Towards a Model-Based Development Approach for Wireless Sensor-Actuator Network Protocols

Position paper

A. Ajith Kumar S.
Bergen University College, Norway
University of Agder, Norway
aaks@hib.no

Kent I. F. Simonsen
Bergen University College, Norway
Danish Technical University, Denmark
kifs@hib.no

ABSTRACT

Model-Driven Software Engineering (MDSE) is a promising approach for the development of applications, and has been well adopted in the embedded applications domain in recent years. Wireless Sensor Actuator Networks consisting of resource constrained hardware and platform-specific operating system is one application area where the advantages of MDSE can be exploited. Code-generation is an integral part of MDSE, and using a multi-platform code generator as a part of the approach has several advantages. Due to the automated code-generation, it is possible to obtain time reduction and prevent errors induced due to manual translations. With the use of formal semantics in the modeling approach, we can further ensure the correctness of the source model by means of verification. Also, with the use of network simulators and formal modeling tools, we obtain a verified and validated model to be used as a basis for code-generation. The aim is to build protocols with shorter design to implementation time and efforts, along with higher confidence in the protocol designed.

Keywords

Model-Driven Software Engineering (MDSE), Wireless Sensor-Actuator Networks (WSAN), Code Generation, Validation, Verification, Simulation, Embedded Software

1. INTRODUCTION

Wireless Sensor-Actuator Networks (WSANs) consists of a network-connected sensors and actuators working towards a specific mission. It is a branch of the existing Wireless Sensor Network (WSN) systems. Sensors and actuators in WSAN are resource constrained small devices usually powered by batteries. WSAN has several application domains such as process automation and factory automation, and is thus widely applicable in an industrial setting. One important setting in which WSAN is applicable is the control-loop of automation processes. Systems with control-loops have stringent requirements, and these applications are often safety-critical. This implies that it is important to have a sound design methodology for developing software solutions to be deployed on the sensors and actuators. Quite often the design methodology includes modeling of the protocols. Later these models are manually converted to simulation code and further analyzed. Some development approaches also includes model-checking to check for correctness. As the last step, models are transformed to the implementation platform code. This approach towards design of WSAN solutions can be combined with existing software engineering approaches, to strengthen the reliability of the software generated and to reduce the time for development.

Model-Driven Software Engineering (MDSE) is one such approach that has long been seen as a prominent approach for software engineering. MDSE uses models as primary artifacts. MDSE is an extensively used methodology across several domains for development of applications. Advantages of MDSE approach include shorter time from design to implementation, verified and validated models used for automatic code generation. In an industrial setting, continuous work is done in reducing the cost of developing software and the time required, and MDSE works towards this. In MDSE approaches, the initial model is an abstract and platform independent representation of the protocol. This abstraction allows the designers to focus on creating a model with proper functioning. Combining this abstraction step with formal approaches allows further improving the verification and validation process. The abstract models can be model-checked for verification and can be further simulated to obtain initial performance assessment results. This minimizes the possibility of errors in the code generated to a further extent.

One tool that allows model-checking and simulation is Coloured Petri Nets (CPN) Tools [5]. It is based on the expressive language CPN combined with the Standard ML programming language. CPN has been previously used for modeling and verification of network protocols [1]. This combined with the PetriCode [14] tool, forms a complete MDSE approach for development of network protocols and can be applied to develop WSAN protocols as well. PetriCode is a code generation tool that takes platform-independent CPN models as input and produces platform-specific code for various platforms like Java, Clojure and Groovy. The aim of this work is to extend PetriCode to support platform-specific code generation for sensor network

CyPhy '14, April 14-17 2014, Berlin, Germany

ACM 978-1-4503-2871-5/14/04 ...$15.00.
http://dx.doi.org/10.1145/2593458.2593465

platform such as the TinyOS [9] and for sensor network simulators. Using a CPN model as a starting point allows us to base our implementations on verified and functionally correct model, giving confidence in the correctness of the generated code. In this article, we mainly focus on providing an MDSE approach for protocol development for WSAN using CPN models and the PetriCode tool. Generation for disparate platforms (MiXiM [6],TinyOS) is still challenging because the model must then support several programming models in the implementations. For the model, however this can be seen as an advantage as it forces the model to focus on the logical operation of the protocol and not include implementations details. As a case study we use the GinMAC [15] protocol specification. GinMAC protocol was designed for WSAN, with strict packet delay requirements.

2. RELATED WORK

Model-driven software engineering has been used in the WSN domain for a while now [10, 18]. There have been several work proposing various frameworks for rapid development of WSN protocols, dominantly based on Doman Specific Modeling Languages (DSML) [2, 4] or the Unified Modeling Language (UML) [10, 18, 13]. In [11], a design framework is proposed which facilitates behavior simulation, and multi-platform code-generation. It requires multiple steps for platform-specific code generation and the simulation done is very basic. In [12], an architectural framework, Architecture for Wireless Sensor and Actuator Network (ArchWiSeN) is proposed. This architecture is based on UML diagrams as platform independent high level models and consider TinyOS platform for code generation and simulation was performed on TOSSIM. Moppet [2] is an MDSE based method that uses feature modeling, in-tool performance estimator and code generation for TinyOS. In [4] the authors concentrated mainly on the modeling aspects proposing an architecture consisting of separate modeling languages for environment, software architecture and node modeling. Mapping is used to create relation between these modeling languages. They also propose code generation as possible future work based on existing model to text generators. In [16], a model-driven development approach is proposed based on a Domain Specific Language (DSL) for WSN. They perform a model-to-model transformation from the initial platform independent model to a platform specific model. This platform specific model (TinyOS) is then used for code generation. In [17] a code-generation technique for nesC was provided. The article [17], focussed only on the code-generation part and specifically the TinyOS platform, thus creating platform-specific code generation software. The key differences between the existing and our MDSE approach are: firstly we provide a formal platform to begin modeling with, which also provides simulation and state-space analysis possibilities, secondly we provide conversion possibility to event-based network simulator (MiXiM [6]) that has been used in WSN research. These features are provided along with code-generation for a sensor specific platform like TinyOS.

3. OVERVIEW OF APPROACH

In this section, we provide an overview of our proposed MDSE approach for protocol design and implementation. The approach is illustrated in figure 1. The starting point of the approach is an abstract model of the given protocol. For modeling purposes, we use CPN Tools [5]. Along with modeling and verification, CPN Tools allows for initial simulation. Using CPN Tools, developers can create a formal executable model of a protocol. Developers can also perform behavioral and functional verification, state-space analysis, and initial simulation. Based on the analysis, developers can verify and validate the given model, based on its requirements specification. In the next step, the refined CPN model is converted to code models using the PetriCode tool [14]. PetriCode is a code generation tool that is designed to automatically generate implementations of network protocols for various platforms. PetriCode requires the models to be annotated with *code generation pragmatics*. These pragmatics are structural annotations on CPN models that are bound to *code-generation templates* via *template bindings*. In PetriCode the pragmatics and the template bindings as well as the structure of the CPN model is used to guide code generation without needing to translate or interpret the ML code of any given CPN model. Using the PetriCode tool, developers can automatically generate code for model-checkers, network simulators, and hardware platforms. The implementation generated for the simulation and model checking platforms are used to perform further analysis. This can further strengthen the developer's confidence in the correctness of the final implementation.

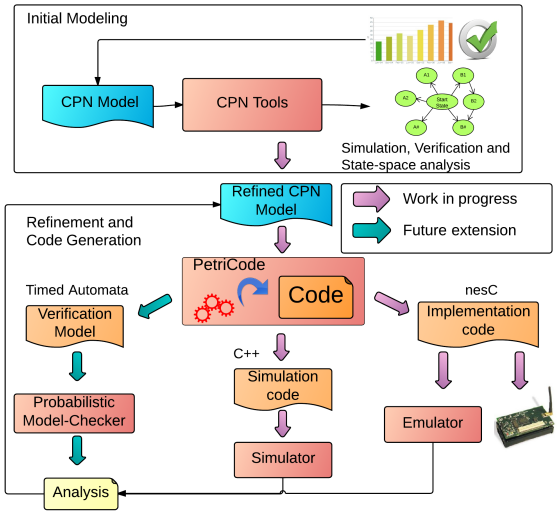

Figure 1: Model-based Development Approach

For the initial work, we have selected certain tools namely, Uppaal [3] /PRISM [7] for model-checking, Omnet-Mixim [6] for network simulation, and TinyOS [9] for platform-specific simulation and implementation. All these have been successfully used in a number of application case studies. Using this approach, before the final implementation, the model can be further tested using the tools listed above. Uppaal/PRISM are powerful probabilistic model-checkers that allow for further exhaustive behavioral and functional analysis. Given the stochastic nature of the wireless channel used for communication in WSAN, it is essential to perform a probabilistic study for the model-checking procedure. Omnet-Mixim is a discrete event simulator that allows for network simulations using pre-defined wireless channel models and also has rich protocol library thus provides necessary infrastructure. From the results obtained by

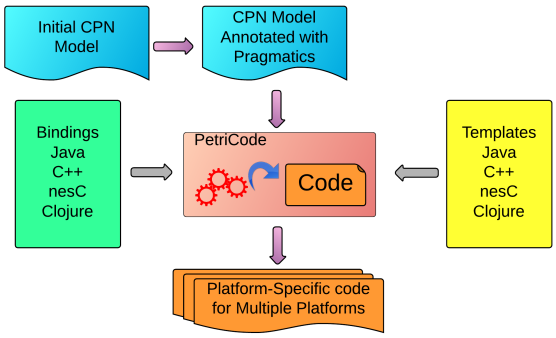

Figure 2: Code generation architecture

model-checking and simulating the model, the CPN model can be further refined to eliminate possible design flaws and to provide high quality software for the implementation. The event based sensor network platform TinyOS is used for hardware implementation code generation. TOSSIM [8] is the emulator for TinyOS platform, thus nesC code can be simulated.

The flexibility of PetriCode tool that makes it possible to generate code for different platforms is an important advantage over most other existing tools. Developers can create their own templates and bindings for a platform based on the pragmatics defined by the tool. For the design, analysis, and implementation of protocols, platform specific codes in C++ for simulation, Timed Automata for model checking, nesC for TinyOS and TOSSIM is required. The challenge in the current work is to be able to extract multi-platform representations out of a single CPN model. Given that different platforms such as TinyOS which has a event-based code structure, Java and C++ have object-oriented structure, it is essential to exploit the similarity in them to be able to generate code from a single model.

4. MODELS AND CODE GENERATION

An important part of MDSE is code-generation. It is essential towards reducing the errors in the coding process and reducing time required to generate code from design. In our framework, we use PetriCode tool [14] to be able to generate platform-specific code and simulation tool code. Our aim is to extend the PetriCode tool to provide implementation and simulation code. The code generation framework is shown in figure 2. The code generation is carried out in four steps, detailed below.

1. Create a CPN model of the protocol for which an implementation is to be obtained.
2. Annotate modeling elements with pragmatics to facilitate code generation. PetriCode models have an explicit control-flow path that is defined by <<Id>> pragmatics. This allows PetriCode to use the CPN model structure to generate code for programming languages of several paradigms
3. Create bindings and templates for the platform to which the resulting code has to be generated.
4. Use PetriCode to generate code using proper bindings.

Multiple steps are involved in going from CPN model to a platform-specific code for which [14] can be referred. In this article, we use PetriCode tool and extend its functionalities to support WSAN protocol development. The bindings and the templates required for the code-generation for every platform has to be specified. These templates and bindings use the pragmatics to interpret the CPN model and create the corresponding implementation code. The PetriCode tool already has predefined bindings for Java, Clojure and Groovy. A sample CPN model for a sensor looks like the one in figure 3. This is the top most level of the layered CPN model, defining a sensor node and is annotated with pragmatics <<Principal()>> and <<Channel()>>. The top most level depicts the layered sensor model with a radio component at the lowest layer (OSI layering), connecting the sensor with the communication channel. Above it, is the Medium Access Control (MAC) layer which handles all the communication through and to the sensor. The application and network part are combined in this model which represents application support and routing decision support protocols. A level lower where the module is defined in detail, inside the Medium Access Control (MAC) module we can see <<Service()>> pragmatic as shown in figure 4. The figure depicts two services being handled in the MAC module: radio packet handler and upper layer packet handler. The module also consists of a variable declaration *SingleSlotBuffer* used by the upper layer packet handler service.

Similar to relating programming concepts between different platforms it is a challenge to relate pragmatics to different code structures across languages. A view of the differences in concepts between the initial pragmatics annotations and platform-specific languages is shown in figure 5. The nesC code structure consists of components, configurations, modules and interfaces. Whereas C++ consists of objects, classes, methods and abstract classes. Initially, we map <<Principal()>> pragmatic to objects in C++ and components in nesC. At the lower level, nesC contains of commands and events (synchronous and asynchronous) and C++ consists of methods. We map the <<Service()>> pragmatic to commands and events in nesC, and to methods in C++. PetriCode already consists of Java templates and bindings. We need to create new bindings and templates for C++ and nesC, adhering to their implementation rules. An example C++ code generated from the CPN model defined in figure 3 is shown in listing 1. For this generation, the binding used is shown in listing 2 and the template example for "Declaration" is shown in listing 3. An example nesC code (skeleton) generated for the same CPN model is shown in listing

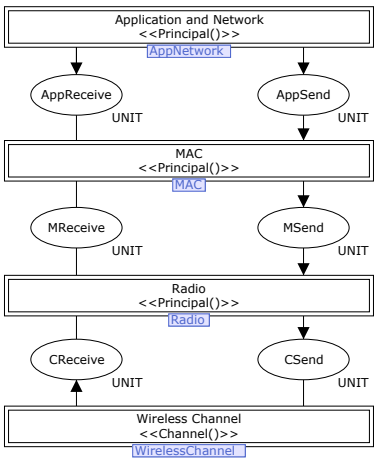

Figure 3: CPN model of a Sensor

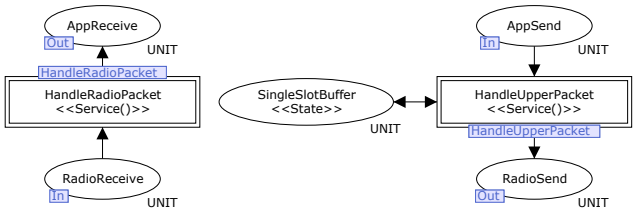

Figure 4: CPN model of the Medium Access Layer

4. The "Declaration" binding is used to generate code for the pragmatic <<State>> which can be seen in figure 4. To obtain the nesC code, we plan to use asynchronous commands and events. An important constituent of a program code block is its control flow. Given the CPN's lack of enforcing a structure particularly to enforce control flow, PetriCode defines certain pragmatics and control flow structure for the <<Service()>> level [14]. Thus at the service level, we need to follow a particular structure that defines the control-flow while transforming the model from an initial CPN model to an annotated PetriCode CPN model. In the current work, we are in the process of creating code-generation for simulation and hardware platforms. Currently an abstract model of the protocol has been created and a C++ conversion (skeleton code) is generated as an example.

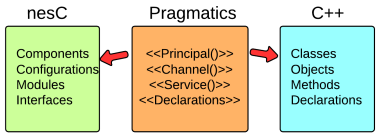

Figure 5: Concept mapping from Pragmatics to nesC and C++

Listing 1: C++ source code file

```
//Mac c++ file
#include <BaseMacLayer.cc>
#include "MAC.h"

Object SingleSlotBuffer;
void MAC::HandleUpperPacket() {
    /*[]*/ /*[]*/
    /*vars: [__TOKEN__:]*/
    Object __TOKEN__ = null;
}
void MAC::HandleRadioPacket() {
    /*[]*/ /*[]*/
    /*vars: [__TOKEN__:]*/
    Object __TOKEN__ = null;
}
```

Listing 2: Template Bindings file

```
//C++.bindings
principal(pragmatic: 'principal',
    template: "./cTmpl/mainClass.tmpl")
service(pragmatic: 'service',
    template: "./cTmpl/externalMethod.tmpl")
DECLARATIONS(pragmatic: '_-DECLARATIONS-_',
    template: './cTmpl/__DECLARATIONS__.tmpl')
```

Listing 3: Template file for declarations binding

```
//__DECLARATIONS__.tmpl
/*vars: ${vars}*/
<%vars.each{%>
<%if( it != '[]' && it != 'msg'){%>Object ${it} = null;
<%}}%>
```

Listing 4: nesC implementation code

```
//MAC Component
module MAC{}
implementation {
Object SingleSlotBuffer;
event void HandleUpperPacket() {
    /*[]*/ /*[]*/ /*vars: [__TOKEN__]*/
    Object __TOKEN__ = null;
}
event void HandleRadioPacket() {
    /*[]*/ /*[]*/ /*vars: [__TOKEN__]*/
    Object __TOKEN__ = null;
    }
}
```

4.1 Current Challenges

The important challenges in the current project using PetriCode for code-generation for multiple platforms are listed below:

1. Finding proper abstractions and abstraction level for the platform-independent models.
2. PetriCode assumes a certain control-flow model in the models. Even though PetriCode has been used to generate code for languages representing different programming paradigms, it may not be entirely trivial to adapt to target platforms and simulators for the embedded domain.
3. Making code-generation as complete as possible with regards to being able to generate most or any protocol with little or no need to create new pragmatics and templates for various target platforms.

5. OUTLOOK

In this article, we have proposed an MDSE approach for designing protocols for WSN and WSAN applications. We carry out high-level abstract modeling in feature rich CPN Tools which also allows for initial verification and simulation unlike other existing tools. We further provide the opportunity to also perform simulation on full-fledged network simulators that focus specifically on network simulations and have been developed over the years to incorporate various wireless features. Thus, initially we extend PetriCode to provide code generation for the network simulator MiXiM and sensor network platform TinyOS in nesC language. The extensions have been partly completed and the work is in progress. We have generated initial skeleton code for C++ and nesC. Given the model of the PetriCode tool, it can also be extended towards providing conversions to other platforms as well as model-checking tools for further verification, essentially probabilistic verification considering the complex nature of wireless channels involved. We can also easily extend to obtain possible code generation to other prominent sensor network platforms namely ContikiOS as well, which is based on the C language. As a case study, we plan to design the GinMAC protocol according to its specifications and generate implementation code.

6. REFERENCES

[1] J. Billington, G. Gallasch, and B. Han. A coloured petri net approach to protocol verification. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 210–290. Springer Berlin Heidelberg, 2004.

[2] P. Boonma and J. Suzuki. Model-driven performance engineering for wireless sensor networks with feature modeling and event calculus. In *Proceedings of the 3rd Workshop on Biologically inspired Algorithms for*

Distributed Systems (BADS), pages 17–24. ACM, 2011.

[3] A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, J. Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In *Proceedings of the 9th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 6919 of *LNCS*, pages 80–96. Springer Berlin Heidelberg, 2011.

[4] 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 3rd International Workshop on Software Engineering for Sensor Network Applications (SESENA)*, pages 1–7, 2012.

[5] K. Jensen, L. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9:213–254, 2007.

[6] A. Köpke, M. Swigulski, K. Wessel, D. Willkomm, P. T. K. Haneveld, T. E. V. Parker, O. W. Visser, H. S. Lichte, and S. Valentin. Simulating wireless and mobile networks in omnet++ the mixim vision. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops (Simutools)*, number 71, pages 1–8, 2008.

[7] M. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0: Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer Berlin Heidelberg, 2011.

[8] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st International Conference on Embedded networked sensor systems (ACM SenSys)*, pages 126–137. ACM, 2003.

[9] 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*, pages 115–148. Springer Berlin Heidelberg, 2005.

[10] F. Losilla, C. V Chicote, B. Alvarez, A. Iborra, and P. Sanchez. Wireless sensor network application development: An architecture-centric MDE approach. In *Software Architecture*, Lecture Notes in Computer Science, pages 179–194. Springer Berlin Heidelberg, 2007.

[11] M. M. R. Mozumdar, F. Gregoretti, L. Lavagno, L. Vanzago, and S. Olivieri. A framework for modeling, simulation and automatic code generation of sensor network application. In *Proceedings of the 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 515–522, 2008.

[12] T. Rodrigues, T. Batista, F. Delicato, P. Pires, and A. Zomaya. Model-driven approach for building efficient wireless sensor and actuator network applications. In *Proceedings of the 4th International Workshop on Software Engineering for Sensor Network Applications(SESENA)*, pages 43–48, 2013.

[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 International Workshop on Software Engineering for Sensor Network Applications (SESENA)*, pages 31–36, 2011.

[14] K. I. F. Simonsen, L. Kristensen, and E. Kindler. Generating protocol software from cpn models annotated with pragmatics. In *Formal Methods: Foundations and Applications*, volume 8195 of *Lecture Notes in Computer Science*, pages 227–242. Springer Berlin Heidelberg, 2013.

[15] P. Suriyachai, J. Brown, and U. Roedig. Time-critical data delivery in wireless sensor networks. In *Proceedings of Distributed Computing in Sensor Systems (DCOSS)*, pages 216–229. Springer Berlin Heidelberg, 2010.

[16] N. X. Thang, M. Zapf, and K. Geihs. Model driven development for data-centric sensor network applications. In *Proceedings of the 9th International Conference on Advances in Mobile Computing and Multimedia*, MoMM '11, pages 194–197. ACM, 2011.

[17] 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 *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE)*, volume 989, pages 259–260. CEUR workshop proceedings, June 2013.

[18] C. Vicente-Chicote, F. Losilla, B. Alvarez, A. Iborra, and P. Sanchez. Applying MDE to the development of flexible and reusable wireless sensor networks. *International Journal of Cooperative Information Systems*, 16:393–412, 2007.