Demo: Towards Bug-free Implementation for Wireless Sensor Networks

Manchun Zheng
National University of Singapore
zmanchun@comp.nus.edu.sg

Jun Sun
Singapore University of Technology and Design
sunjun@sutd.edu.sg

David Sanán
Singapore University of Technology and Design
davidsanan@sutd.edu.sg

Yang Liu
National University of Singapore
liuyang@comp.nus.edu.sg

Jin Song Dong
National University of Singapore
dongjs@comp.nus.edu.sg

Yu Gu
Singapore University of Technology and Design
jasongu@sutd.edu.sg

Abstract

In this demonstration, a systematically domain-specific model checker, NesC@PAT, is presented. The tool takes NesC programs as input, and automatically verifies WSNs against properties specified in the form of deadlock freeness, state reachability or linear temporal logic formulas. We will show that NesC@PAT is able to find errors caused by rarely unexpected scenarios, which are difficult to be detected by general simulating or debugging.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods, Model checking

General Terms

Design, Verification

Keywords

NesC, TinyOS, model checking

1 Introduction

TinyOS has been widely used for developing wireless sensor network (WSN) applications. The programming language of TinyOS applications, NesC [3], provides fine-grained control over the underlying devices and resources. However, due to the event-driven feature of TinyOS/NesC and the concurrent execution of sensors and computations, it could be challenging to understand, analyze or debug NesC programs or WSN operations. Unexpected behaviors, like the overflow of the task queue, can evolve to very rare and buggy scenarios which are difficult to be detected by debugging tools [2] or simulating tools like TOSSIM [4].

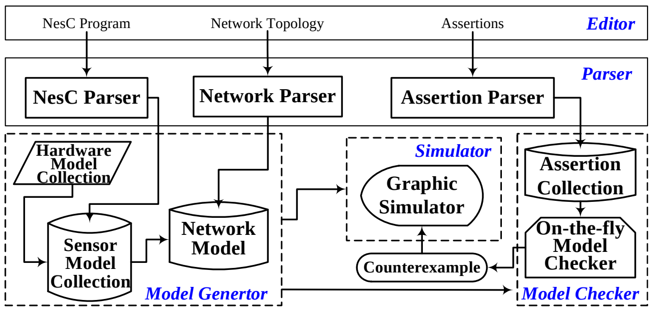

Figure 1. Overview of NesC@PAT

Model checking [1] is a verification technique to check desired properties (often expressed in temporal logic) by systematically exploring all possible scenarios of the given system. This technique has been applied to verifying both software and hardware systems. One recent success is the full verification of the Intel i7 chip using model checking techniques. One representative existing model checking tool for NesC programs is T-Check [5], which is built upon TOSSIM [4] and uses explicit state model checking to verify safety and liveness properties. T-Check revealed several bugs of components/applications in the TinyOS distribution. However, T-Check has limited capability in detecting concurrent and low-level errors because TOSSIM executes events atomically and abstracts away interrupt-driven concurrency. The assertions of T-Check are specified in propositional logic, and are incapable of specifying significant temporal properties like the *infinitely often* releases of a buffer or the *alternate* occurrences of two events.

Our contribution in this demonstration is a fully automatic model checking tool, NesC@PAT, which takes NesC programs and a network topology as the input and verifies WSNs against properties specified as deadlock freeness, state reachability or linear temporal logic (LTL) formulas. The expressive power of LTL has allowed the tool to specify a larger set of properties, compared to that supported by T-Check. With the aim of minimum manual effort, a model generator is integrated to generate models from NesC code automatically, preserving the interrupt-driven concurrency among tasks and events, which is essential for TinyOS applications. NesC@PAT covers most NesC language features in-

```
result_t tryNextSend(){
    atomic{
        if(!sendTaskBusy){
            post sendTask();
            sendTaskBusy = TRUE;
        }
    }...
}
```

(a) Buggy code

```
result_t tryNextSend(){
    atomic{
        if(!sendTaskBusy){
            if(SUCCESS != post sendTask())
                sendTaskBusy = FALSE;
            else sendTaskBusy = TRUE;
        }...
}
```

(b) Revised code

Figure 2. A Motivating Scenario

cluding pointer, command, event and task or even advanced features like parameterized wiring and hierarchical wiring.

2 Overview of NesC@PAT

With the **Parser** and the **Model Generator**, the NesC program running on a node, like the scratch showed in Figure 2(a), is translated into a Label Transition System (LTS), avoiding manual construction of models and making the tool useful in practice. The operational semantics for each NesC language structure has been defined [6], which provides the basis for constructing LTSs from NesC programs. Moreover, the interrupt-driven feature of the TinyOS execution model is preserved in the generated LTS, which allows concurrency errors among tasks and interrupts to be detected. Radio, timer, sensor device and other devices are abstracted in the Hardware Model Collection, which is also taken into account in the generation of LTS. With the network topology and individual node LTSs, the LTS of a WSN is composed, considering the non-determinism between sensor nodes.

The **Model Checker** conducts an exhaustive search (optimized by partial order reduction) of the generated LTS state space, and it returns a counterexample if an assertion (i.e., a correctness criterion) is violated. Currently, it integrates model checking algorithms for verifying deadlock freeness, state reachability and LTL formulas. This provides flexibility for specifying significant goals to verify WSNs against. Taking the code in Figure 2(a) as an example, if a previous *sendTask* has not been posted successfully, then *sendTaskBusy* will remain *TRUE*. As a result, the statement *post sendTask*() will not able to execute any more. Such a scenario is undesirable and should be avoided under any circumstance. With NesC@PAT, it is very convenient to find this buggy behavior by model checking the code with the LTL property $\Box\Diamond sendTaskBusy = FALSE$ (**P1**), i.e. *sendTaskBusy* is *always eventually* set to *FALSE*.

Using the **Simulator**, users can easily simulate the visualized execution of a node or a whole network step by step. At each step only a fine-grained statement (e.g., updating a variable) is executed, which provides detailed runtime behaviors to be monitored. Moreover, if a property is violated, the Model Checker will report a counterexample to the Simulator, so that users can reason about it and correct the buggy code. Verifying the code in Figure 2(a) against **P1**, the Model Checker will return a counterexample, in which there is a state where *post sendTask*() fails. Simulating this counterexample helps to correct the program to be the revised one in Figure 2(b), where *sendTaskBusy* is set to *FALSE* if the statement *post sendTask*() fails.

3 Demonstration Highlights

NesC@PAT is developed with a well-organized GUI, based on which we will present the benefits brought forth by a systematically domain-specific model checker. In specific, we will show how to prepare a WSN with different NesC programs on sensor nodes and a set of properties to be verified. It will be shown that node-level verification can help reduce errors before network-level verification. Multiple WSN applications, including Trickle algorithm and certain routing protocols, will be verified on the spot with appropriate properties, in order to exemplify finding significant errors by model checking. For violated properties, we will explain how to use the **Simulator** to visualize the counterexample to analyze and refine the source code. We will also discuss how to define important and desired properties in terms of state reachability or LTL formulas with regard to the particular requirements of different WSNs.

4 Conclusion and Future Work

In this demonstration, we present a systematic and fully automatic model checker NesC@PAT, for modeling and verifying WSNs with NesC programs. In the current tool, we assume that the network topology is static. A future direction is to establish network dynamics, e.g., nodes joining or leaving, communication failures, etc, and to apply probabilistic model checking for verification. Another direction is to optimize the tool for better scalability and efficiency by symmetric reduction or symbolic model checking.

5 References

[1] C. Baier and J. P. Katoen. *Principles of Model Checking*. The MIT Press, May 2008.

[2] Q. Cao, T. F. Abdelzaher, J. A. Stankovic, K. Whitehouse, and L. Luo. Declarative Tracepoints: a Programmable and Application Independent Debugging System for Wireless Sensor Networks. In *Proceedings of the 6th International Conference on Embedded Networked Sensor Systems*, SenSys'08, pages 85–98, Raleigh, NC, USA, 2008. ACM.

[3] D. Gay, P. Levis, R. v. Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the 24th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'03, pages 1–11, California, USA, 2003.

[4] P. Levis, N. Lee, M. Welsh, and D. E. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys'03. ACM, 2003.

[5] P. Li and J. Regehr. T-Check: bug finding for sensor networks. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks*, IPSN'10, pages 174–185, Stockholm, Sweden, 2010.

[6] M. Zheng, J. Sun, Y. Liu, J. S. Dong, and Y. Gu. Automatic Verification of TinyOS Applications for Wireless Sensor Networks. Technical report, National Univ. of Singapore, 2011. http://www.comp.nus.edu.sg/~pat/NesC_TechReport.pdf .