Proceedings of the Federated Conference on Computer Science and Information Systems pp. 741–750
DOI: 10.15439/2020F133
ISSN 2300-5963 ACSIS, Vol. 21

RE4TinyOS: A Reverse Engineering Methodology for the MDE of TinyOS Applications

Hussein M. Marah
International Computer Institute
Ege University, Izmir, Turkey
hussein.marah@gmail.com

Moharram Challenger
Department of Computer Science
Univeristy of Antwerp and Flanders Make, Belgium
moharram.challenger@uantwerpen.be

Geylani Kardas
International Computer Institute
Ege University, Izmir, Turkey
geylani.kardas@ege.edu.tr

***Abstract*—In this paper, we introduce a tool-supported reverse engineering methodology, called RE4TinyOS to create or update application models from TinyOS programs for the construction of Wireless Sensor Networks. Integrating with an existing model-driven engineering (MDE) environment, use of RE4TinyOS enables the model-code synchronization where any modification made in the TinyOS application code can be reflected into the application model and vice versa. Conducted case studies exemplified this model-code synchronization as well as the capability of creating application models completely from already existing TinyOS applications without models, which is crucial to integrate the implementations of the third party TinyOS applications into the MDE processes. Evaluation results showed that RE4TinyOS succeeded in the reverse engineering of all main parts of two well-known 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, Reverse Engineering, Wireless Sensor Network, TinyOS, RE4TinyOS.**

I. Introduction

WIRELESS Sensor Networks (WSN) have gained significant popularity and implemented in different areas (e.g. health systems, field monitoring, transportation, military applications and environmental sensing) to control both the status of physical objects and the surrounding circumstances like sound, pressure, vibration, light, temperature, and motion according to the type of the sensors used in the network [1]. WSNs use low-power micro-controllers and devices due to the power consumption constraints that must be adhered to.

One of the widely used operating systems for WSNs is TinyOS [2]. 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 lower-power wireless devices [2][3]. 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 [3]. Adoption to this language may be difficult and again time-consuming for the programmers.

As successfully applied in many other domains, model-driven engineering (MDE) can provide a convenient way of developing TinyOS applications for WSNs by leveraging the abstraction level before delving into programming with nesC. Within this context, in our previous work [4], we introduced the use of a domain-specific modeling language (DSML), called DSML4TinyOS, for the MDE of TinyOS applications. A metamodel for TinyOS was derived and a graphical modeling syntax was formalized from this metamodel to lead modeling TinyOS applications. nesC code of the modeled applications can be automatically generated with the model-to-code transformations again defined in DSML4TinyOS. However, this mechanism lacks the synchronization between a TinyOS application model and the generated code when any change is made in this code. Mostly, the auto-generated code is modified to completely meet with the requirements of the TinyOS application. Furthermore, the application may evolve according to changing requirements in the future. After the code modifications are performed, related changes will make models at different levels asynchronous and inconsistent [5]. Thus we need to propagate these changes to the other models and ensure a proper model synchronization [6]. In order to provide this synchronization which is missing in the MDE of TinyOS applications, in this paper, we introduce a tool-supported reverse engineering methodology, called RE4TinyOS. 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 DSML4TinyOS language to construct a complete model-driven roundtrip engineering [7] process for TinyOS applications. As depicted in Figure 1, evolution of the TinyOS models can be managed within this roundtrip MDE process which is a combination of the forward and reverse engineering of TinyOS models. TinyOS models can be created with using DSML4TinyOS language and the corresponding TinyOS code can be automatically generated. When this code is modified and becomes TinyOS code', RE4TinyOS reverse engineering methodology can be applied on this modified code to retrieve the corresponding modified model (still an instance of TinyOS metamodel) which properly reflects the changes in the appli-

cation code.

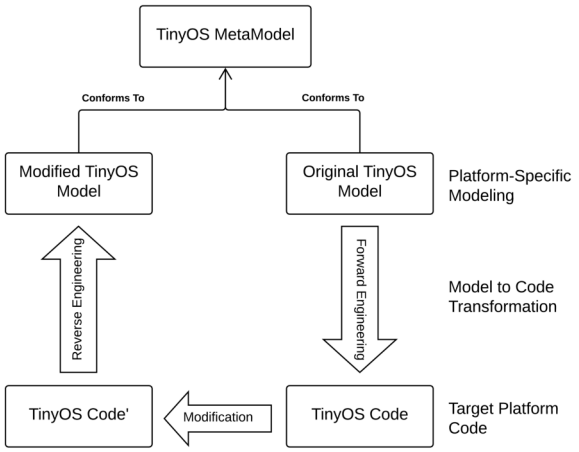

Fig. 1: Forward and reverse engineering for TinyOS applications

The remainder of the paper is organized as follows: Section 2 discusses the related work in this area. RE4TinyOS methodology and supporting parser and interpreter tools are introduced in Section 3. The usability of the methodology is demonstrated and evaluated in Section 4. Section 5 concludes the paper.

II. Related Work

In recent years, there is a significant interest of the researchers to apply MDE and its techniques for WSN and IoT development. The main goal of applying MDE approach is to facilitate the task of developing, building and deploying different WSN and IoT applications. Malavolta and Muccini [8] and Essaadi et al. [9] present good overviews of applied MDE approaches for this domain.

For example, ScatterClipse, a generative plugin-oriented tool-chain, is proposed in [10] 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. Thang and Geihs [11] 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 [12] 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 [13] brings an MDE approach for prototyping and optimization of WSN applications while Veiset and Kristensen [14] introduce the use of Coloured Petri Net models for generating TinyOS protocol software. Likewise, the use of a domain-specific language (DSL), called SenNet, for WSN application development is proposed in [15] to prepare WSN applications using multi-abstraction levels. Finally, Rodrigues et al. [16] aim at facilitating the development tasks required for Wireless Sensor and Actuator Network (WSAN) applications via 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.

The above mentioned studies provide various noteworthy approaches both for modeling WSN applications in different abstraction levels and code generation for WSN development, mostly assisted with tools. Moreover, some of them specifically support the development of TinyOS applications within the MDE perspective. However, 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 model and code does not exist. We believe that RE4TinyOS reverse engineering methodology, introduced in this paper, may contribute to these efforts by filling this gap as well as supporting the roundtrip engineering of TinyOS WSN applications within a toolchain consists of both generating code from TinyOS application models and retrieving models from the existing codes automatically.

Taking into consideration of applying reverse engineering in the context of MDE, various adoptions exist for different domains as surveyed in [17]. Perhaps one of the most popular approaches is MoDisco [18], 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. [19] 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 [20], model discovery from Java source code to extract the business rules [21], generating GUI models of the explicit layouts especially for Java Swing user interfaces [22], restoring extended entity-relationship schema from NoSQL property graph databases [23] and even achieving reusable and evolvable model transformations [24]. However, reverse engineering of WSN applications is not addressed again in all these studies.

III. RE4TinyOS Methodology

Figure 2 represents the use of RE4TinyOS methodology for the MDE-based reverse engineering of WSN applications running on TinyOS. The figure gives a straightforward depiction of how reverse engineering works according to MDE concepts to convert the TinyOS code to a TinyOS model for any application.

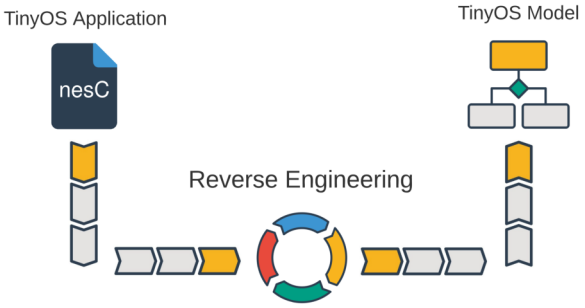

Fig. 2: Overview of the proposed reverse engineering approach

TinyOS applications are written in a special programming language, called nesC [25] 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 [25]. 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.

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. The language recognizer is used for different purposes like building a compiler or maybe analyze parts of code to perform some operations [26] [27]. 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.

In this study, ANTLR was chosen to build the TinyOS parser. ANTLR (ANother Tool for Language Recognition) is a well-known computer-based language recognition tool, or more specifically a parser generator [28] [26] [27].

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 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). The next listing (Coding 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.

Coding 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 model and the second for the configuration that ends with “EOF” condition. The “componentDeclaration” line includes two main parts which are “moduleDeclaration” and “configurationDeclaration” respectively. The separator character ‘|’ 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 'l' 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 [4], 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 using this parser and benefiting 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 [27]. In our case, the target language is Java. An overview of the constructed TinyOS parser is shown in Figure 3.

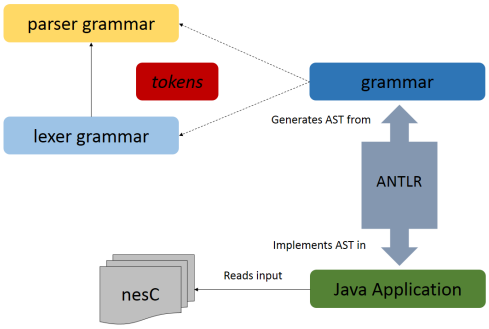

Fig. 3: Parsing process for TinyOS applications

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" [27] 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 4).

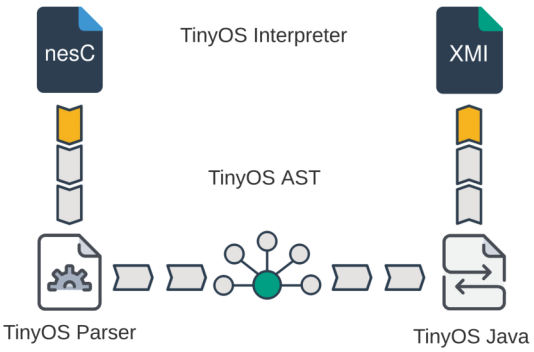

Fig. 4: TinyOS Interpreter structure

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.

DSML4TinyOS is a tool-supported DSML which facilitates the development of TinyOS applications according to MDE principles and techniques. The tool enables TinyOS developers to develop applications from scratch by visually modelling these applications and generate code as the final artefact. DSML4TinyOS uses the TinyOS metamodel introduced in [4] as the abstract syntax. It has an EMF-based graphical syntax and the graphical modeling environment required for creating DSML4TinyOS models according to DSML4TinyOS syntax and semantics definitions. DSML4TinyOS modeling environment (see Figure 5) was built on the widely used Sirius platform. 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. DSML4TinyOS benefits from the features of Acceleo code generator to parse instance TinyOS models and create the templates of the implementation files.

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 from the existing implementations can also be shown and processed again inside DSML4TinyOS tool. Hence, the synchronization of the system model and the

Concept	Notation	Concept	Notation
Mote		Application	
Module		Configuration	
Components		Interface	
Module_Signature		Configuration_Signature	
Component		nesC	
Function		Wiring	
Event		Command	
Module_Implementation		Configuration_Implementation	
Task		IncludeDeclaration	
		Helper_Function	

Table. 1: DSML4TinyOS concrete syntax notations

existing implementation is realized in case of any modification made on the model or the code.

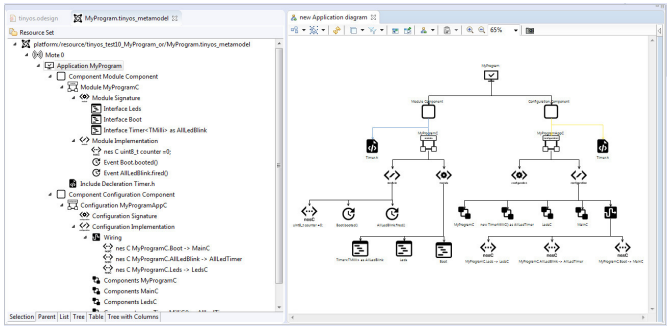

Fig. 5: DSML4TinyOS graphical modeling environment

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.

IV. Case Studies

In order to demonstrate and evaluate the usability of RE4TinyOS methodology and its tool, a multi-case evaluation study has been performed. The first 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 roundtrip engineering process. The remaining two case studies consider the usability of RE4TinyOS methodology within the scope of the reverse engineering of already existing TinyOS applications publicly available from the official TinyOS repository in Github.

A. Supporting model - code synchronization

This section discusses the MDE of an application for a TinyOS mote, 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 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.led0On()” one by one corresponding to each “Counter” value.

The Above described TinyOS application was modeled graphically with using DSML4TinyOS and nesC code of this application was automatically generated.

Coding 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();}
   }
}
```

Coding 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;
}
```

The previous two listings include the code fragment generated from this model for the module part (Coding 2) and the configuration part (Coding 3) of the TinyOS application. Also, the Figure 6 shows the model of the MyProgram application (as a DSML4TinyOS instance), the instance model represents the two parts of code 'Module' and 'Configuration' for the application in a single model.

When any change made in the application code, these 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 (Coding 4 and Coding 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.

Coding 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();}
}
```

Coding 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 7) and it was examined that RE4TinyOS

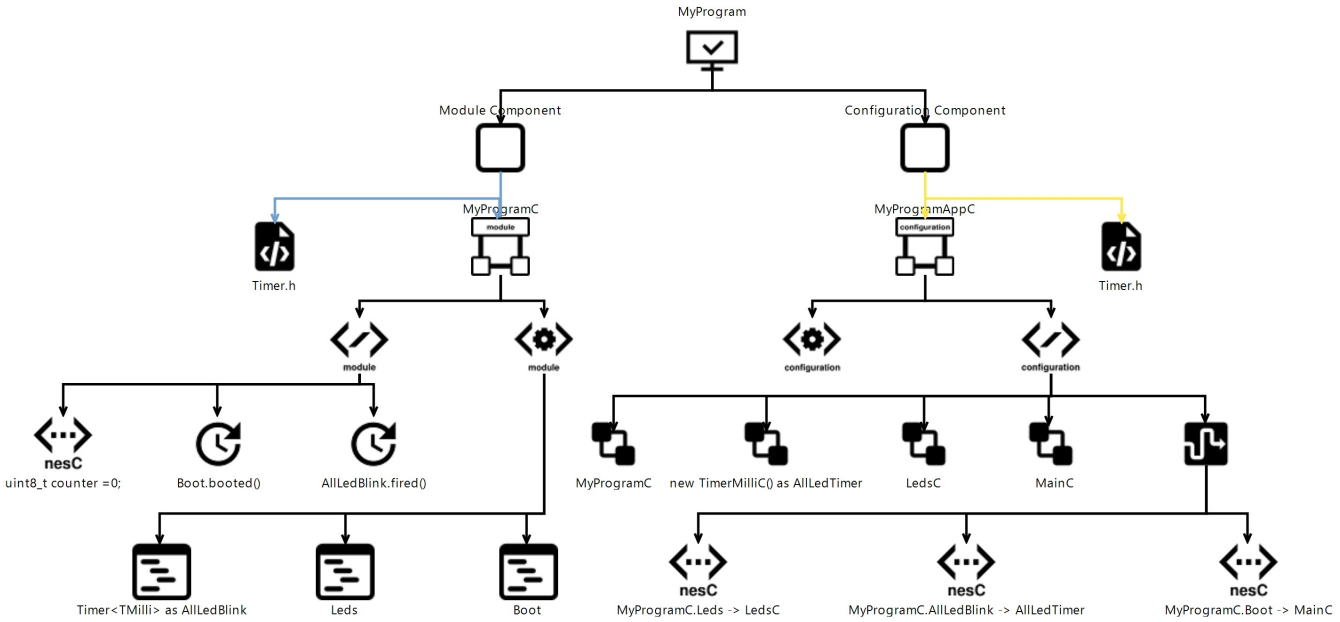

Fig. 6: Graphical model of the original TinyOS application

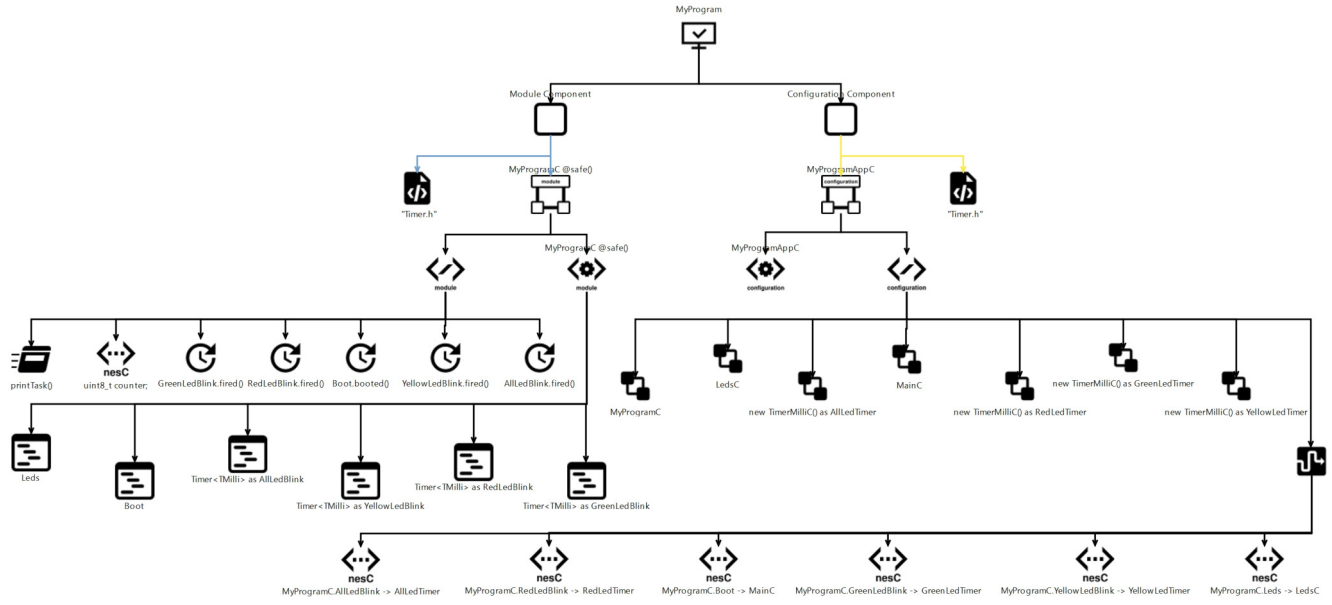

Fig. 7: Graphical model of the modified TinyOS application

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 7, the modifications were seamlessly integrated into the modified and new model with preserving the unchanged model components.

B. Integrating already existing implementations into modeling

Although the previous case study shows how RE4TinyOs tool enables retrieving TinyOS application models from the code and updating the model when the code is modified, we also need to evaluate the capability of creating application models completely from already existing code which is crucial to integrate the implementations of the third party

applications into the MDE processes. In here, already existing code means the application was not previously designed and implemented with using DSML4TinyOS and RE4TinyOS tool chain. Hence, it does not own an application model to be used as an input for further system developments. For the purpose of evaluating this capability of RE4TinyOS, we considered the reverse engineering of two existing TinyOS applications which are well-known and publicly available from the official TinyOS repository in Github. In the following, first these two applications and the generated models are introduced briefly, then the qualitative assessment results are discussed.

1) AntiTheft WSN: AntiTheft is an 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:

- Alerting via turning on the light (e.g. a red LED)
- Also making a beep sound
- Reporting to the other nodes within the range by broadcasting messages, and nodes will also turn on their red LEDs.
- Reporting to a central node using a multi-hop routing algorithm.

The complete nesC code of the AntiTheft application, accessed from TinyOS Github repository [29], was given as input to RE4TinyOS tool and the serialized model file was generated. When this file was opened in DSML4TinyOS modeling environment, the graphical model of the application was shown successfully (see figure 8). Parts of the TinyOS application including components, interfaces, commands, and events are now represented in DSML4TinyOS notation as the result of the applied reverse engineering methodology.

2) Sense WSN: The Sense is another application also available in the main TinyOS Github repository. As its name denotes, it 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. Similar to AntiTheft application, the complete code of the Sense application achieved from the Github repository [30] was processed by RE4TinyOS tool and the model of the application was generated without any error. Figure 9 shows this model opened in the DSML4TinyOS modeling environment.

C. Discussion

First case study, conducted for the MDE of a TinyOS LED display application, demonstrated the use of RE4TinyOS methodology and its tool to support the model-code synchronization where the application model is kept up-to-date in each modification made in the application code. The case study also exemplified the use of DSML4TinyOS and RE4TinyOS tool chain leading the roundtrip engineering of the TinyOs applications.

The remaining case studies enabled the assessment of the proposed reverse engineering methodology brought by RE4TinyOS especially for the already existing TinyOS applications which were not previously designed and implemented with using DSML4TinyOS and/or RE4TinyOS tools. Moreover, the fact that the code of these applications are publicly available in TinyOS Github and written by other developers, contributed to the objectiveness of the performed evaluation.

When the complete code of both Anti-Theft and Sense applications, which are ready to be executed, was given as input to RE4TinyOS, the embedded parser of the RE4TinyOS was able to automatically generate serialized versions of the TinyOS software models of these applications, and the produced models were processed and successfully opened in the DSML4TinyOS IDE. This also confirms that, if needed, RE4TinyOS tool can also be used independently from the MDE tool chain, 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 and then reflect these changes to the exact implementations.

Finally, it is worth indicating that RE4TinyOS succeeded in retrieving the models for all main parts of AntiTheft and Sense applications, including "event", "task", "component", "interface", "Command", "Helper-function", and "Wiring" (see figures 8 and 9). Although, block structures of the application events were also retrieved, internal specifications of some of these events could not be fully represented in the output model since corresponding meta-entities and relations are missing in the TinyOS metamodel currently used by the RE4TinyOS parser. However, these unconverted specifications were still kept 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.

V. Conclusion

A reverse engineering methodology and its tool, both called RE4TinyOS, have been introduced in this paper. 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. Conducted case studies 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. However, the achieved results also showed that some of the internal TinyOS event specifications of these existing applications can not be represented in the newly generated models since corresponding meta-entities are missing in the

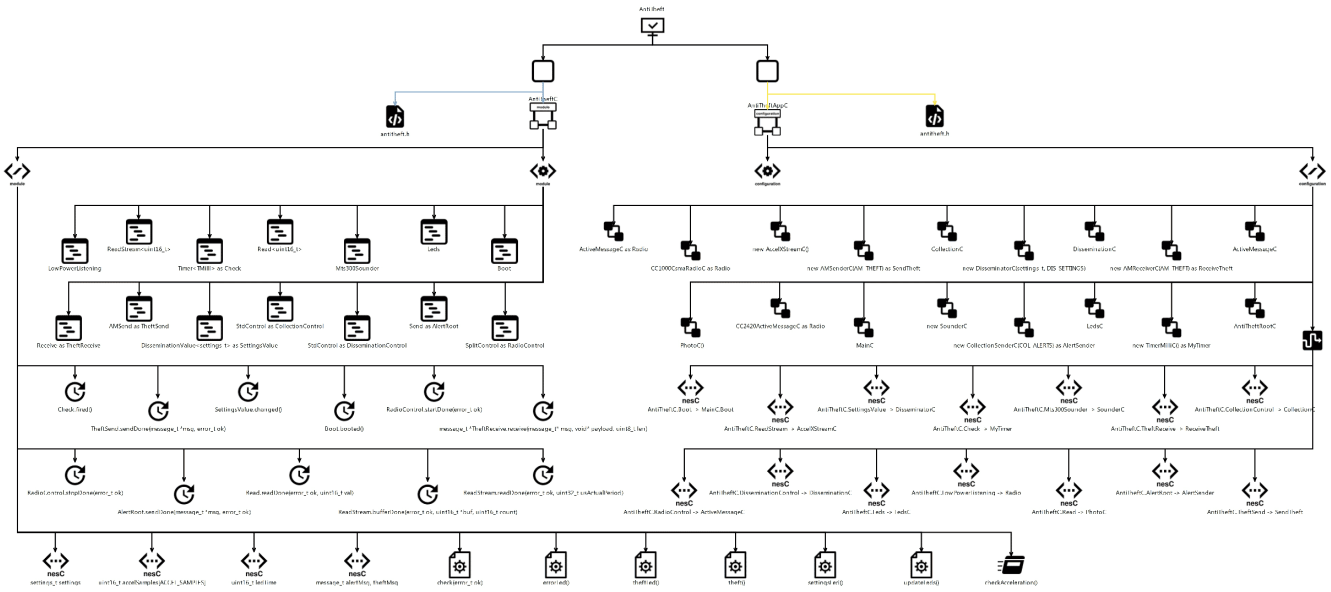

Fig. 8: Graphical model of the AntiTheft application

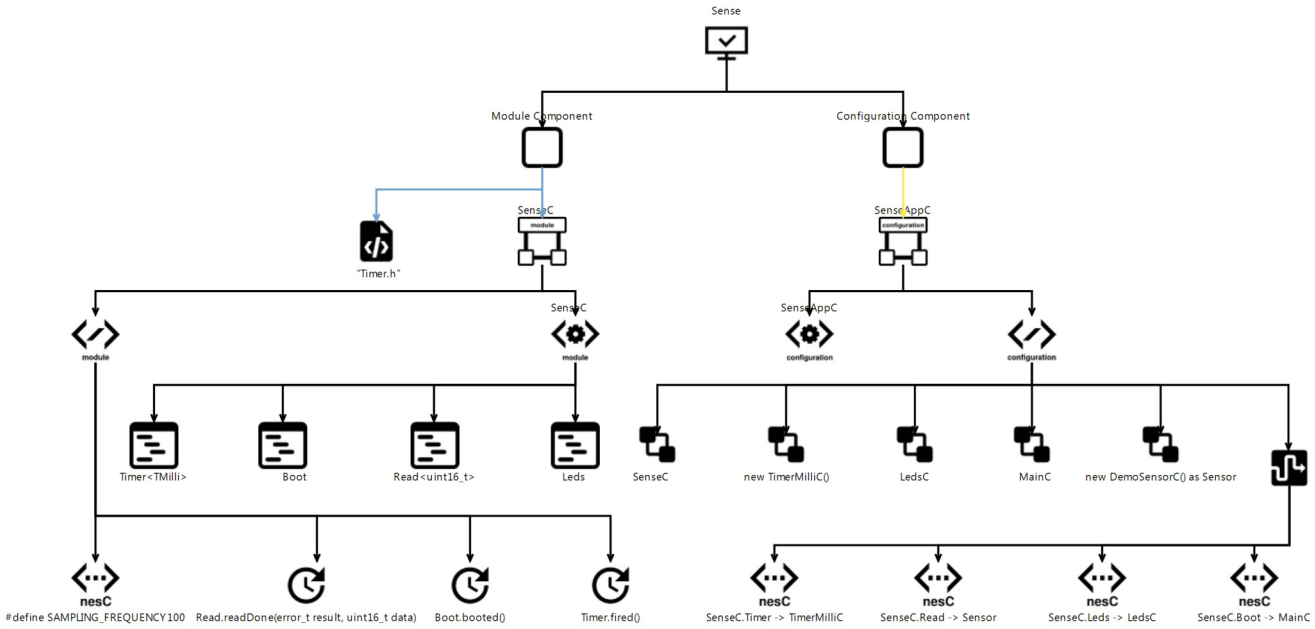

Fig. 9: Graphical model of the Sense application

current TinyOS metamodel used by the RE4TinyOS parser. In our future work, we aim at first extending this metamodel to cover all event internals while keeping the abstraction level and then improving the parser features with the utilization of this new metamodel.

ACKNOWLEDGMENT

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] M. A. Matin and M. Islam, "Overview of wireless sensor network," *Wireless Sensor Networks-Technology and Protocols*, pp. 1–3, 2012.

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

[3] P. Levis and D. Gay, *TinyOS Programming*. Cambridge University Press, 2009.

[4] H. M. Marah, R. Eslampanah, and 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.

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

[6] H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronization," *Software & Systems Modeling*, vol. 8, no. 1, pp. 21–43, 2009. doi: 10.1007/s10270-008-0089-9

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

[8] I. Malavolta and 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: https://doi.org/10.1109/SEAA.2014.61

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

[10] M. A. Saad, E. Fehr, N. Kamenzky, and 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: https://doi.org/10.1109/ISPA.2008.22

[11] N. X. Thang and 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*, ser. SESENA '10. ACM, 2010, pp. 61–65. doi: https://doi.org/10.1145/1809111.1809128

[12] K. Doddapaneni, E. Ever, O. Gemikonakli, I. Malavolta, L. Mostarda, and 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*, ser. SESENA '12. IEEE Press, 2012, pp. 1–7. doi: https://doi.org/10.1109/SESENA.2012.6225729

[13] R. Shimizu, K. Tei, Y. Fukazawa, and 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*, ser. SESENA '11. ACM, 2011, pp. 31–36. doi: https://doi.org/10.1145/1988051.1988058

[14] V. Veiset and 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.

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

[16] T. Rodrigues, F. C. Delicato, T. Batista, P. F. Pires, and L. Pirmez, "An approach based on the domain perspective to develop WSAN applications," *Software & Systems Modeling*, vol. 16, no. 4, pp. 949–977, 2017. doi: 10.1007/s10270-015-0498-5

[17] C. Raibulet, F. A. Fontana, and M. Zanoni, "Model-driven reverse engineering approaches: A systematic literature review," *IEEE Access*, vol. 5, pp. 14 516–14 542, 2017. doi: 10.1109/ACCESS.2017.2733518

[18] H. Brunelire, J. Cabot, G. Dup, and F. Madiot, "MoDisco: A model driven reverse engineering framework," *Information and Software Technology*, vol. 56, no. 8, pp. 1012–1032, 2014. doi: 10.1016/j.infsof.2014.04.007

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

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

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

[22] . Sanchez Ramon, J. Sanchez Cuadrado, and J. Garcia Molina, "Model-driven reverse engineering of legacy graphical user interfaces," *Automated Software Engineering*, vol. 21, no. 2, pp. 147–186, 2014. doi: 10.1007/s10515-013-0130-2

[23] I. Comyn-Wattiau and 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: https://doi.org/10.1109/BigData.2017.8257957

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

[25] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," *Acm Sigplan Notices*, vol. 38, no. 5, pp. 1–11, 2003.

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

[27] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.

[28] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[29] TinyOS_Github_Repository, "Tinyos antitheft application," 2013. doi: https://github.com/tinyos/tinyos-main/tree/master/apps/AntiTheft

[30] TinyoS_Github_Repository, "Tinyos sense application," 2013. doi: https://github.com/tinyos/tinyos-main/tree/master/apps/Sense