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].
post sendTask

based on which we will present the benefits brought forth

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.

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 mode 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 □ (!= sendTaskBusy) 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, on which we will present the benefits brought forth

Figure 2. A Motivating Scenario

(a) Buggy code

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

(b) Revised code

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