0On();
                                humidBool = 1;
                        } else {
                                call Leds.led0Off();
                                humidBool = 0;
                        } }

}
```

Listing 7: Skeleton of Humidity Sensor.

```
configuration SenseC{
}
implementation{
       components MainC, LedsC;
       components TempM;
       components ActiveMessageC;
       components new AMSenderC(AM_RADIO);
       components new TimerMilliC() as TempTimer;
       components new TimerMilliC() as NetworkTimer;
       components SerialPrintfC;
       components new SensirionSht11C() as HumidRead;

       TempM.Boot -> MainC;
       TempM.Leds -> LedsC;
       TempM.T1 -> TempTimer;
       TempM.T2 -> NetworkTimer;
       TempM.HumidRead ->  HumidRead.Humidity;
       TempM.Packet -> AMSenderC;
       TempM.AMSend -> AMSenderC;
       TempM.AMControl -> ActiveMessageC;
}
```

Listing 8 : Skeleton of Humidity Configuration