Model-driven round-trip engineering for TinyOS-based WSN applications

Model-driven Round-trip Engineering for TinyOS-based WSN Applications

Hussein Marah[1,2], Geylani Kardas*[2], Moharram Challenger[3,4]

[1] *Johannes Kepler University Linz, Austria*
[2] *International Computer Institute, Ege University, Turkey*
[3] *Department of Computer Science, University of Antwerp, Belgium*
[4] *AnSyMo/CoSys Core Lab, Flanders Make Research Center, Flanders, Belgium*
[1]hussein.marah@jku.at
[2]geylani.kardas@ege.edu.tr
[3]moharram.challenger@uantwerpen.be

Abstract

Wireless Sensor Network (WSN) applications working on TinyOS operating system is widely used in various areas. However, the requirement of managing the power constraints makes TinyOS different from ordinary systems and hence building WSNs with TinyOS can be a challenging and time-consuming task. As successfully applied in many other domains, model-driven engineering (MDE) can facilitate the design and implementation of such applications. Within this context, the researchers have performed noteworthy studies on deriving various MDE approaches and tools. However, these studies do not support the synchronization between TinyOS application models and the generated programs especially when any change is made in the programs. Hence, in this paper, we introduce a model-driven round-trip engineering (RTE) methodology in which both the MDE of TinyOS applications and synchronization between TinyOS instance models and corresponding code are provided with the use of a toolchain. A domain-specific modeling language, called DSML4TinyOS is used for the MDE of such applications while existing TinyOS programs can be reverse engineered in an environment, called RE4TinyOS. Models retrieved from the existing programs can be automatically processed again with DSML4TinyOS. Evaluation results showed that it is possible to obtain the configurations of the TinyOS applications completely just modeling with DSML4TinyOS whereas the same process leads the automatic creation of almost half of the module parts of these applications. Moreover, the time required for developing such systems from scratch decreased approximately to its half. The results also showed that both model-code synchronization and the integration of existing TinyOS applications which do not have system models previously, into the proposed MDE are possible with using RE4TinyOS. RE4TinyOS succeeded in the reverse engineering of all main parts of the TinyOS applications taken from the official TinyOS Github repository and generated models were able to be visually processed in the MDE environment for further modifications.

Keywords: Model-driven Engineering, Round-trip Engineering, Wireless Sensor Networks, TinyOS, DSML4TinyOS, RE4TinyOS

*Corresponding author

1 Introduction

Due to having low-power micro-controllers and spatially dispersed sensors, Wireless Sensor Networks (WSNs) [1] are used in many application domains (e.g. healthcare [2], field monitoring [3], transportation [4], military [5] and smart buildings [6]) to control both the status of physical objects and the surrounding circumstances like sound, pressure, vibration, light, temperature, and motion. Moreover, WSNs also enable the acquisition of long-term industrial environmental data for the Internet of Things (IoT) applications [7].

One of the widely used operating systems for WSNs is TinyOS [8]. TinyOS is an open-source operating system for WSNs, developed in the University of California, Berkeley. It is a lightweight and flexible operating system that offers a set of services such as communication, timers, sensing, storage and these services can be reusable to compose larger applications. These features make TinyOS a reliable and efficient system for programming, configuring and running low-power wireless devices [8][9]. However, especially the requirement of managing the power constraints makes TinyOS different from ordinary systems and hence building WSNs with TinyOS can be a challenging and time-consuming task. Moreover, the developers need to have deep knowledge and skills in the special programming language of TinyOS, called nesC to implement such systems [9]. Adoption to this language may be difficult and again time-consuming for the programmers.

As successfully applied in many other domains (e.g. [10, 11, 12, 13]), model-driven engineering, (MDE), mostly supported with the use of domain-specific languages (DSLs) [14, 15] or domain-specific modeling languages (DSMLs) [16], can also provide a convenient way of developing TinyOS applications for WSNs by leveraging the abstraction level before delving into programming with nesC and hence help minimizing the abovementioned difficulties of WSN development. Within this context, there are many studies taking into account the use of MDE (e.g. [17, 18, 19, 20, 21, 22, 23]) and / or proposing new DSLs / DSMLs (e.g. [24, 25, 26, 27]) for WSN and TinyOS application development. However, these studies do not support the synchronization between TinyOS application models and the generated nesC programs especially when any change is made in the programs. Mostly, following the code generation from the models, the auto-generated code is modified to completely meet with the requirements of the TinyOS applications. Furthermore, the applications may naturally evolve according to the changing requirements in the future. After the code modifications are performed, related changes will make models at different levels asynchronous and inconsistent [28]. Thus we need to propagate these changes to the models and ensure a proper model-code synchronization [29] for TinyOS application models. Hence, in this paper, we introduce a model-driven round-trip engineering (RTE) methodology in which both the MDE of TinyOS applications and synchronization between TinyOS instance models and corresponding code are provided with the use of a toolchain.

As depicted in Figure 1, evolution of the TinyOS models can be managed within the proposed model-driven RTE process which is a combination of the forward and reverse engineering of TinyOS models. TinyOS models can be designed and created with using a DSML called DSML4TinyOS and the corresponding TinyOS code can be automatically generated. When this code is modified and becomes TinyOS code', this modified code can be processed in a reverse engineering platform, called RE4TinyOS, to retrieve the corresponding modified model (still an instance of TinyOS metamodel) which properly reflects the changes in the application code. RE4TinyOS enables retrieving TinyOS application models from any existing nesC code. In addition to support the reverse engineering of such applications, use of RE4TinyOS also integrates with the current MDE process brought by the DSML4TinyOS

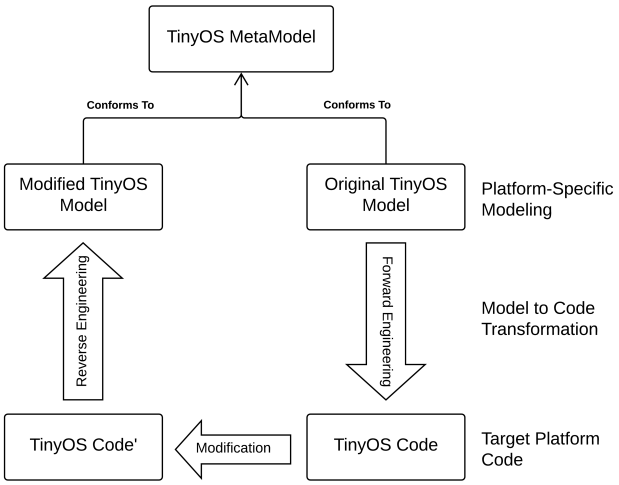

Figure 1: Forward and reverse engineering for TinyOS applications

language to construct a complete model-driven RTE [30] process for TinyOS applications.

This study is an extended version of our previous study introduced in [31]. It differs from the latter by including: 1) a complete model-driven RTE methodology composed of forward engineering and reverse engineering steps, 2) an improved case study of using this new RTE methodology showing how TinyOS applications can be both modeled and generated as well as modifications made in the TinyOS programs can be automatically reflected to the models, 3) an evaluation of using the proposed RTE toolchain in the development of eight different TinyOS applications with changing complexities. In this evaluation, application generation and development time performances of using the proposed DSML are measured and the analysis of the applied reverse engineering process is assessed.

The remainder of the paper is organized as follows: Section 2 gives the related work in this area. Proposed model-driven RTE methodology is discussed in Section 3 with all its steps and the tools. The usability of this methodology is demonstrated with a case study in Section 4. Quantitative analysis and qualitative assessment of using the RTE toolchain are discussed in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

In recent years, there is a significant interest of the researchers to apply MDE and its techniques for WSN and IoT development (e.g. [24, 20, 21, 26, 23, 27]). In these studies, the main goal of applying MDE approach is to facilitate the task of developing, building and deploying different WSN and IoT applications [19, 32]. Various recent surveys and reviews (e.g. [19, 33, 13, 34, 35, 36]) also discuss the wide use of MDE for WSN and IoT as well as its applicability e.g. with visual programming environments.

Vicente-Chicote et al. [17] define three levels of abstraction which allow designers to build domain-specific models, component-based architecture descriptions and platform-specific models for the WSN application development. Automatic model transformations are implemented between these abstraction levels for the MDE of such WSN applications. Baobab

metamodeling framework, introduced in [37], enables defining functional and non-functional aspects of a WSN separately as software models, validates them and generates code automatically within an MDE perspective. Similarly, ScatterClipse, a generative plugin-oriented tool-chain, is proposed in [38] to develop WSN applications running on the ScatterWeb sensor boards by using MDE. The tool aims to automate and standardize the generation of application system families for these sensor boards.

Mozumdar et al. [39] propose a system development framework based on Simulink to design and implement sensor network components at both protocol and application layers. Following modeling and simulation steps, it is possible to generate application code from the simulated models for different target operation systems. Thang and Geihs [40] address the problem of optimizing power consumption and memory usage in the application design process and introduces an approach that integrates Evolutionary Algorithms with MDE where the system metamodels are generated to select the optimal model according to some performance criteria. Another modeling framework [18] allows developers to model separately the WSN software architecture and the features of the low-level hardware as well as the physical environment of the nodes of a WSN. The framework is capable of generating code from the created models which can be used for specific purposes such as analysis.

The study in [41] brings an MDE approach for prototyping and optimization of WSN applications while Veiset and Kristensen [42] introduce the use of Coloured Petri Net models for generating TinyOS protocol software. Likewise, the use of a DSL, called SenNet, for WSN application development is proposed in [25, 43] to prepare WSN applications using multi-abstraction levels. Rodrigues et al. first introduce a layered WSN application development approach [44] to meet the system development requirements of both novice and experienced programmers and then they provide an improved version of their development model [22] which aims at facilitating the development tasks required for Wireless Sensor and Actuator Network (WSAN) applications inside an MDA-based process. The proposed infrastructure is composed of a platform-independent model (PIM), a platform-specific model (PSM), and a transformation process which allows modeling and generation of these applications.

Another DSL, called LWiSSy, is introduced in [24] to model WSAN systems based on the three dimensions, namely developer profile, specification granularity and design. The developer profile considers domain or network experts while specification granularity ranges from network to single node programming. Finally, design dimension is specified with using a metamodel. Code generation is provided again according to MDA specifications. Tei et al. [20] define an MDD-based stepwise software development process for WSNs by both enabling separation of concerns and bringing reusable solutions for network-related issues. Based on their evaluation, this new process improves WSN software development by significantly reducing the modeling steps for node-level and group-level processes. Taking into consideration the event-driven mechanism and protothread architecture of Contiki operating system, Durmaz et al. [21] introduce a metamodel for IoTs. This metamodel is extended in [23] to create a modeling environment which provides generating programs for Contiki-based IoT systems. The use of this new IoT modeling environment is exemplified for the implementation of a smart fire detection system.

Barricelli and Valtolina [26] investigate the use of a visual language's elements to indirectly manage physical devices and their data streams without the need to know technical specification of the devices, the applications, and the data. For this purpose, a rule language is developed for the eWellness domain which allows coaches and trainers to express complex rules and temporal constraints for team sports through a visual interface. Similarly, Smart Block [27] benefits from the features of the visual block programming to create IoT appli-

cations. Originating from IoT automation, Smart Block supports writing IoT applications in the event-condition-action (ECA) style which is implemented with Google Blockly. The use of this language also enables checking the redundancy, inconsistency, and circularity in the ECA rules before generating code for the application.

The above mentioned studies provide various noteworthy approaches both for modeling WSN and IoT applications in different abstraction levels and code generation for these applications, mostly assisted with tools. Moreover, some of them specifically support the development of TinyOS applications within the MDE perspective. However, to the best of our knowledge, none of them considers the reflection of changes made after in the generated code to the corresponding application models, i.e. an approach for constructing the synchronization between WSN/IoT model and code does not exist. In addition, with a few exceptions (e.g. [24, 20, 27]) the evaluation of the proposed MDE approaches in these studies is mostly limited with single case studies showing how these approaches can be applied. We believe that the methodology, introduced in this paper, may contribute to these efforts by filling this gap as well as supporting the round-trip engineering of TinyOS WSN applications within a toolchain consists of both generating code from TinyOS application models and retrieving models from the existing code automatically.

Taking into consideration of applying reverse engineering and/or round-trip engineering in the context of MDE, various adoptions exist for different domains as surveyed in [45]. Perhaps one of the most popular approaches is MoDisco [46], which follows the MDE concepts and techniques to represent the legacy software systems in a different formalism by using reverse engineering. The infrastructure of MoDisco introduces generic components that can be used in the model-driven reverse engineering process (e.g., generic metamodels, model navigation, model transformation and model customization). Favre et al. [47] describe an operation for generating MDA models that combines the process of static and dynamic analysis. Model recovery is illustrated with the reverse engineering of Java code to get class and state diagrams. Fruitful applications of model-driven reverse engineering can also be seen in e.g. transforming legacy COBOL code into models [48], model discovery from Java source code to extract the business rules [49], generating GUI models of the explicit layouts especially for Java Swing user interfaces [50], restoring extended entity-relationship schema from NoSQL property graph databases [51] and even achieving reusable and evolvable model transformations [52]. Also, another different approach is presented in [53] to achieve model synchronization from model transformations implemented by using Atlas Transformation Language (ATL). With a unidirectional transformation between metamodels, the approach can automatically synchronize models in the metamodels by disseminating changes over these models.

The study conducted in [54] tries to tackle the issues when developing control algorithms for mechatronic by proposing a round-trip engineering approach which allows a semi-automatic integration of hardware properties into the control model. Another study [55] aims at applying a round-trip engineering process for NoSQL database systems. The developed framework is based on MDE and combines the forward and reverse engineering to enrich the created models. A programming toolkit for native Java applications based on the round-trip engineering is presented in [56]. This toolkit provides a visualization to the source code so the code can be edited and changes can be reflected to the design model. Similarly, Buchmann et al. [57], provides a round-trip engineering tool which supports a bidirectional transformation between class diagrams and Java source code. However, round-trip engineering of WSN and TinyOS applications is not addressed again in all these studies.

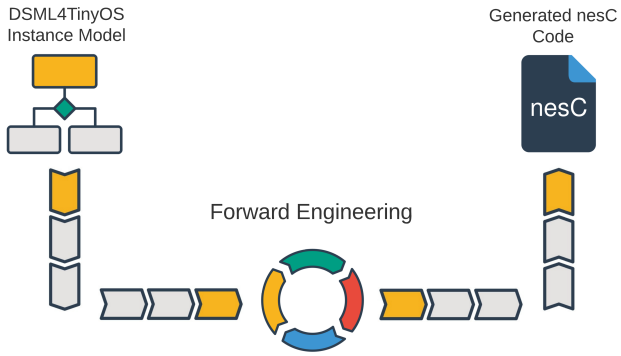

Figure 2: Overview of the proposed forward engineering approach

3 Round-trip Engineering Methodology

In this section, our RTE methodology for developing TinyOS applications is discussed. In many MDE processes, model transformations are applied between source and target models, usually from higher to lower abstraction levels. Changes on the artifacts achieved from the transformation processes are inevitable, so, it is important to preserve the coherency of the whole system and keep the source and target consistent [28]. Thus, the role of RTE engineering is to provide a bidirectional synchronization mechanism which embodies a path where a bidirectional synchronization is provided across the models. RTE also enables moving forward and backward inside the whole process without losing the trace [58].

As we stated before, there are two main parts of our RTE methodology: the first part covers the MDE of TinyOS applications in accordance with MDA by applying a forward engineering (FE) process. In this part, TinyOS applications are modeled and the corresponding code is generated using the DSML4TinyOS modeling environment. The second part of the RTE allows software models to be recovered from an existing TinyOS application using the reverse engineering (RE) features brought by the RE4TinyOS environment. Changes made in an existing TinyOS application are automatically reflected into the model of this application hence the model-code synchronization is provided. These two parts of the methodology can be operated in a lifecycle, making model-driven RTE possible for TinyOS applications. Following subsections discuss these parts of the methodology.

3.1 Forward Engineering with DSML4TinyOS

In the proposed methodology, model-driven development of TinyOS applications is provided using a DSML, called DSML4TinyOS. DSML4TinyOS language is supported with an IDE in which the graphical syntax of the language is used for creating TinyOS application models and then the corresponding code is generated from these models by automatically applying the translational semantics of the language, which is a well-known and widely applied technique in MDE (e.g. [59, 60]). As illustrated in Figure 2, this top-down software development process leads to the FE of TinyOS applications in which an automatic transformation from system models at a higher abstraction level to TinyOS (nesC) code at a more concrete level is possible by using only DSML4TinyOS.

DSML4TinyOS is a tool-supported DSML which facilitates the development of TinyOS

applications according to MDE principles and techniques. Its tool enables TinyOS developers to develop applications from scratch by visually modelling these applications and generate code as the final artefact. The abstract syntax of DSML4TinyOS language is based on the metamodel we introduce in [61]. Originating from TinyOS programming language (nesC), this metamodel includes the meta-entities and their relations for modeling TinyOS applications. TinyOS applications mainly have two parts, "Module" and "Configuration". These parts are also called components and they compose the core of the TinyOS application structure. These components must be linked together to execute the programs. The "Module" part is responsible to define the interfaces as "uses" or "provides", also, it implements the application's desired functions. A module defines "Events" and "Commands" and it owns a C-like syntax on its implementation part. The "Configuration" works as "wire station" for the TinyOS application, where the components used in the "Module" file are defined and linked together to create the complete program [9].

The tool supporting DSML4TinyOS has an EMF-based graphical modeling environment which enables creating DSML4TinyOS instance models according to DSML4TinyOS syntax and semantics definitions. Related modeling environment (see Figure 3) is built on the widely used Sirius platform.

Concept	Notation	Concept	Notation
Mote		Application	
Module		Configuration	
Components		Interface	
Module_Signature		Configuration_Signature	
Component		nesC	
Function		Wiring	
Event		Command	
Configuration Implementation		Module Implementation	
Task			

Table. 1: DSML4TinyOS concrete syntax notations

Table 1 lists the graphical notations used for the concrete syntax of the DSML4TinyOS language. TinyOS application models can be created by simply adding the language elements from the menu of the DSML4TinyOS tool. Implementation of the modeled applications can be automatically achieved via the code generation. In section 4, we will discuss both the example model shown in Figure 3 and nesC code (see Listing 2 and Listing 3) generated from this instance model. DSML4TinyOS benefits from the features of Acceleo [1] code generator to perform the code/text (M2T) transformations to generate the templates of the implementation files from instance TinyOS models. In addition, to improve the performance of the DSML4TinyOS, a static semantics in the modeling environment is defined

[1] Acceleo M2T Language, https://www.eclipse.org/acceleo/

and domain rules, such as name conflicts, are implemented as constraints to automatically check the application instance models designed by the developers before generating code. These constraints are implemented in Acceleo Query Language(AQL) [2]. The static semantics improves the quality of models and accordingly the quality of the generated code.

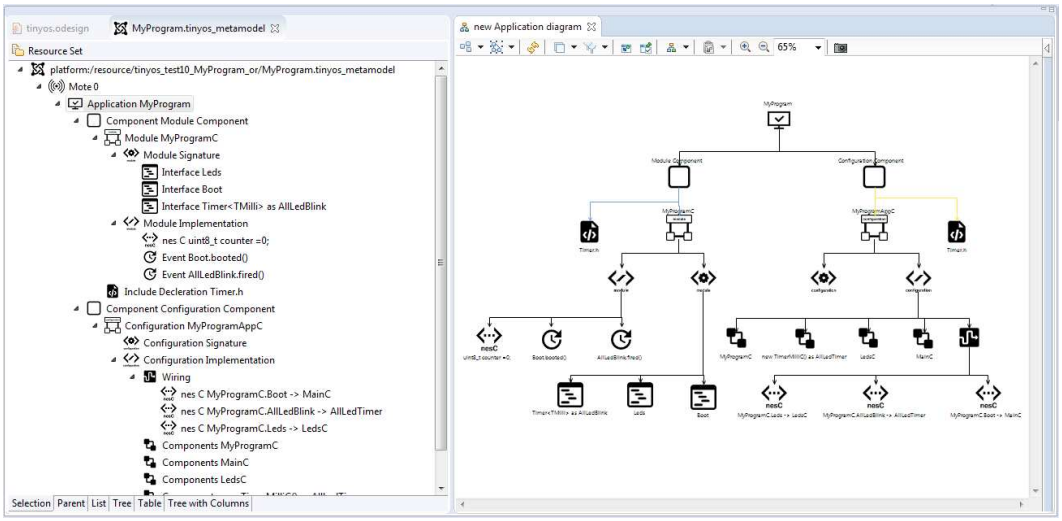

Figure 3: DSML4TinyOS graphical modeling environment

As mentioned above, TinyOS application models, conforming to the TinyOS metamodel, are stored as XMI files and they can be modified inside the DSML4TinyOS tool by adding or removing components. These changes are automatically reflected into the corresponding application code again by the tool. Similarly, the TinyOS application models retrieved by the RE4TinyOS interpreter (discussed in the next subsection) from the existing implementations can also be shown and processed again inside DSML4TinyOS tool. Hence, the synchronization of the system model and the existing implementation is realized in case of any modification made on the model or the code.

3.2 Reverse Engineering with RE4TinyOS

This section discusses the second part of the RTE methodology which enables the MDE-based RE of WSN applications running on TinyOS. Figure 4 gives a straightforward depiction of how RE works according to MDE concepts to convert the TinyOS code to a TinyOS model for any application.

TinyOS applications are written in a special programming language, called nesC for networked embedded systems. The nesC programming model combines the features of C programming language with the special needs in the WSN domain such as event-driven execution and component-oriented design [62]. In this study, we introduce the RE4TinyOS tool, which is designed to read any TinyOS application code written in nesC as the input and automatically generate the counterpart domain model representing this TinyOS application. Each RE output model is a serialized DSML4TinyOS instance and these models can be

[2] Acceleo Query Language, https://www.eclipse.org/acceleo/documentation/aql.html

automatically opened in the graphical modeling environment of DSML4TinyOS for further processing if needed.

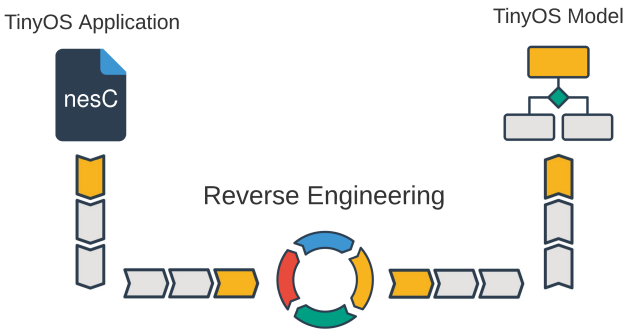

Figure 4: Overview of the proposed reverse engineering approach

To recognize the syntax and all the valid components (symbols, characters and expressions) of a particular programming language, a language recognizer or language interpreter is needed to read the elements and differentiate them from other normal statements of this language [63]. The language recognizer is used for different purposes like building a compiler or maybe analyze parts of the code to perform some operations. As it is well-known, parsing is the process of syntax analysis and breaks down the syntax of the language into smaller structures of symbol strings conforming to the formal rules and the grammar that govern the language. Also, parsers or syntax analyzers provide the identification of the languages. Since our aim is to retrieve the model of the WSN application from its program code, parsing is an essential process to identify and analyze the input TinyOS code.

We followed a two-step method to create the environment required to the reverse engineering of TinyOS applications. The first step is to design the parser, called TinyOS parser, that can read any TinyOS code, and by parsing the input, we can obtain the useful or desired parts of the TinyOS code in order to use them to build the model. The second step is implementing this parser design as a Java application that can read any TinyOS application code and extract the main elements and components from the code and hence build the TinyOS model.

Among many alternatives for the parser generator (e.g. ANTLR[3], GNU Bison[4], Gold[5], Ragel[6], Yacc[7]), ANTLR was utilized to build the TinyOS parser in this study. ANTLR (ANother Tool for Language Recognition) is a well-known computer-based language recognition tool, or more specifically a parser generator [64]. Features such as having a simple design, facilitating both lexing and parsing, providing various tools to inspect the parse tree and debugging the grammar as well as its flexibility to support various programming languages caused us to prefer ANTLR.

During a parser design, writing the grammar is a very crucial phase. It is the phase where the parser designers write the rules (Lexer and Parser rules) depending on analyzing the target system for their domains which in our case is the TinyOS system (i.e., the rules

[3]ANTLR, https://www.antlr.org/
[4]GNU Bison, https://www.gnu.org/software/bison/
[5]Gold, http://www.goldparser.org/
[6]Ragel, https://github.com/adrian-thurston/ragel
[7]Yacc, https://www.tuhs.org/cgi-bin/utree.pl?file=V6/usr/source/yacc

are written according to what type of input that will be parsed and what are the important information and parts are needed to be extracted). Listing 1 includes a small fragment from the parser rules we created by using ANTLR. In this parser implementation, more than 300 lines of grammar were prepared besides the lexer rules.

Listing 1: Excerpts from TinyOS parser rules

```
compilationUnit
: (includeDeclarationModule* componentDeclaration)? (includeDeclarationConfiguration*
↪  componentDeclaration EOF) ;
includeDeclarationModule
: '#' INCLUDE qualifiedName ;
includeDeclarationConfiguration
: '#' INCLUDE qualifiedName ;
qualifiedName
: singleLine ;
componentDeclaration
: moduleDeclaration
| configurationDeclaration ;
//This part is for the module file
moduleDeclaration
: moduleSignature  moduleImplementation ;
moduleSignature
: MODULE moduleName '('? ')'?  moduleSignatureBody ;
moduleName
: singleLine ;
moduleSignatureBody
: '{' usesOrProvides* '}' ;
usesOrProvides
: usesState
| providesState ;
usesState
: USES INTERFACE usesInterfaceDescription* ';'
| USES '{' (INTERFACE usesInterfaceDescription ';')* '}' ;
providesState
: PROVIDES INTERFACE providesInterfaceDescription* ';'
| PROVIDES '{' (INTERFACE providesInterfaceDescription ';')* '}' ;
```

The above excerpts show the general structure of the written parser rules. For instance, the line that starts with "compilationUnit", is considered as the start point of the whole parsing process. It states that two options exists; the first for the TinyOS model and the second for the TinyOS configuration that ends with "EOF" condition. The "componentDeclaration" line includes two main parts which are "moduleDeclaration" and "configurationDeclaration" respectively. The separator character 'I' declares that when the parsing process starts it has two options, module or configuration as they are the two main files of any TinyOS application. "moduleDeclaration" contains the details of the declaration. It has two parts which are "moduleSignature" and "moduleImplementation" respectively. It is worth indicating that these two parts are not separated by the 'I' character, which means

that any module should have both signature and implementation.

Since our aim is to build models by parsing TinyOS programs, the metamodel for TinyOS, which we previously introduced in [61], was considered as the main reference model and the TinyOS Parser was written and designed with consistency to the TinyOS metamodel.

The next step after creating the TinyOS Parser is to use this parser and hence benefit from its features. ANTLR has the property to transform or, in more specific words, generate codes from ANTLR-based parsers to several commonly-used programming languages like Java, Python, JavaScript, Go, C++ and Swift [64]. In our case, the target language is Java. An overview of the constructed TinyOS parser is shown in Figure 5.

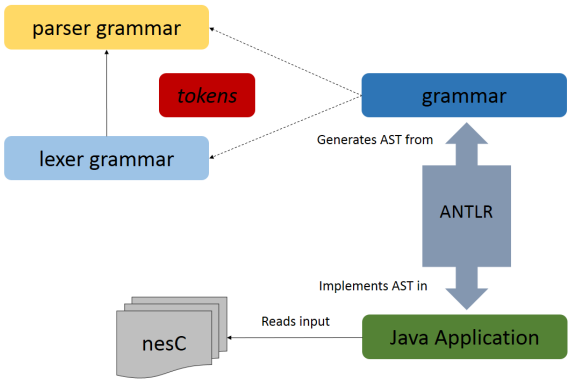

Figure 5: Parsing process for TinyOS applications

As depicted in the previous figure, our TinyOS Parser is taking the produced tokens from the Lexer and constructs a data structure known as Abstract Syntax Tree (AST) for the parsed TinyOS code. The created AST here records how the input structure and the components have been recognized by the TinyOS Parser. By default, the runtime library in ANTLR provides a mechanism for walking through the constructed AST and this operation is called a tree-walking. In our approach, the primary provided parse-tree-walker mechanism called "Parse-Tree Listener" [64] was used to walk the built tree of the TinyOS applications. Finally, the "Parse-Tree Listener" is integrated and implemented in a Java application-specific code which reads TinyOS programs (nesC codes) as input and calls every node in the constructed tree of the parsed TinyOS code by providing a subclass for every TinyOS Parser grammar that enables the application to enter and exit from every triggered node in order to obtain and extract the required information to build theTinyOS model from the code.

Since the Eclipse Modeling Framework (EMF) uses the XML Metadata Interchange (XMI) standard to express models by mapping their corresponding information and write all this information into the XMI file extension, this standard was utilized to build the TinyOS models inside the developed Java application. The Java application could extract all the required and important information from the input files (nesC code) and convert this information into a TinyOS model, i.e. XMI file containing a representation of the TinyOS application according to the TinyOS metamodel.

Above described processes of using TinyOS parser and the Java application are combined together to create the TinyOS Interpreter executed by the RE4TinyOS tool (Figure 6).

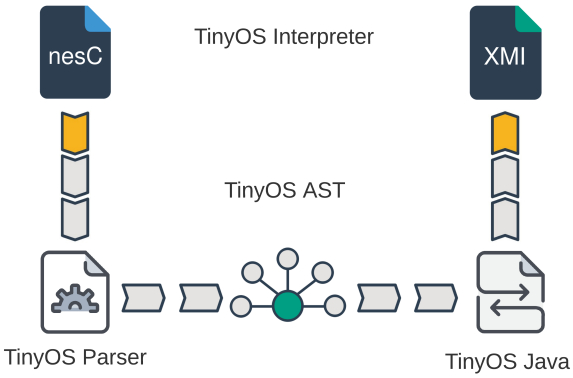

Figure 6: TinyOS Interpreter structure

The generated XMI files containing the model representations of the input TinyOS applications can be opened inside the DSML4TinyOS modeling tool without any human intervention. Hence, these model instances conforming to the TinyOS metamodel, can be visually seen and ready for modifications if needed.

To summarize, by applying the RE4TinyOS methodology, the software model of an existing TinyOS application can be achieved automatically. For this purpose, a developer only needs to give the code file of the related TinyOS application as the input for our RE4TinyOS tool. The built-in interpreter generates the corresponding model. This model is XMI serialized and can be opened and visually edited inside the DSML4TinyOS tool. If needed, any change made in the model is reflected into the code without any developer intervention.

4 Case Study

To demonstrate the usability of the proposed RTE methodology, a case study is discussed in this section. This case study exemplifies how the synchronization between TinyOS models and the corresponding code can be provided with the use of both DSML4TinyOS and RE4TinyOS tools together within a model-driven RTE process. Figure 7 depicts the application of our RTE methodology for this purpose. First, a software developer creates the model of the required TinyOS application in DSML4TinyOS from scratch and the corresponding code for this design is automatically generated. Then, the developer modifies the application code e.g. due to a change request and the corresponding TinyOS model, which is the updated version of the initial model, is generated inside RE4TinyOS. Finally, this updated model can be re-opened in the IDE of DSML4TinyOS and used for further processing.

In this study, let us consider the MDE of an application for a TinyOS mote (a WSN node), which displays the light emitting diodes (LEDs) on this mote when needed. The application, simply called MyProgram for the demonstration purposes, uses the “Boot” interface, executes the event “Boot.booted()” and calls the three LEDs via commands. In the “Boot.booted()” event, the command “AllLedBlink.startPeriodic(1000)” will be called. This command initializes a timer that gives interrupts for every 1000 milliseconds. Also, the

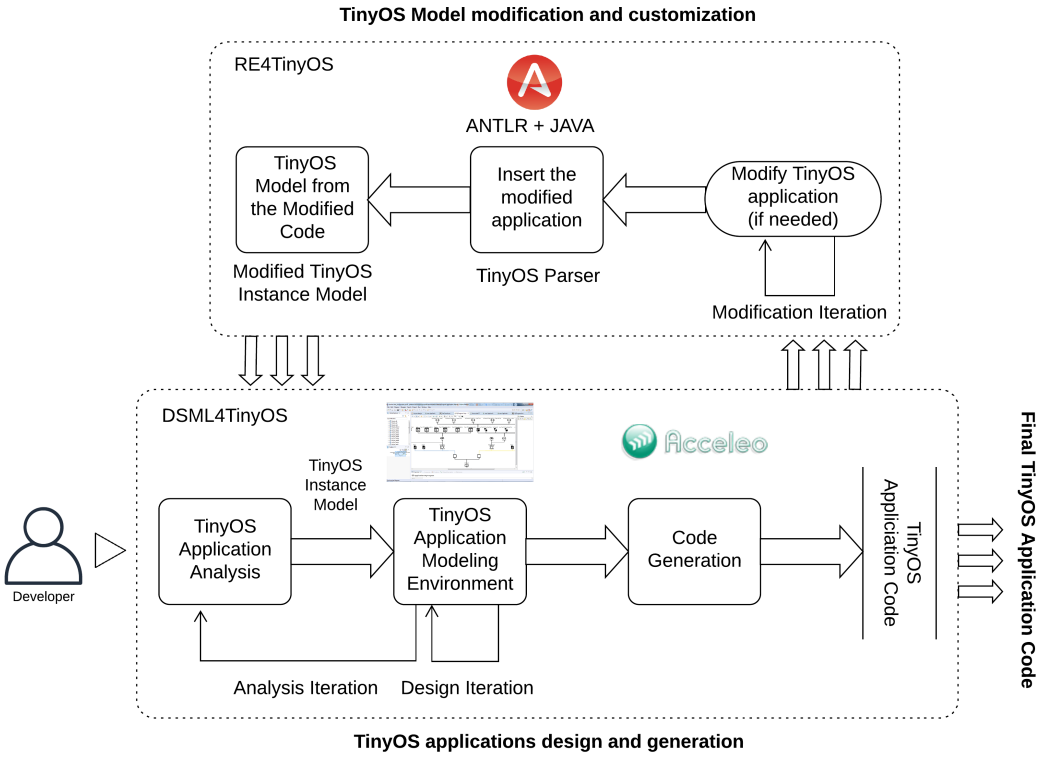

Figure 7: Using RTE to develop TinyOS applications

application displays a counter on the three LEDs of the mote. It uses the timer interface "Timer<TMilli>as AllLedBlink" and executes the second event by firing the timer in the event "AllLedBlink.fired()". Inside this event, the three commands are called. The event will call the command "Leds.led0On()", "Leds.led1On()", and "Leds.led2On()" one by one corresponding to each "Counter" value.

The above described TinyOS application was modeled graphically using DSML4TinyOS. Figure 8 shows the model of this application (as a DSML4TinyOS instance). This instance model represents the two parts of the required program ('Module' and 'Configuration') in a single model.

When a developer completes the design of a TinyOS application model in the IDE of DSML4TinyOS, corresponding nesC code for this design can be automatically generated inside the same IDE according to the Acceleo-based translational semantics of the DSML. Regarding our case study, following two listings include some code fragments generated from the above model (shown in Figure 8) for the module part (Listing 2) and the configuration part (Listing 3) of the related TinyOS application.

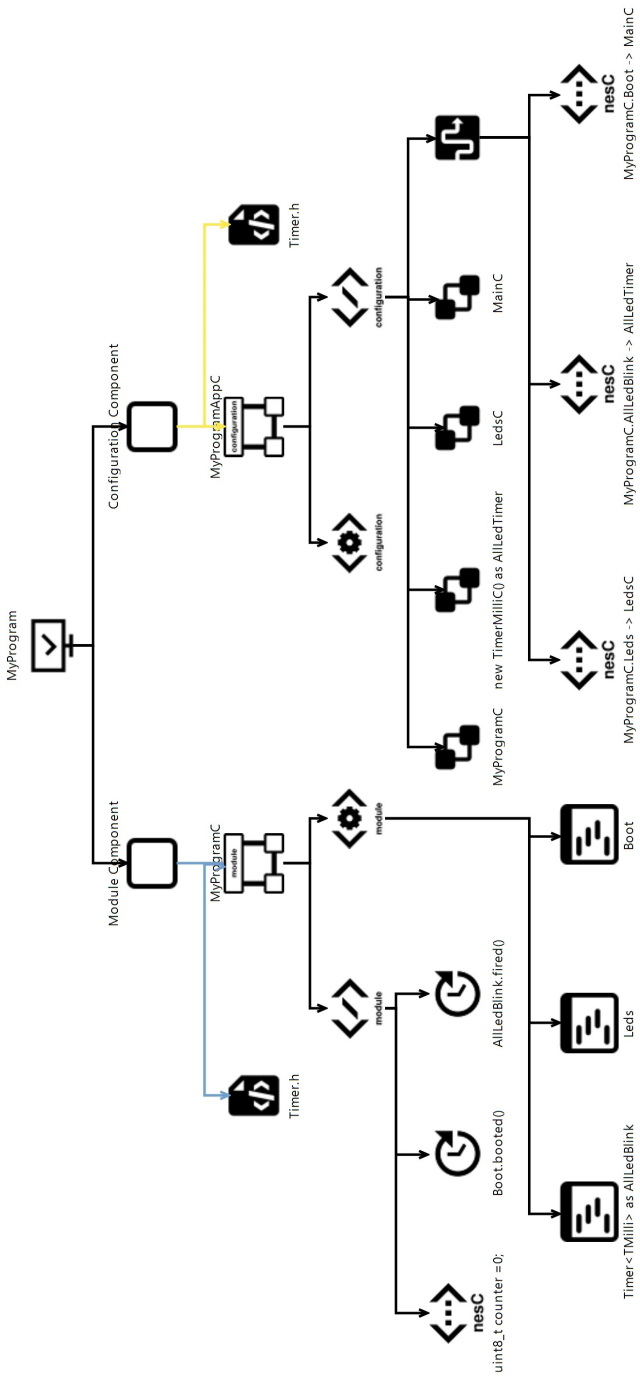

Figure 8: Graphical model of the original TinyOS application

Listing 2: nesC Module code auto-generated from the original application model

```
#include "Timer.h"
module MyProgramC @safe(){
    uses interface Leds;
    uses interface Boot;
    uses interface Timer<TMilli> as AllLedBlink;
}
implementation {
    uint8_t counter =0;
    event void Boot.booted() {
    /* Turn the three leds on */
    call Leds.led0On();
    call Leds.led1On();
    call Leds.led2On();
    /* call the timer every 1000 milliseconds */
    call AllLedBlink.startPeriodic( 1000 );
        }
    event void AllLedBlink.fired() {
    counter++;
    if (counter & 0x1) {
        call Leds.led0On(); }
    else { call Leds.led0Off();}
    if (counter & 0x2) {
        call Leds.led1On();}
    else { call Leds.led1Off();}
    if (counter & 0x4) {
        call Leds.led2On(); }
    else { call Leds.led2Off();}
   }
}
```

Listing 3: nesC Configuration code auto-generated from the original application model

```
#include "Timer.h"
configuration MyProgramAppC {
}
implementation {
    components MyProgramC;
    components MainC;
    components LedsC;
    components new TimerMilliC() as AllLedTimer;
    MyProgramC.Boot -> MainC;
    MyProgramC.AllLedBlink -> AllLedTimer;
    MyProgramC.Leds -> LedsC;
}
```

Listing 4: Modified nesC Module code of the application

```
#include "Timer.h"
#include "printf.h"
module MyProgramC @safe() {
  uses interface Leds;
  uses interface Boot;
  uses interface Timer <TMilli> as AllLedBlink;
  uses interface Timer <TMilli> as RedLedBlink;
  uses interface Timer <TMilli> as GreenLedBlink;
  uses interface Timer <TMilli> as YellowLedBlink;
}
implementation {
  uint8_t counter;
 task void printTask() {
  printf("Print task\n");}
  event void Boot.booted() {
   for (counter = 0; counter <= 31; counter++) {
   if (counter == 10)
    call RedLedBlink.startOneShot(counter);
   else if (counter == 20)
    call GreenLedBlink.startOneShot(counter);
   else if (counter == 30)
    call YellowLedBlink.startOneShot(counter);
   else printf("It will not blink any led\n"); }
    call AllLedBlink.startPeriodic(50);
    dbg("MyProgramC", "Application booted.\n");
    post printTask();
  }
  event void AllLedBlink.fired() {
    call Leds.led0On();
    call Leds.led1On();
    call Leds.led2On(); }
  event void RedLedBlink.fired() {
    printf("Blink the red led\n");
    call Leds.led0Toggle();}
  event void GreenLedBlink.fired() {
    printf("Blink the green led\n");
    call Leds.led1Toggle();}
  event void YellowLedBlink.fired() {
    printf("Blink the yellow led\n");
    call Leds.led2Toggle();}
}
```

When any change made in the application code, these changes can be reflected to the corresponding model with using the RE4TinyOS tool. Now, let us suppose that a developer wants to modify the above program with adding three new timers and a task. In the

modified application, every interface will blink just one specific led: “Timer<TMilli>as RedLedBlink” will blink the red led, “Timer<TMilli>as GreenLedBlink” will blink the green led and “Timer<TMilli>as YellowLedBlink” will blink the yellow led respectively. Hence, every event will be triggered independently: “RedLedBlink.fired()” will trigger the red led timer, “GreenLedBlink.fired()” will trigger the green led timer and “YellowLedBlink.fired()” will trigger the yellow led timer. Inside “Boot.booted()” event, a ”for loop” with including an ”if statement” is added to the code to test the counter, call one of the timers that will be fired and call the command to turn on the LED. Also, a new task is added and it will be called in “Boot.booted()” event. Following code listings (Listing 4 and Listing 5) include the modified versions of the module and configuration components of our TinyOS program in which the added / changed parts are highlighted in cyan color.

Listing 5: Modified nesC Configuration code of the application

```
#include "Timer.h"
#include "printf.h"
configuration MyProgramAppC {}
implementation {
  components MyProgramC, MainC, LedsC;
  components new TimerMilliC() as AllLedTimer;
  components new TimerMilliC() as RedLedTimer;
  components new TimerMilliC() as GreenLedTimer;
  components new TimerMilliC() as YellowLedTimer;
  MyProgramC.Boot -> MainC;
  MyProgramC.AllLedBlink -> AllLedTimer;
  MyProgramC.RedLedBlink -> RedLedTimer;
  MyProgramC.GreenLedBlink -> GreenLedTimer;
  MyProgramC.YellowLedBlink -> YellowLedTimer;
  MyProgramC.Leds -> LedsC;
}
```

To propagate above code modifications to the model of the application, RE4TinyOS tool was used. New version of the program was given as input to the RE4TinyOS and the tool successfully produced the serialized file for the model. This model was opened in the DSML4TinyOS modeling environment (see Figure 9) and it was examined that RE4TinyOS maintained the synchronization between the model and the code by automatically inserting new model elements and changing existing elements (e.g. “Boot.booted()” event was changed due to its new function implementation). As can also be seen from Figure 9, the modifications were seamlessly integrated into the modified model with retaining the unchanged model components.

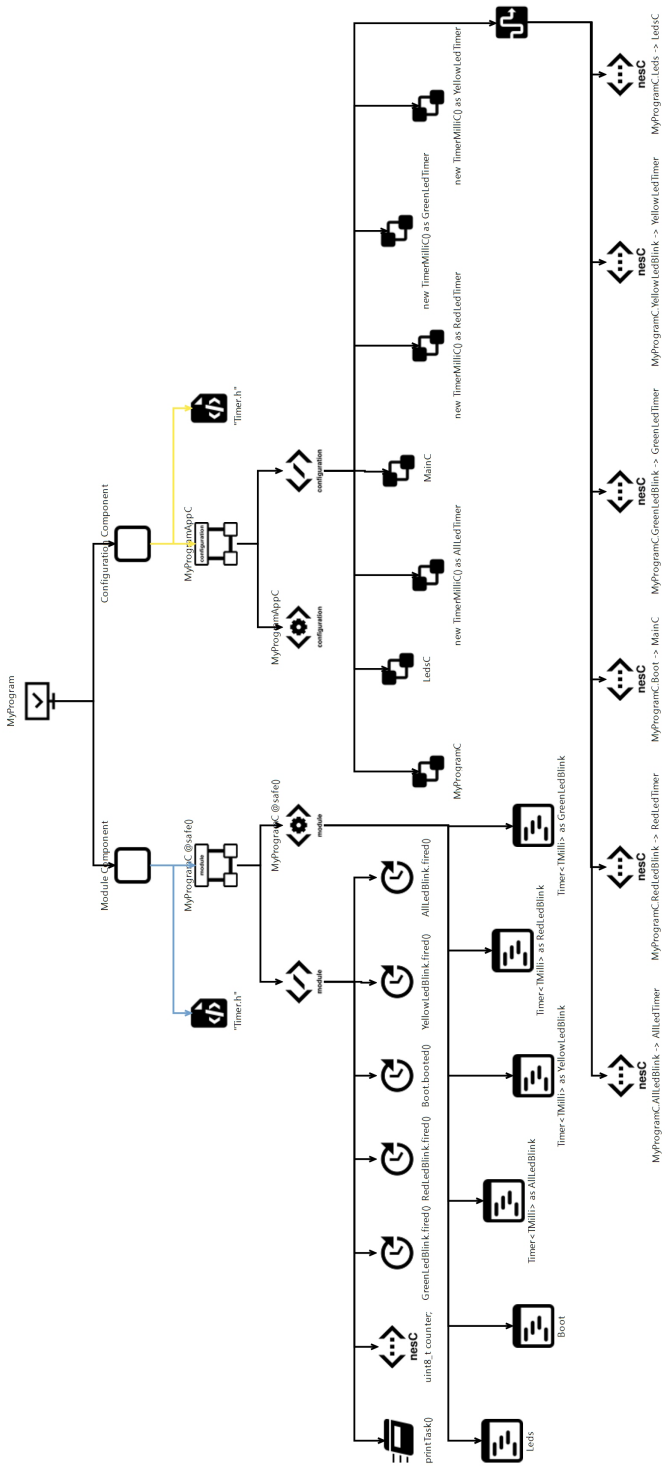

Figure 9: Graphical model of the modified TinyOS application

5 Evaluation

An evaluation of using both DSML4TinyOS and RE4TinyOS environments inside the proposed model-driven RTE toolchain was performed by taking into consideration the development of eight TinyOS applications, each for different domain with changing complexities, i.e. the number of sensors, type of network devices, structure of the messages and network events vary in the modeled applications. The TinyOS applications considered in this study are well-known in the WSN research community. All of them were originally developed by other TinyOS developers, most of them from the University of California. The complete implementations of all these applications are publicly available from the official TinyOS repository in Github [65]. For the construction of the evaluation study and the assessment of the achieved artefacts, we followed our multi-case evaluation method which we previously applied in the assessment of various DSMLs and MDE processes (e.g. [66, 67, 68, 69]) for different domains. Quantitative analysis and qualitative assessment steps of this method were updated and improved in this study for the usability evaluation of DSML4TinyOS and RE4TinyOS environments as will be discussed below.

During our evaluation, at first, each WSN application were modeled in DSML4TinyOS IDE according to the system specifications given in the GitHub distribution site. Then code for each application was automatically generated from these models and this code was compared with the complete implementation of the related application in TinyOS GitHub repository. In addition, we also measured and compared the time elapsed for developing these applications with and without using DSML4TinyOS to investigate how much it leads to reducing the total effort of the developers, i.e. the end users of DSML4TinyOS. Hence, a quantitative assessment of the language's throughput and development time performances was performed. In the second part of our evaluation, complete implementations of all these TinyOS applications in GitHub were processed in the RE4TinyOS environment to evaluate the capability of creating application models completely from already existing code which is crucial to integrate the implementations of the third party WSN applications into the MDE processes. In here, already existing code means the application was not previously designed and implemented with using DSML4TinyOS and RE4TinyOS toolchain. Hence, it does not own an application model to be used as an input for further system developments.

Before discussing the evaluation results, eight TinyOS applications used in the study are briefly described below. For more information about these applications, please see [65].

AntiTheft

AntiTheft is a WSN application for detecting thefts, that uses various aspects of TinyOS and its services. AntiTheft application can detect a theft by monitoring two events:

1. The change in the light level: It assumes that a stolen mote will be situated in a dark place.

2. The change in the acceleration rate: When thieves steal anything, they usually move too fast and run.

So, the application will report the theft by: 1) Alerting via turning on the light (e.g. a red LED), 2) Making a beep sound, 3) Reporting to the other nodes within the range by broadcasting messages and nodes will also turn on their red LEDs, 4) Reporting to a central node using a multi-hop routing algorithm

BarrierBounce

This application uses the Active Message (AM) operations for the packet transmission and reception. The application runs three independent threads, each thread has an infinite loop that sends a packet and waits for receiving back the same packet. Three LEDs are used as indicators if the messages were received successfully, where Thread 0, Thread 1, Thread 2 use LED0, LED1, and LED2 respectively. The application is operated between two motes and works in an asynchronous fashion.

MultihopOscilloscope

This is a data-collection application that works to sample its sensors in a periodic time frame and then broadcasts the messages received from its sensors (e.g. for temperature or light) after each reading. Also, a Java application is provided to display the readings from the sensors.

MViz

The MViz is a multihop collection network application, where the nodes with specific IDs are used as the collection roots. The application samples the sensor values from a specific WSN platform and sends them to the collection roots. Then, these roots send the packets to the serial port to visualize them on MViz Java network visualization tool.

PacketParrot

This application keeps logs for the packets when they are received from the radio channel to the flash and erases the logs on subsequent power cycle. The application uses three LEDs. It turns on the yellow led when the packet is received and turns off if the packet is logged successfully. Also, it turns on the red LED after erasing the log packets it receives from the radio to flash, while the green LED toggles when transmitting the logged packets. This application is designed to show the LogWrite and LogRead abstractions on the TinyOS.

RadioStress

This application is designed to use three different threads to send three different messages with unique IDs for the ActiveMessage (AM), which is the core of TinyOS communication abstraction [8]. Upon receiving messages, the receiver will receive these messages depending on their IDs using three different threads, and then it toggles one of the LEDs for every message when the transmission is successful.

Sense

The Sense is a simple sensing application that periodically samples data from the sensors by initializing a timer which will signal a "read event" and displays the bits of the sampled readings on the LEDs of the nodes.

TestDissemination

This application is designed to send two constant data objects from a node with value (TOS_NODE_ID mod 4 equals 1) to all the other nodes in the network. The sender will send a 32-bit value and a 16-bit value and it will toggle its LED0 and LED1 and fires a timer. When the receiver receives the correct message, it toggles its LED0 and prints the message "Received new correct 32-bit value" if the 32-bit value is received and prints the message "Received new correct 16-bit value" if the 16-bit value is received.

All of the the above applications were modeled from scratch inside the graphical editor of DSML4TinyOS and corresponding code for the applications was generated. This generated code was compared to the original code, i.e. complete implementation of each application available at [65]. Accordingly, the percentage of the code generation was calculated by comparing the lines of code (LoC) for the generated applications with the LoC of the original applications. The bar chart in Figure 10 shows the comparison results for both Module and Configuration parts of all these TinyOS applications.

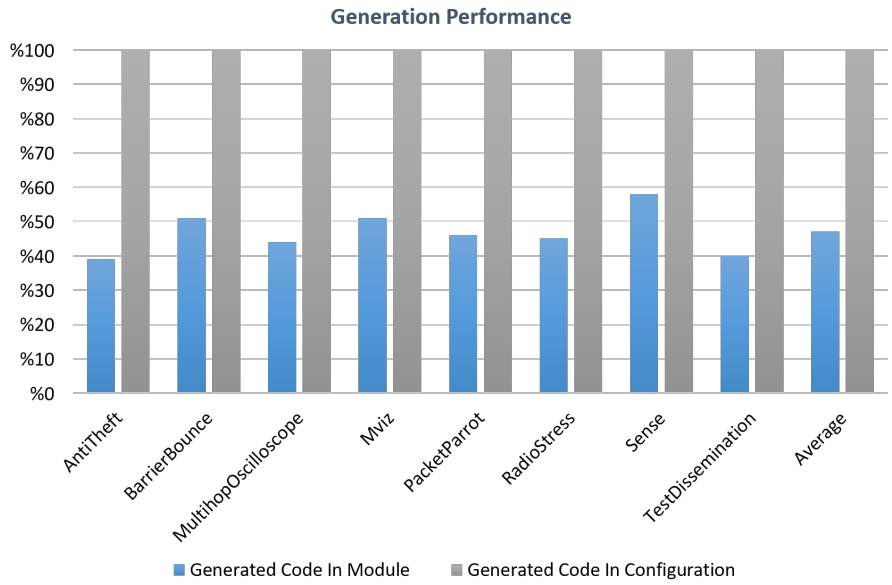

Figure 10: The percentage of the auto-generated code of Module and Configuration parts for all TinyOS applications

As can be seen from the bar chart, LoC generation ratios of the module components of *AntiTheft, BarrierBounce, MultihopOscilloscope, MViz, PacketParrot, RadioStress, Sense and TestDissemination* applications are *39%, 51%, 44%, 51%, 46%, 45%, 58%, 40%* respectively. That means, for instance, approximately *42%* LoC (delta code) need to be added into the auto-generated code of the module part of the Sense application to achieve the full implementation. However, the results also show that DSML4TinyOS succeeded in generating the configuration parts of all TinyOS applications completely in this study.

The main reason of having low ratio of LoC generation for module parts in comparison with the configuration parts is the modules for all TinyOS applications should inevitably include the complete definitions of all fundamental application functions which are much

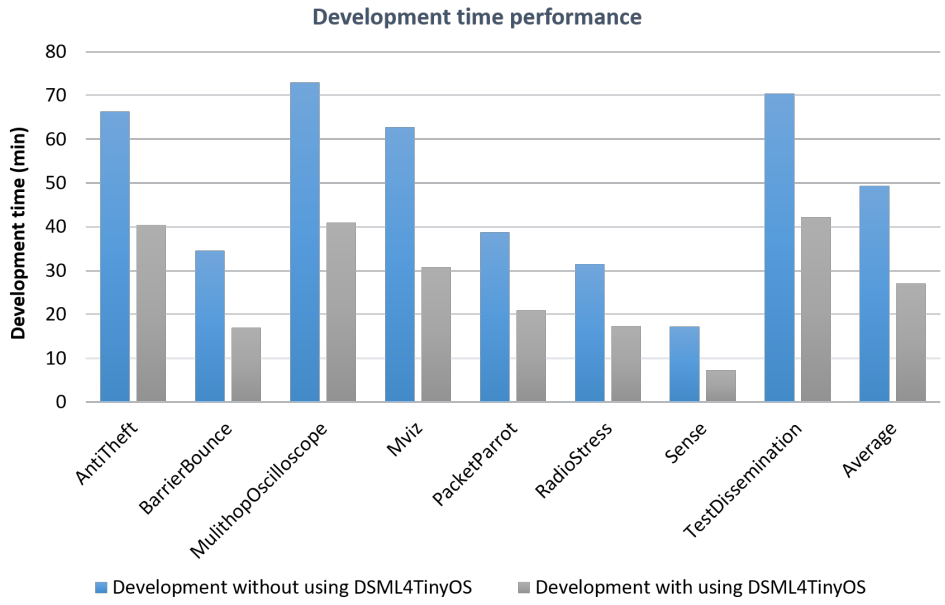

Figure 11: Comparison of the time elapsed for developing WSN applications with and without using DSML4TinyOS

more bigger than the TinyOS application configurations. Moreover, we also realized that the many parts of the module components contain C-like selection and repetition statements (control structures) as well as many variable definition and assignment commands. Related statements and commands pertain to the platform-specific programming infrastructure and naturally they are not defined in the metamodel of DSML4TinyOS language to provide a higher abstraction level of modeling TinyOS applications. Since the graphical syntax of the language only allows modeling WSN components and their relations according to TinyOS specifications, code generated from these models needs to be extended with abovementioned platform-specific statements and commands for the complete implementations.

It is worth indicating that the slight variability encountered in the ratios of the auto-generated modules originates from the changing complexity and comprehensiveness of all these applications used in this evaluation. The number and the complexity of the WSN events and the functions covered by these applications directly affect the throughput of the code generation mechanism in DSML4TinyOS since additional control structures and definitions need to be inserted and these are not directly supported in DSML4TinyOS models as discussed above. For instance, code generation ratio for the AntiTheft application was relatively lower e.g. in comparison with the Sense application due to the complexity of both the architecture of the related WSN and the composition of the monitoring events.

Taking into account all measurements made during MDE of all TinyOS applications in this study, we can conclude that, on the average, approximately 47% of the module parts of the TinyOS applications can be automatically generated by just modeling with DSML4TinyOS while use of the language enables generating 100% of the configurations for all applications.

Considering the impact of using the proposed MDE approach in the amount of effort

taken for the development of WSN applications, we measured the time elapsed for developing each application both with and without using DSML4TinyOS. During development with using DSML4TinyOS and its tool, the total time elapsed for system modeling, code generation and code completion steps was calculated. For the system development without using the DSML, system analysis, design and implementation (coding the whole application in nesC) steps in the conventional TinyOS application development process were taken into account and the total elapsed time to complete all these steps for each case study was again calculated. As the chart in Figure 11 shows, on the average, it took approximately 41 minutes to develop these WSN applications without using DSML4TinyOS while the utilization of this DSML and its IDE enabled reducing the application development to 24 minutes on the average. In other words, the time required to develop these applications decreased by approximately 45%.

In the second step of our evaluation process, the usability of the RE4TinyOS environment was assessed. Similar to the case study discussed in Section 4, nesC code generated by DSML4TinyOS for these eight TiynOS applications were given as input to RE4TinyOS and the serialized version of the corresponding instance models in XMI were automatically produced by the built-in interpreter of RE4TinyOS. These models were successfully opened in DSML4TinyOS. That result is somehow expected since the code initially processed by RE4TinyOS was already previously generated by modeling in DSML4TinyOS. Hence, it was easy to restore the corresponding models again from this application code. However, we also experienced that RE4TinyOS succeeded in reflecting any changes made in the auto-generated code of all these TinyOS programs into the corresponding system models as long as the changes made on this code were correct according to nesC specifications. RE4TinyOS was able to integrate any modification, addition or deletion of nesC program parts into the existing DSML4TinyOS instance models of these applications leading the synchronization between the updated versions of TinyOS application code and models.

Perhaps the most interesting part of the evaluation was to assess the efficiency of using RE4TinyOS for the reverse engineering of the already existing TinyOS applications which were not previously designed and implemented with using DSML4TinyOS and/or RE4TinyOS tools. For this purpose, we used the full implementations of these eight TinyOS applications publicly available in TinyOS Github repository [65]. As previously indicated, these applications were written by other developers and we did not make any change on the related program files (TinyOS modules and configurations) before using them in this evaluation.

When the complete code of all these eight TinyOS applications was given as input, RE4TinyOS environment was again succeeded in automatically generating the serialized versions of the TinyOS software models of all these applications, and the produced models were processed and successfully opened in the DSML4TinyOS IDE. For instance, Figures 12 and 13 show the TinyOS models generated from the reverse engineering of the Sense and AntiTheft applications downloaded from the TinyOS GitHub repository. As can be seen from these screenshots, parts of the TinyOS applications including components, interfaces, commands, and events are all now represented in DSML4TinyOS notation.

Evaluation made over the reverse engineering of these TinyOS applications confirms that, if needed, RE4TinyOS tool can also be used independently from the MDE toolchain, i.e. the TinyOS application that will be processed by the RE4TinyOS tool could be previously implemented via using any other method and environment. The developers can achieve software models of these existing applications. Furthermore, it is straightforward to visually work on these recovered models at a higher level of abstraction, make modifications on them

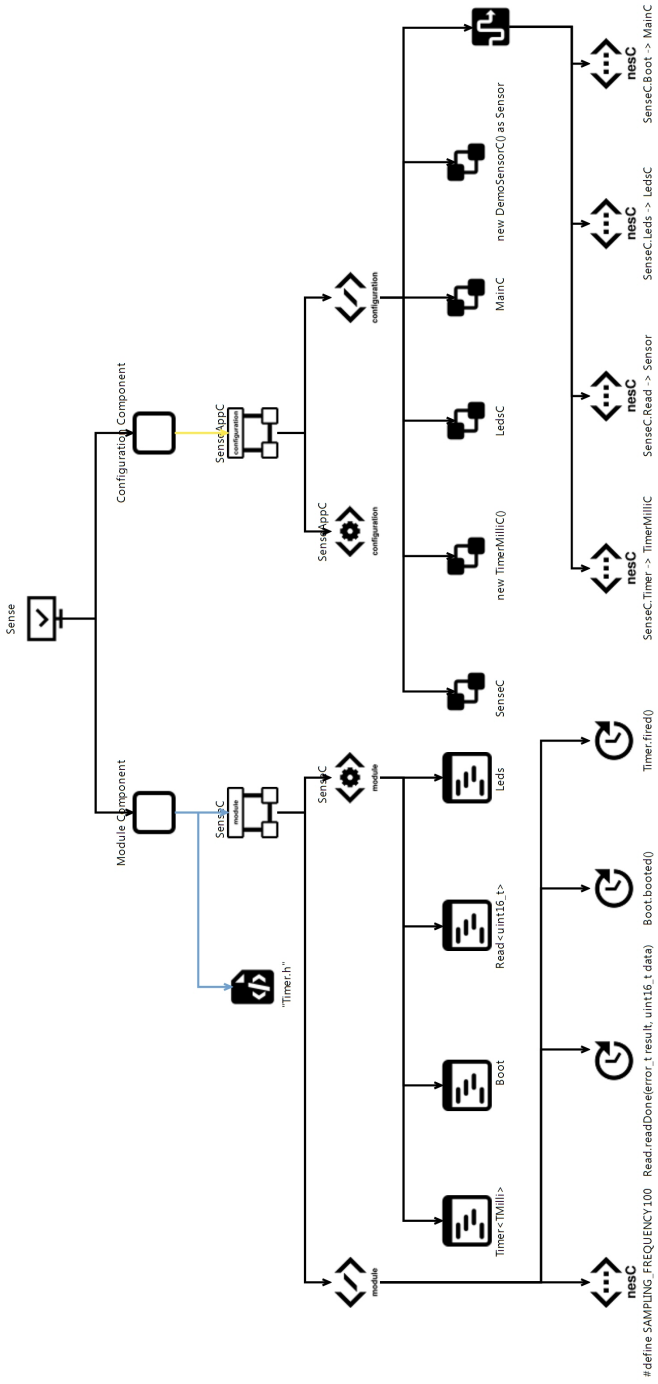

Figure 12: Graphical model of the reverse engineered Sense application

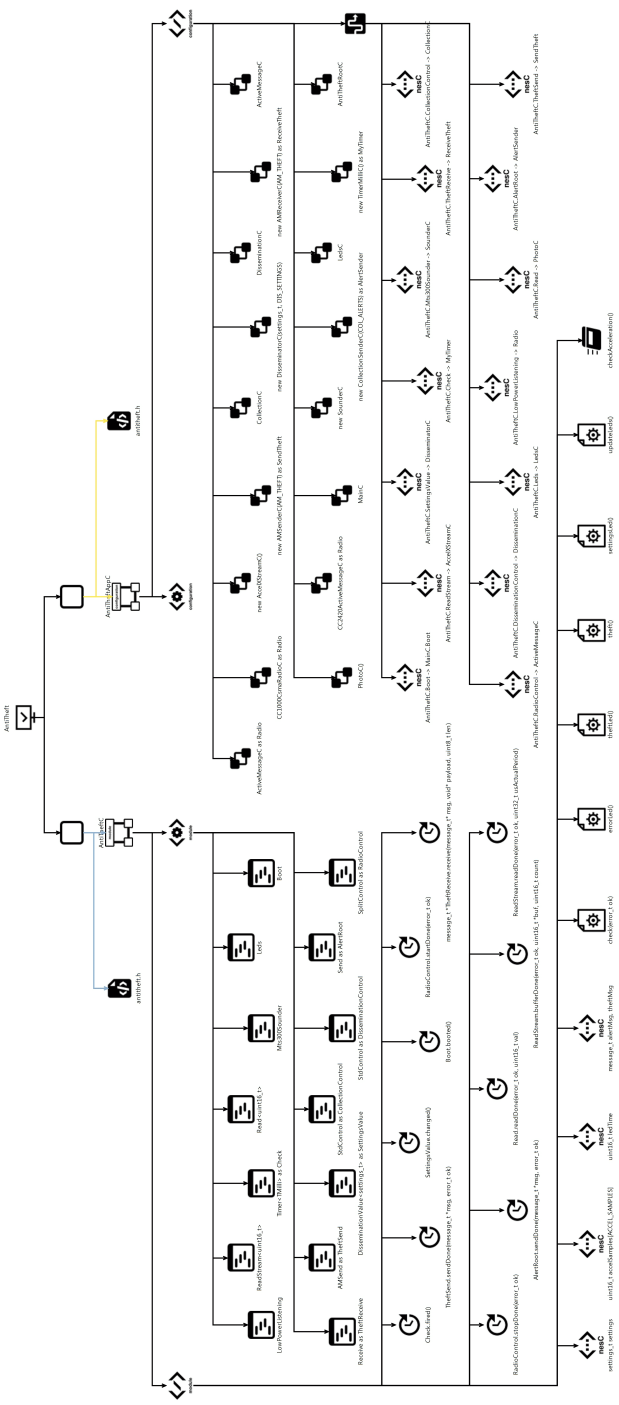

Figure 13: Graphical model of the reverse engineered AntiTheft application

and then reflect these changes to the exact implementations.

Our evaluation also showed that the models for all main parts of these TinyOS applications were retrieved by using RE4TinyOS without any error. These retrieved parts include *event*, *task*, *component*, *interface*, *Command*, *Helper-function*, and *Wiring* definitions. Block structures of the application events were also retrieved. However, it was also detected that the applied reverse engineering process was unable to completely represent internal descriptions of some of these events (covering e.g. some selective and repetitive control statements and variable definitions which are too specific to the underlying language's syntax) as graphical components in the output model since corresponding meta-entities and relations are missing in the TinyOS metamodel currently used by the RE4TinyOS parser. Although that caused the reverse engineering of these events incomplete, RE4TinyOS supported still keeping these unconverted specifications as annotations inside the serialized model and when any changes made to the model in the visual editor, these specifications were automatically integrated with the new code generated from the modified model.

Finally, a limitation on the visual appearance of the models retrieved from the legacy applications was encountered since these models have no built-in layout information. When the application models initially created in DSML4TinyOS are updated with too many components due to the significant amount of modifications in the corresponding code, similar visual impairments may occur again. This lack of layout information makes the look of the model unorganized inside the IDE and some manual intervention is needed to re-organize the graphical appearance of the new or updated elements of the model. Although, it is possible to automatically re-organize the appearance of the whole model using the features of Sirius and/or Eclipse, this re-organization mostly spoils the layout of the unaltered elements too. However, that technical limitation is originated from the layout mechanism of Eclipse and can be eliminated e.g. using another graphical editing framework for DSML4TinyOS IDE.

5.1 Threats to the validity

As it is the case in any experimental study, there are some threats to the validity of the performed evaluation. The threats can be originated from different validity types including internal validity, external validity, construct validity, and conclusion validity described in [70].

Internal validity of our experiments relates to the degree to which the design, implementation and evaluation of all TinyOS applications in here are likely to prevent systematic errors. To minimize such threats, we followed a strict protocol for both the MDE of these applications and evaluating the results. Moreover, a multi-case evaluation method that was observed to work efficiently before was also used in this study. Especially, this enabled us to perform both the quantitative analysis of the throughput and time performance and the qualitative assessment of the reverse engineering.

For the threats to external validity, generalization of the achieved results should be considered. This threat was mitigated by especially considering eight different TinyOS applications which were all known by the WSN research community and more importantly all of them were implemented previously by the TinyOS developers who did not involve in the design and implementation of the model-driven RTE toolchain being evaluated in this study. For instance, the capability of the RE4TinyOS tool was assessed by the reverse engineering of these third party TinyOS programs which were not previously modeled in DSML4TinyOS. Similarly, use of both the graphical modeling and code generation features of DSML4TinyOS

were demonstrated during the conducted evaluation again with considering these already existing TinyOS applications instead of fictitious / trivial ones just created for this study.

Construct validity refers to what extent the operational measures and the cases that are studied really represent what the researchers has in mind. In our work, we aim at evaluating the usability of this new model-driven RTE methodology and its toolchain for the development of TinyOS applications. Code generation performance were measured and the reverse engineering of the applications were tested by considering which parts / components of the TinyOS programs can be retrieved and represented at the corresponding models. The evaluation was made over eight different TinyOS applications. This number can be found satisfactory when comparing with the similar MDE studies for WSNs and TinyOS applications where mostly just single application development was considered to demonstrate the usability. Furthermore, the TinyOS applications selected for our evaluation have different complexities considering the structure of the WSNs-to-be-implemented by means of the size and type of the sensors, devices and messages as given in their descriptions. To mitigate this threat, this evaluation also included the MDE of the applications which were ready to execute and accessible from the TinyOS repository.

Finally, for the conclusion validity, we need to consider the credibility of the achieved results. Within this context, we believe that selecting multiple case studies with varying complexities also helped to minimize the risk on the conclusion validity. For instance, the quantitative analysis on the throughput performance showed that configuration parts of all TinyOS applications can be generated completely from the models while only approximately the half of each module can be generated. However, the results also showed that the difference between the individual rates of LoC generation for each module was not so high as depicted in Figure 10 although the complexity and the size of the TinyOS applications may vary.

6 Conclusion

A model-driven RTE methodology for the design and implementation of TinyOS applications have been introduced in this paper. The methodology herein consists of two parts, forward and reverse engineering. Forward engineering of the TinyOS applications is provided using a DSML, called DSML4TinyOS and its IDE. WSN applications to be executed on TinyOS are modeled by using the graphical syntax of DSML4TinyOS language. Both module and configuration parts of the applications can be modeled. nesC code for the implementation of the designed models is automatically generated in the same IDE over a series of model-to-code transformations. Reverse engineering of the TinyOS applications is realized by using another IDE, called RE4TinyOS. RE4TinyOS enables retrieving the application models from TinyOS programs written in nesC, which paves the way for using these models inside an MDE toolchain. Hence, any modification made in the application code can be reflected into the application model and vice versa. Models retrieved by RE4TinyOS can be processed inside DSML4TinyOS environment and can be updated with new TinyOS components if needed. Both DSML4TinyOS and RE4TinyOS IDEs introduced in this paper are publicly available at [71] including application examples.

The potential users of the proposed RTE methodology along with the supported IDE can be both novice and expert TinyOS application developers. Novice developers who have no or little TinyOS programming experience may quickly design and implement their WSNs on TinyOS by using the modeling features of DSML4TinyOS. Specifically, MDE

environment supported with DSML4TinyOS may also facilitate WSN developers' learning and adaptation to TinyOS in case they are already familiar with the WSN domain or even they own experience on using WSN platforms other than TinyOS. Developers experienced on TinyOS programming may benefit from both applying the proposed framework and utilizing DSML4TinyOS and RE4TinyOS since the evaluation in our study exposed that graphical modeling and code generation from these models significantly reduced the development effort. Furthermore, RTE brought by RE4TinyOS may assist the experienced users e.g. in handling and updating already existing TinyOS applications to achieve more improved versions of the same applications. Despite all these features and opportunities, model-driven RTE tools introduced in this paper may still fail to fully support the software and hardware heterogeneity encountered in various industrial WSN applications. For instance, WSN application models created with DSML4TinyOS can be too complex and difficult to manage as the heterogeneity in the WSN components of the developed system continuously increases. In fact, this problem is not specific to our proposal and as also indicated in [72], it is one of the current important challenges of applying MDE techniques on developing WSN and IoT systems. Nevertheless, we need further investigation on DSML4TinyOS language's support on modeling heterogeneity to provide its wider adoption and use in the development of large-scale industrial WSN applications.

Conducted multi-case evaluation showed that it is possible to obtain the configurations of the TinyOS applications completely just modeling with DSML4TinyOS whereas the same process leads the automatic creation of almost half of the module parts of these applications. Interestingly, module generation rates are closer to each other in individual application basis although the complexity of the applications vary. This result can be found promising in the sense that the similar code generation ratios can be possible when other TinyOS applications are developed with our MDE methodology. Our evaluation also showed that the time required for developing TinyOS applications from scratch decreased approximately to its half when DSML4TinyOS and its IDE are used. Moreover, we investigated the efficiency of using RE4TinyOS environment for the reverse engineering of the applications which are previously developed with or without using DSML4TinyOS. The results showed that both model-code synchronization and the integration of existing TinyOS applications which do not have system models previously, into the proposed MDE are possible with using RE4TinyOS. Models for all main parts of the complete applications in the TinyOS Github repository were successfully retrieved without any error. However, some of the internal TinyOS event specifications of these existing applications could not be represented in the newly generated models since corresponding meta-entities are missing in the current TinyOS metamodel used by the RE4TinyOS parser. In our future work, we aim at first extending this metamodel with some additional components for event internals and then improving the parser features with the utilization of this new metamodel. Another feature that we plan to integrate into RE4TinyOS, is the ability to add some elements automatically once their counterparts are included in the model. For example, if "uses interface" definition for a specific element is added in the "Module" file, then the counterpart element to be used to wire the both components will be defined and added automatically in the "Configuration" file. This feature can minimize the error-rate and increase the productivity especially during the reverse engineering of the existing WSN applications.

Another future work will consider extending our model-driven RTE methodology to support the development of WSN applications to be worked on other operating systems and platforms. For instance, the syntax and semantics definitions of the current DSML can be extended to support WSN operating systems other than TinyOS such as Contiki,

LiteOS, MANTIS and Nano-RK, so modeled WSN applications can be executed on various platforms.

Acknowledgement

Hussein Marah would like to thank Turkish government for Turkiye Scholarships (YTB) program. This research was partially supported by Flanders Make, a Flemish strategic research center for the manufacturing industry.

References

[1] I. F. Akyildiz, M. C. Vuran, Wireless Sensor Networks, John Wiley & Sons, 2010.

[2] Y. Zhang, L. Sun, H. Song, X. Cao, Ubiquitous WSN for healthcare: Recent advances and future prospects, IEEE Internet of Things Journal 1 (4) (2014) 311–318. doi:10.1109/JIOT.2014.2329462.

[3] T. H. Feiroz Khan, D. S. Kumar, Ambient crop field monitoring for improving context based agricultural by mobile sink in WSN, Journal of Ambient Intelligence and Humanized Computing 11 (1) (2020) 1431–1439. doi:10.1007/s12652-019-01177-6.

[4] X. Hu, L. Yang, W. Xiong, A novel wireless sensor network frame for urban transportation, IEEE Internet of Things Journal 2 (6) (2015) 586–595. doi:10.1109/JIOT.2015.2475639.

[5] K. Ghosh, S. Neogy, P. K. Das, M. Mehta, Intrusion detection at international borders and large military barracks with multi-sink wireless sensor networks: An energy efficient solution, Wireless Personal Communications 98 (1) (2018) 1083–1101. doi:10.1007/s11277-017-4909-5.

[6] H. Ghayvat, S. Mukhopadhyay, X. Gui, N. Suryadevara, WSN- and IoT-based smart homes and their extension to smart buildings, Sensors 15 (5) (2015) 10350–10379. doi:10.3390/s150510350.

[7] Q. Chi, H. Yan, C. Zhang, Z. Pang, L. D. Xu, A reconfigurable smart sensor interface for industrial wsn in iot environment, IEEE Transactions on Industrial Informatics 10 (2) (2014) 1417–1425. doi:10.1109/TII.2014.2306798.

[8] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, D. Culler, TinyOS: An operating system for sensor networks, in: W. Weber, J. M. Rabaey, E. Aarts (Eds.), Ambient Intelligence, Springer Berlin Heidelberg, 2005, pp. 115–148. doi:10.1007/3-540-27139-2_7.
URL https://doi.org/10.1007/3-540-27139-2_7

[9] P. Levis, D. Gay, TinyOS Programming, Cambridge University Press, 2009.

[10] J. Whittle, J. Hutchinson, M. Rouncefield, The state of practice in model-driven engineering, IEEE Software 31 (3) (2014) 79–85. doi:10.1109/MS.2013.65.

[11] B. Lelandais, M.-P. Oudot, B. Combemale, Applying model-driven engineering to high-performance computing: Experience report, lessons learned, and remaining challenges, Journal of Computer Languages 55 (1) (2019) 1–10. doi:10.1016/j.cola.2019.100919.

[12] B. Terzić, V. Dimitrieski, S. Kordić, G. Milosavljević, I. Luković, Development and evaluation of microbuilder: a model-driven tool for the specification of rest microservice software architectures, Enterprise Information Systems 12 (8-9) (2018) 1034–1057. doi:10.1080/17517575.2018.1460766.

[13] M. A. Mohamed, M. Challenger, G. Kardas, Applications of model-driven engineering in cyber-physical systems: A systematic mapping study, Journal of Computer Languages 59 (1) (2020) 1–19. doi:10.1016/j.cola.2020.100972.

[14] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys 37 (4) (2005) 316–344. doi:10.1145/1118890.1118892.

[15] T. Kosar, S. Bohra, M. Mernik, Domain-specific languages: A systematic mapping study, Information and Software Technology 71 (1) (2016) 77–91. doi:10.1016/j.infsof.2015.11.001.

[16] F. Ulrich, Domain-specific modeling languages: Requirements analysis and design guidelines, in: I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen, B. Jorn (Eds.), Domain Engineering, Springer, 2013, pp. 133–157. doi:10.1007/978-3-642-36654-3_6.

[17] C. Vicente-Chicote, F. Losilla, B. lvarez, A. Iborra, P. Sanchez, Applying MDE to the development of flexible and reusable wireless sensor networks, International Journal of Cooperative Information Systems 16 (3-4) (2007) 393–412. doi:10.1142/S021884300700172X.

[18] K. Doddapaneni, E. Ever, O. Gemikonakli, I. Malavolta, L. Mostarda, H. Muccini, A model-driven engineering framework for architecting and analysing wireless sensor networks, in: Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications, SESENA '12, IEEE Press, 2012, pp. 1–7. doi:10.1109/SESENA.2012.6225729.
URL https://doi.org/10.1109/SESENA.2012.6225729

[19] I. Malavolta, H. Muccini, A study on MDE approaches for engineering wireless sensor networks, in: 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, 2014, pp. 149–157, ISSN: 2376-9505. doi:10.1109/SEAA.2014.61.
URL https://doi.org/10.1109/SEAA.2014.61

[20] K. Tei, R. Shimizu, Y. Fukazawa, S. Honiden, Model-driven-development-based stepwise software development process for wireless sensor networks, IEEE Transactions on Systems, Man, and Cybernetics: Systems 45 (4) (2015) 675–687. doi:10.1109/TSMC.2014.2360506.

[21] C. Durmaz, M. Challenger, O. Dagdeviren, G. Kardas, Modelling Contiki-Based IoT Systems, in: R. Queirós, M. Pinto, A. Simões, J. P. Leal, M. J. Varanda (Eds.), 6th Symposium on Languages, Applications and Technologies (SLATE 2017), Vol. 56 of

OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017, pp. 5:1–5:13. doi:10.4230/OASIcs.SLATE.2017.5.
URL http://drops.dagstuhl.de/opus/volltexte/2017/7940

[22] T. Rodrigues, F. C. Delicato, T. Batista, P. F. Pires, L. Pirmez, An approach based on the domain perspective to develop WSAN applications, Software & Systems Modeling 16 (4) (2017) 949–977. doi:10.1007/s10270-015-0498-5.

[23] T. Asici, B. Karaduman, R. Eslampanah, M. Challenger, J. Denil, H. Vangelhuwe, Applying model driven engineering techniques to the development of contiki-based iot systems, in: 2019 IEEE/ACM 1st International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT), 2019, pp. 25–32. doi:10.1109/SERP4IoT.2019.00012.
URL https://doi.org/10.1109/SERP4IoT.2019.00012

[24] P. Dantas, T. Rodrigues, T. Batista, F. C. Delicato, P. F. Pires, W. Li, A. Y. Zomaya, Lwissy: A domain specific language to model wireless sensor and actuators network systems, in: 2013 4th International Workshop on Software Engineering for Sensor Network Applications (SESENA), 2013, pp. 7–12. doi:10.1109/SESENA.2013.6612258.
URL https://doi.org/10.1109/SESENA.2013.6612258

[25] A. J. Salman, A. Al-Yasiri, Developing domain-specific language for wireless sensor network application development, in: Internet Technology and Secured Transactions (ICITST), 2016 11th International Conference for, IEEE, 2016, pp. 301–308.

[26] B. R. Barricelli, S. Valtolina, A visual language and interactive system for end-user development of internet of things ecosystems, Journal of Visual Languages & Computing 40 (1) (2017) 1–19. doi:10.1016/j.jvlc.2017.01.004.

[27] N. Bak, B.-M. Chang, K. Choi, Smart block: A visual block language and its programming environment for iot, Journal of Computer Languages 60 (1) (2020) 1–19. doi:10.1016/j.cola.2020.100999.

[28] T. Hettel, M. Lawley, K. Raymond, Model synchronisation: Definitions for round-trip engineering, in: A. Vallecillo, J. Gray, A. Pierantonio (Eds.), Theory and Practice of Model Transformations, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 31–45.

[29] H. Giese, R. Wagner, From model transformation to incremental bidirectional model synchronization, Software & Systems Modeling 8 (1) (2009) 21–43. doi:10.1007/s10270-008-0089-9.

[30] L. Favre, Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution, Engineering Science Reference, 2010, google-Books-ID: e4RLuAAACAAJ.

[31] H. M. Marah, M. Challenger, G. Kardas, RE4TinyOS: A reverse engineering methodology for the MDE of TinyOS applications, in: Proceedings of the 15th Conference on Computer Science and Information Systems (FedCSIS 2020), Track on Software and Systems Engineering, IEEE, 2020, pp. 741–757. doi:10.15439/2020F133.
URL https://dx.doi.org/10.15439/2020F133

[32] B. Karaduman, T. Aşıcı, M. Challenger, R. Eslampanah, A cloud and contiki based fire detection system using multi-hop wireless sensor networks, in: Proceedings of the Fourth International Conference on Engineering & MIS 2018, 2018, pp. 1–5. doi:10.1145/3234698.3234764.
URL https://doi.org/10.1145/3234698.3234764

[33] P. P. Ray, A survey on visual programming languages in internet of things, Scientific Programming 2017 (1) (2017) 1–6. doi:10.1155/2017/1231430.

[34] F. Essaadi, Y. Ben Maissa, M. Dahchour, MDE-based languages for wireless sensor networks modeling: A systematic mapping study, in: R. El-Azouzi, D. S. Menasche, E. Sabir, F. De Pellegrini, M. Benjillali (Eds.), Advances in Ubiquitous Networking 2, Lecture Notes in Electrical Engineering, Springer, 2017, pp. 331–346. doi:10.1007/978-981-10-1627-1_26.
URL https://doi.org/10.1007/978-981-10-1627-1_26

[35] F. Paterno, S. Carmen, End-user development for personalizing applications, things, and robots, International Journal of Human-Computer Studies 131 (1) (2019) 120–130. doi:10.1016/j.ijhcs.2019.06.002.

[36] E. Coronado, F. Mastrogiovanni, B. Indurkhya, G. Venture, Visual programming environments for end-user development of intelligent and social robots, a systematic review, Journal of Computer Languages 58 (1) (2020) 1–20. doi:10.1016/j.cola.2020.100970.

[37] B. Akbal-Delibas, P. Boonma, J. Suzuki, Extensible and precise modeling for wireless sensor networks, Lecture Notes in Business Information Processing 20 (2009) 551–562. doi:10.1007/978-3-642-01112-2_55.

[38] M. A. Saad, E. Fehr, N. Kamenzky, J. Schiller, ScatterClipse: A model-driven tool-chain for developing, testing, and prototyping wireless sensor networks, in: 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications, 2008, pp. 871–885, ISSN: 2158-9208. doi:10.1109/ISPA.2008.22.
URL https://doi.org/10.1109/ISPA.2008.22

[39] M. M. R. Mozumdar, F. Gregoretti, L. Lavagno, L. Vanzago, S. Olivieri, A framework for modeling, simulation and automatic code generation of sensor network application, in: Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON'08. 5th Annual IEEE Communications Society Conference on, IEEE, 2008, pp. 515–522.

[40] N. X. Thang, K. Geihs, Model-driven development with optimization of non-functional constraints in sensor network, in: Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications, SESENA '10, ACM, 2010, pp. 61–65. doi:10.1145/1809111.1809128.
URL https://doi.org/10.1145/1809111.1809128

[41] R. Shimizu, K. Tei, Y. Fukazawa, S. Honiden, 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, SESENA '11, ACM, 2011, pp. 31–36. doi:10.1145/1988051.1988058.
URL https://doi.org/10.1145/1988051.1988058

[42] V. Veiset, L. M. Kristensen, Transforming platform independent CPN models into code for the TinyOS platform: A case study of the RPL protocol, in: PNSE+ModPE, 2013, pp. 259–260.

[43] A. Salman, Reducing complexity in developing wireless sensor network systems using model-driven development, phdthesis, University of Salford (2017).
URL http://usir.salford.ac.uk/44127/

[44] T. Rodrigues, P. Dantas, P. F. Pires, L. Pirmez, T. Batista, C. Miceli, A. Zomaya, et al., Model-driven development of wireless sensor network applications, in: Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on, IEEE, 2011, pp. 11–18.

[45] C. Raibulet, F. A. Fontana, M. Zanoni, Model-driven reverse engineering approaches: A systematic literature review, IEEE Access 5 (2017) 14516–14542. doi:10.1109/ACCESS.2017.2733518.

[46] H. Brunelire, J. Cabot, G. Dup, F. Madiot, MoDisco: A model driven reverse engineering framework, Information and Software Technology 56 (8) (2014) 1012–1032. doi:10.1016/j.infsof.2014.04.007.

[47] L. Favre, L. Martinez, C. Pereira, MDA-based reverse engineering of object oriented code, in: T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, R. Ukor (Eds.), Enterprise, Business-Process and Information Systems Modeling, Lecture Notes in Business Information Processing, Springer, 2009, pp. 251–263. doi:10.1007/978-3-642-01862-6_21.
URL https://doi.org/10.1007/978-3-642-01862-6_21

[48] F. Barbier, S. Eveillard, K. Youbi, O. Guitton, A. Perrier, E. Cariou, Model-driven reverse engineering of cobol-based applications, in: Information Systems Transformation, Elsevier, 2010, pp. 283–299.

[49] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, J. Perronnet, A model driven reverse engineering framework for extracting business rules out of a java application, in: A. Bikakis, A. Giurca (Eds.), Rules on the Web: Research and Applications, Lecture Notes in Computer Science, Springer, 2012, pp. 17–31. doi:10.1007/978-3-642-32689-9_3.
URL https://doi.org/10.1007/978-3-642-32689-9_3

[50] O. Sanchez Ramon, J. Sánchez Cuadrado, J. Garcia Molina, Model-driven reverse engineering of legacy graphical user interfaces, Automated Software Engineering 21 (2) (2014) 147–186. doi:10.1007/s10515-013-0130-2.

[51] I. Comyn-Wattiau, J. Akoka, Model driven reverse engineering of NoSQL property graph databases: The case of neo4j, in: 2017 IEEE International Conference on Big Data (Big Data), 2017, pp. 453–458. doi:10.1109/BigData.2017.8257957.
URL https://doi.org/10.1109/BigData.2017.8257957

[52] J. Sánchez Cuadrado, E. Guerra, J. de Lara, Reverse engineering of model transformations for reusability, in: D. Di Ruscio, D. Varr (Eds.), Theory and Practice of Model Transformations, Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 186–201. doi:10.1007/978-3-319-08789-4_14.

[53] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, H. Mei, Towards automatic model synchronization from model transformations, in: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007, pp. 164–173.

[54] K. Vanherpen, J. Denil, H. Vangheluwe, P. De Meulenaere, Model transformations for round-trip engineering in control deployment co-design., in: SpringSim (TMS-DEVS), 2015, pp. 55–62.

[55] J. Akoka, I. Comyn-Wattiau, Roundtrip engineering of nosql databases, Enterprise Modelling and Information Systems Architectures (EMISAJ) 13 (2018) 281–292.

[56] A. H. Eden, E. Gasparis, J. Nicholson, R. Kazman, Round-trip engineering with the two-tier programming toolkit, Software Quality Journal 26 (2) (2018) 249–271.

[57] T. Buchmann, B. Westfechtel, Towards incremental round-trip engineering using model transformations, in: 2013 39th Euromicro Conference on Software Engineering and Advanced Applications, IEEE, 2013, pp. 130–133.

[58] B. Hailpern, P. Tarr, Model-driven development: The good, the bad, and the ugly, IBM systems journal 45 (3) (2006) 451–461.

[59] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. K. Selim, E. Syriani, M. Wimmer, Model transformation intents and their properties, Software & Systems Modeling 15 (3) (2016) 647–684. doi:10.1007/s10270-014-0429-x.

[60] E. Syriani, L. Luhunu, H. A. Sahraoui, Systematic mapping study of template-based code generation, Computer Languages, Systems & Structures 52 (1) (2018) 43–62. doi:10.1016/j.cl.2017.11.003.

[61] H. M. Marah, R. Eslampanah, M. Challenger, DSML4TinyOS: Code Generation for Wireless Devices, in: ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS), Model-Driven Engineering for the Internet-of-Things (MDE4IoT), 2018, pp. 509–514.

[62] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, D. Culler, The nesC language: A holistic approach to networked embedded systems, Acm Sigplan Notices 38 (5) (2003) 1–11.

[63] T. Parr, K. Fisher, LL(*): The foundation of the ANTLR parser generator, in: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, ACM, 2011, pp. 425–436, event-place: San Jose, California, USA. doi:10.1145/1993498.1993548.
URL https://doi.org/10.1145/1993498.1993548

[64] T. Parr, The Definitive ANTLR 4 Reference, 2nd Edition, Pragmatic Bookshelf, 2013.

[65] TinyOS_Github_Repository, TinyOS GitHub Application Repository (2017).
URL https://github.com/tinyos/tinyos-main/tree/master/apps

[66] M. Challenger, G. Kardas, B. Tekinerdogan, A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems, Software Quality Journal 24 (3) (2016) 755–795. doi:10.1007/s11219-015-9291-5.

[67] G. Kardas, B. T. Tezel, M. Challenger, Domain-specific modelling language for belief-desire-intention software agents, IET Software 12 (4) (2018) 356–364. doi:10.1049/iet-sen.2017.0094.

[68] S. Arslan, G. Kardas, Dsml4dt: A domain-specific modeling language for device tree software, Computers in Industry 115 (1) (2020) 1–13. doi:10.1016/j.compind.2019.103179.

[69] O. F. Alaca, B. T. Tezel, M. Challenger, M. Goulao, V. Amaral, G. Kardas, Agentdsm-eval: A framework for the evaluation of domain-specific modeling languages for multi-agent systems, Computer Standards & Interfaces 76 (1) (2021) 1–20. doi:10.1016/j.csi.2021.103513.

[70] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[71] RTE_Github_Repository, Round-trip engineering for tinyos applications (2021). URL https://github.com/husseinmarah/RTE

[72] F. Ciccozzi, I. Crnkovic, D. Di Ruscio, I. Malavolta, P. Pelliccione, R. Spalazzese, Model-driven engineering for mission-critical iot systems, IEEE Software 34 (1) (2017) 46–53. doi:10.1109/MS.2017.1.