Towards a Model Checker for NesC and Wireless Sensor Networks *

Manchun Zheng[1], Jun Sun[2], Yang Liu[1], Jin Song Dong[1], and Yu Gu[2]

[1] School of Computing, National University of Singapore
{zmanchun,liuyang,dongjs}@comp.nus.edu.sg
[2] Singapore University of Technology and Design
{sunjun,jasongu}@sutd.edu.sg

Abstract. Wireless sensor networks (WSNs) are expected to run unattendedly for critical tasks. To guarantee the correctness of WSNs is important, but highly nontrivial due to the distributed nature. In this work, we present an automatic approach to directly verify WSNs built with TinyOS applications implemented in the NesC language. To achieve this target, we firstly define a set of formal operational semantics for most of the NesC language constructs for the first time. This allows us to capture the behaviors of sensors using labelled transition systems (LTSs), which are the underlying semantic models of NesC programs. Secondly, WSNs are modeled as the composition of sensors with a network topology. Verifications of individual sensors and WSNs become possible by exploring the corresponding LTSs using techniques like model checking. With substantial engineering efforts, we implement this approach in a toolkit named NesC@PAT, to support verifications of deadlock-freeness, state reachability and temporal properties for WSNs. NesC@PAT has been applied to analyze and verify WSNs, with *unknown* bugs being detected. To the best of our knowledge, NesC@PAT is the first model checker which takes NesC language as the modeling language and completely preserves the interrupt-driven feature of the TinyOS execution model.

1 Introduction

Wireless sensor networks (WSNs) are used in critical areas like military surveillance, environmental monitoring, seismic detection [2] and so forth. Such systems are expected to run unattendedly for a long time in environments that are usually unstable. Thus it is important for them to be highly reliable and correct. TinyOS [16] and NesC [7] have been widely used as the programming platform for developing WSNs, which adopt a low-level programming style [13]. Such a design provides fine-grained controls over the underlying devices and resources, but meanwhile makes implementations difficult to understand, analyze or verify. The challenges of modeling and formally verifying WSNs with NesC programs are as follows.

- The syntax and semantics of NesC are complex [7] compared to those of formal modeling languages. To the best of our knowledge, there has not been any formal semantics for the NesC language. Thus establishing formal models from NesC programs is non-trivial.

* This research is supported in part by Research Grant IDD11100102 of Singapore University of Technology and Design, IDC and MOE2009-T2-1-072 (Advanced Model Checking Systems).

- TinyOS provides hardware operations on motes (i.e. sensors) which can be invoked by NesC programs including messaging, sensing and so on [16,6]. Therefore, modeling NesC programs (which executes on TinyOS) requires modeling the behaviors of hardware at the same time.
- TinyOS adopts an interrupt-driven execution model, which introduces local concurrency (i.e. intra-sensor concurrency) between tasks and interrupts. Local concurrency increases the complexity of model checking NesC programs.

Related Work A number of approaches and tools have been published to analyze, simulate, debug and verify WSNs. W. Archer et al. presented their work on interface contracts for TinyOS components in [3]. Contract checking exposes bugs and hidden assumptions caused by improper interface usages, and adds plentiful safety conditions to TinyOS applications. Nguyet and Soffa [23] presented an approach to explore the internal structure of WSN applications using control flow graphs, but it supported no error detection. V. Menrad et al. proposed to use Statecharts to achieve readable yet more precise formulations of interface contracts [22]. These approaches contribute to improving the correctness of usages of interfaces, but are incapable of verifying any specific property like safety or liveness.

The tool *FSMGen* presented by N. Kothari et al. [13] infers compact, user-readable Finite State Machines from TinyOS applications and uses symbolic execution and predicate abstraction for static analysis. This tool captures highly abstract behaviors of NesC programs and reveals some errors. However, low-level interrupt driven code is not applicable since the tool is based on a coarse approximation of the TinyOS execution model. Some essential features like loops are not supported and the tool provides no supports for analyzing the concurrent behaviors of a WSN (rather than a single sensor).

Bucur and Kwiatkowska [4] proposed *Tos2CProver* for debugging and verifying TinyOS applications at compile-time, checking memory-related errors and other low-level errors upon registers and peripherals. Checking run-time properties like the unreachability of error states is not supported in *Tos2CProver*. Again, this approach only checks errors for single-node programs and lacks the ability to find network-level errors.

Hanna et al. proposed *SLEDE* [8,9] to verify security protocol implementations in the NesC language, by extracting PROMELA [11] models from NesC implementations. This approach is translation-based, and it abstracts away NesC semantics like the concurrency between tasks and interrupts, thus failing to find concurrency-related bugs that are significant. Moreover, *SLEDE* is dedicated for security protocols, and not applicable for verifying non-security properties like liveness.

T-Check [18] is built upon the TinyOS simulator TOSSIM [15] and uses explicit state model checking techniques to verify safety and liveness properties. T-Check revealed several bugs of components/applications in the TinyOS distribution, however, it has limited capability in detecting concurrent errors due to the limitation of TOSSIM, e.g., in TOSSIM, events execute atomically and are never preempted by interrupts. Moreover, the assertions of T-Check are specified in propositional logic, which is incapable of specifying important temporal properties like the *infinitely often* release of a buffer or the *alternate* occurrences of two events.

While the existing approaches have contributed a lot for analyzing and finding bugs of TinyOS applications or WSNs, few of them simulate or model the interrupt-driven

execution model of TinyOS. Further, only a few are dealing with WSNs, which are obviously more complex than individual sensors. In this paper, we propose a systematic and self-contained approach to model check WSNs built with TinyOS applications (i.e. NesC programs). Our work includes a component model library for hardware, and the formalized definitions of NesC programs and the TinyOS execution model. Based on these, the labelled transition systems (LTSs) of individual sensors can be constructed directly from NesC programs. The LTS of a WSN is then composed (on-the-fly) from the LTSs of individual sensors and a network topology. A network topology specifies how the sensors are connected. Model checking algorithms are developed to supports verifications of deadlock-freeness, state reachability and liveness properties specified as linear temporal logic (LTL) [21] formulas. Both the state space of a WSN and that of an individual sensor can be explored for verifications. With substantial engineering efforts, our approach has been implemented as the NesC module in PAT [19,25,20], named NesC@PAT (available at http://www.comp.nus.edu.sg/~pat/research). In this paper, we use NesC@PAT to verify the Trickle [17] algorithm of WSNs. A bug of the algorithm is found and it has been confirmed by implementing a WSN using real sensors (Iris motes). This shows that our approach can assist developers for behavioral analysis, error detection, and property verification of WSNs.

Contribution We highlight our contributions in the following aspects.

- Our approach works directly on NesC programs, without building (abstract) models before verifying WSNs. Manual construction of models is avoided, which makes our approach useful in practice.
- We formally define the operational semantics of NesC and TinyOS as well as WSNs. New semantic structures are introduced for modeling the TinyOS execution model and hardware-related behaviors like timing, messaging, etc.
- The interrupt-driven feature of the TinyOS execution model is preserved in the sensor models generated in our approach. This allows concurrency errors between tasks and interrupts to be detected.
- Our approach supports verifications of deadlock-freeness, state reachability and liveness properties. This provides flexibility for verifying various properties to guarantee the correctness of sensor networks.

The rest of the paper is organized as follows. Section 2 introduces NesC and TinyOS, and discusses the complexity and difficulty caused by specific features of NesC and TinyOS. The formal definitions of sensors and the operational semantics of TinyOS applications are presented in Section 3. Section 4 defines WSNs formally and introduces how the LTS of a WSN is obtained. Section 5 presents the architecture of our tool NesC@PAT and experimental results of verifying the Trickle algorithm. Finally, Section 6 concludes the paper with future works.

2 Preliminaries

This section briefly introduces the NesC programming language and the TinyOS operating system. Section 2.1 illustrates the specific features of NesC which make it complex

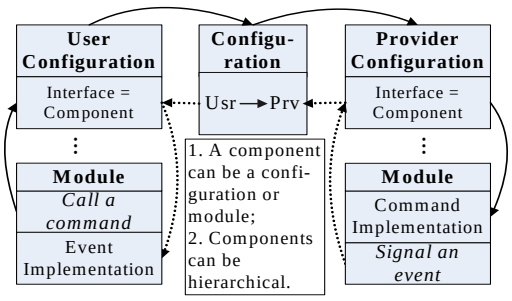

Fig. 1: Call graph of NesC programs

```
1.  int main() ··· {
2.      atomic {
3.          platform_bootstrap();
4.          call Scheduler.init();
5.          call PlatformInit.init();
6.       ···}
7.      __nesc_enable_interrupt();
8.      signal Boot.booted();
9.      call Scheduler.taskLoop();
10.      return -1;
11. }
```

Fig. 2: TinyOS Boot Sequence

to directly verify NesC programs. Section 2.2 introduces TinyOS with an explanation of its execution model and hardware abstraction architecture.

2.1 The NesC Language

The programming language NesC (Nested C) [7] is proposed for developing WSNs. NesC is a dialect of C, which embodies the structural concepts and the execution model of TinyOS applications. NesC has two basic modular concepts: interface and component. An *interface* declares a number of commands and events. A *command* denotes the request for an operation, and an *event* indicates the completion of that operation. In this way, NesC achieves a non-blocking programming convention, implementing operations in a split-phase mode. In other words, the request of a certain operation and its completion are separated.

An interface can be either *provided* or *used* by a component. In NesC, there are two types of *component*, i.e. configuration and module. A *configuration* indicates how components are wired to one another via interfaces. A *module* implements the *commands* declared by its provided interfaces and the *events* declared by its used interfaces. Commands and events are two types of functions, and *task* is the third. Usually, a component calls commands declared by its used interfaces, and signals events declared by its provided interfaces. Table 1 exemplifies the common-used constructs of the NesC language, and the corresponding operational semantics is discussed in Section 3.

A call graph describes the wiring relation between components. Fig. 1 illustrates a general call graph of NesC programs. Inside a configuration, a second-level configuration can be wired to a third component, where the second-level configuration itself contains a wiring relation between a set of components. Thus, the call graph of a NesC program might be a hierarchical 'tree' of components, where intermediate nodes are configurations and leaves are modules.

NesC is an extension of the C language. It does not support advanced C features like dynamic memory allocation, function pointer, multi-thread and so on, which makes it an 'easier' target for formal verification. Nonetheless, it supports almost the same set of operators, data types, statements and structures as C does and, in addition, NesC-

NesC Construct	Example	Remark
Command	*command* error_t AMControl.start() {···}	There are commands, events and tasks besides ordinary functions. The only difference among them is the way of invocation. A task is a parameterless function without any return value.
Event	*event* message_t* Receive.receive (message_t* msg, ···, uint8_t len) {··· return msg; }	
Task	*task* void setLeds() {···}	
Call	*call* Timer.startPeriodic(250);	Call, signal and post are function calls, invoking commands, events and tasks, respectively.
Signal	*signal* Timer.fired();	
Post a task	*post* setLeds();	
Atomic	*atomic*{x = x + 1; call AMSend.send(dst, pkt);}	Interrupts are disabled within an atomic block.

Table 1: Common-used NesC Constructs

specific features such as calling a command, signaling an event, posting a task and so forth. Verifying NesC programs is thus highly non-trivial, as illustrated in the following.

- Function calls like calling a command or signaling an event could be complex if the module invoking the command/event and that implementing it are wired via a hierarchical call graph. Further, nested function calls are supported in NesC, and it becomes even more complicated if there are recursive function calls.
- NesC allows local variables declared in functions or even in blocks of functions, just like C does. A traditional way to analyze local variables is to use stacks. Dealing with local variables significantly increases the complexity of verification.
- NesC is a typed programming language, and all data types of C including array and struct are supported. There are also type operations (e.g. type casting) supported by NesC. Therefore, modeling NesC should take into account typed aspects.
- There are other expressive features of NesC, which are inherited from C, however make it complex. Examples of such features include pointers, parameters being types, definition of types, pre-compilation, etc.

We remark that our approach targets at NesC programs and does not necessarily support verification of C programs. In the following, we briefly explain how we support the above 'troubling' features.

Fortunately, NesC is static [7], i.e. there is no dynamic memory allocation or function pointer. Thus the variable access and the call graph can be completely captured at compile time. In our work, we treat pointers as normal variables, the value of which is a reference to a certain variable. We develop a parser to produce the call graph of a NesC program with the function (command, event, task or normal function) bodies defined by each component. A nested search algorithm is designed to traverse the call graph for fetching the corresponding function body once a function invocation is evidenced.

Local variables are modeled statically in our approach, with a renaming method to avoid naming conflicts. Nested function calls are supported with the assumption that there are no circles within the calling stacks. This is because that we rename local variables according to the positions of their declarations. Thus distinguishing the local

variables between two invocations of the same function can be tricky and costly. However, the restriction is modest. The most common invocation circle of NesC programs lies in the split-phase operations, i.e. when a command finishes it signals an event and in that event when it is completed it calls back the command again. However, [14] recommends NesC programmers to avoid such a way of programming. Even in this situation we can still get rid of naming conflicts of local variables because a new invocation of a function is assured to be at the end of the previous one.

Typed information is captured and we distinguish variables declared as different types and analyze functions with parameters being types. Our work also supports defining new types by *struct* and *typedef*. Moreover, pre-compilation is supported, as well as capturing information from *.h* files. More details of tackling NesC language features can be found in our technical report in [1].

2.2 TinyOS and Its Execution Model

TinyOS [6,16] is the run-time environment of NesC programs. The behavior of a NesC program is thus related to the interrupt-driven execution model of TinyOS [14]. Tasks are deferred computations, which are scheduled in FIFO order. Tasks always run till completion, i.e. only after a task is completed, then a new task can start its execution. In contrast, interrupts are preemptive and always preempt tasks. In our work, this interrupt-driven feature is captured using an interrupt operator ($\triangle$).

The operating system TinyOS is implemented in NesC, with a component library for operations on devices like sensing, messaging, timing, etc. The TinyOS component library adopts a three-layer Hardware Abstraction Architecture (HAA), including Hardware Presentation Layer (HPL), Hardware Adaptation Layer (HAL) and Hardware Interface Layer (HIL) [16,6]. The design of HAA gradually adapts the capabilities of the underlying hardware platforms to platform-independent interfaces between the operating system and the application code. Specific semantic structures are introduced for modeling hardware devices, which will be discussed in Section 3.

Since TinyOS 2.0, each NesC application should contain a component *MainC* (predefined by TinyOS), which implements the boot sequence of a sensor [14]. Fig. 2 sketches the function that implements the boot sequence. At first, the scheduler, hardware platform and related software components are initialized (line 3-5). Then interrupts are enabled (line 7) and the event *booted* of interface *Boot* (*Boot.booted*) is signaled (line 8), after which the scheduler recurrently runs tasks that have been posted (line 9). The execution of line 2 to 7 is usually short and always decided by TinyOS thus our approach assumes that this part is always correct and begins modeling the behaviors of a sensor at the execution of event *Boot.booted*.

3 Formalizing Sensors with NesC Programs

This section presents the formalization of sensors running TinyOS applications. In particular, we present the operational semantics of the NesC programming constructs, and introduce dedicated semantic structures for capturing the TinyOS execution model and hardware behaviors and then the LTS semantics of sensors.

The behaviors of a sensor are determined by the scheduler of tasks and the concurrent execution between tasks and device interrupts.

Definition 1 (Sensor Model). *A sensor model $\mathcal{S}$ is a tuple $\mathcal{S} = (A, T, R, init, P)$ where A is a finite set of variables; T is a queue which stores posted tasks in FIFO order; R is a buffer that keeps incoming messages sent by other sensors; init is the initial valuation of the variable set A; and P is a program, composed by the running NesC program M that can be interrupted by various devices H, i.e., $P = M \triangle H$.*

H models (and often abstracts) the behaviors of hardware devices such as Timer, Receiver and Reader (i.e. the sensing device). Because tasks are deferred computations, when posted they are pushed into the task queue T for scheduling in FIFO order. We remark that T and R are empty initially for any sensor model $\mathcal{S}$. The interrupt operator ($\triangle$) is introduced to capture the interrupt-driven feature of the TinyOS execution model, which will be explained later in this section.

The variables in A are categorized into two groups. One is composed of variables declared in the NesC program, which are further divided into two categories according to their scopes, i.e. component variables and local variables. Component variables are defined in a component's scope, whereas local variables are defined within a function's or a block's scope. In this work, all variables including local variables are loaded to the variable set A at initialization. To avoid naming conflicts, the name of each variable is first prefixed with the component name. A local variable is further renamed with the line number of its declaration position. The other is a set of *device status variables* that capture the states of hardware devices. For example, *MessageC.Status* is introduced to model the status of the messaging device. A status variable is added into A after compilation if the corresponding device is accessed in the NesC program.

Example 1. **Trickle** [17] is a code propagation algorithm which is intended to reduce network traffic. Trickle controls message sending rate so that each sensor hears a small but enough number of packets to stay up-to-date. In the following, the notion *code* denotes large-size data (e.g. a route table) each sensor maintains, while *code summary* denotes small-size data that summarizes the *code* (e.g. the latest updating time of the route table). Each sensor periodically broadcasts its code summary, and

- stays quiet if it receives an identical code summary;
- broadcasts its code if it receives an older summary;
- broadcasts its code summary if it receives an newer summary.

We have implemented this algorithm in a NesC program *TrickleAppC* (available in [1]), with the modification that a sensor only broadcasts the summary initially (instead of periodically) and if it receives any newer summary. The struct *MetaMsg* is defined to encode a packet with a summary, and *ProMsg* is defined to encode a packet with a summary and the corresponding code. Initially, each node broadcasts its summary (a *MetaMsg* packet) to the network . If an incoming *MetaMsg* packet has a newer summary, the sensor will broadcast its summary; if the received summary is outdated, the sensor will broadcast its summary and code (a *ProMsg* packet). An incoming *ProMsg* packet with a newer summary will update the sensor's summary and code accordingly.

```
event void AMControl.startDone(error_t rs){
    if(rs! = SUCCESS){
    //The previous request fails
      call AMControl.start();
    } else { sendSummanry(); }
}
```

Fig. 3: Event *AMControl.startDone*

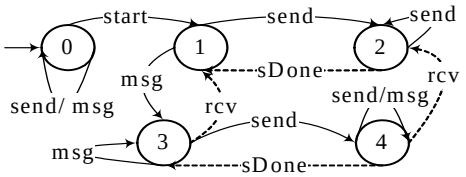

Fig. 4: The *MessageC* Model

Assume that a sensor executes *TrickleAppC*. By Definition 1, the corresponding sensor model is $\mathcal{S} = (A, T, R, init, P)$. In *TrickleAppC*, component *TrickleC* is referred as *App*, so *App* is used for renaming its variables. The variable set after renaming is $A = \{$ *MessageC.Status*, *App.summary*, *App.code*, *App*.34.*pkt*, $\cdots\}$, where variables with two-field names (e.g. *App.summary*) are component variables, except for *MessageC.Status* (a device status variable) and those with three-field names (e.g. *App*.34.*pkt*) are local variables. *init* is the initial valuation where *MessageC.Status* $=$ *OFF*, *App.summary* $= 0$, *App.code* $= 0$, *App*.34.*pkt* $=$ *null*, $\cdots$. In *TrickleAppC*, only the messaging device *MessageC* is accessed, therefore, initially the program *P* is event *Boot.booted* interrupted by the messaging device, i.e., $P = M_b \triangle$ *MessageC*. Event *Boot.booted* is implemented by *TrickleC*, and the following is its function body.

```
event void Boot.booted(){
    call AMControl.start(); //Start the messaging device
}
```

Calling *AMControl.start* will execute the corresponding command implemented by *ActiveMessageC*. Component *ActiveMessageC* is defined in the TinyOS component library for activating the messaging device. When *AMControl.start* is completed, the event *AMControl.startDone* (implemented by *TrickleC*) will be signaled. If *AMControl.start* returns *SUCCESS*, the function *sendSummary* is called for sending the summary. Otherwise, the command *AMControl.start* is re-called, as shown in Fig. 3. We use a model *MessageC* to describe the behaviors of the messaging device of a sensor. If *AMControl.start* is performed successfully, the program *P* of $\mathcal{S}$ will become $P = M_b' \triangle$ *MessageC*$'$, and the value of *MessageC.Status* will be modified. □

The models of the hardware devices are developed systematically. According to the TinyOS component library, we develop a component model library for most common-used components like *AMSenderC*, *AMReceiverC*, *TimerC*, etc[1]. We currently model hardware at HAA's top layer, i.e. Hardware Interface Layer, ignoring differences between the underlying platforms. For example, components *ActiveMessageC*, *AMSenderC* and *AMReceiverC* from the TinyOS component library are designed for different operations on the messaging device, such as activation, message transmission and message reception. Although there may be multiple *AMSenderC*'s or *AMReceiverC*'s in a NesC program, they all share the same messaging device. In Fig. 4, action *start* is a command from *ActiveMessageC*, action *send* is a request for sending a message from an *AMSendderC*, and action *msg* is the arrival of an incoming message. Action *sDone* and

[1] The current component library is not yet complete but sufficient for many NesC programs.

action *rcv* are interrupts triggered by *MessageC*, which signal the *sendDone* event of *AMSenderC* and the *receive* event of *AMReceiverC*, respectively.

Definition 2 (Sensor Configuration). *Let $\mathcal{S} = (A, T, R, init, P)$ be a sensor model. A sensor configuration C of $\mathcal{S}$ is a tuple (V, Q, B, P) where V is the valuation of variables A; Q is a sequence of tasks, being the content of T; B is a sequence of messages, being the content of R; P is the executing program.*

For a sensor model $\mathcal{S} = (A, T, R, init, P)$, its initial configuration $C^{init} = (init, \varnothing, \varnothing, P)$, in respect that initially task queue T and message buffer R are empty. A transition is written as $(V, Q, B, P) \xrightarrow{e} (V', Q', B', P')$ (or $C \xrightarrow{e} C'$ for short). Next, we define the behavior of a sensor as an LTS.

Definition 3 (Sensor Transition System). *Let $\mathcal{S} = (A, T, R, init, P)$ be a sensor model. The transition system of $\mathcal{S}$ is defined as a tuple $\mathbb{T} = (\mathbb{C}, init, \rightarrow)$, where $\mathbb{C}$ is the set of all reachable sensor configurations and $\rightarrow$ is the transition relation.*

The transition relation is formally defined through a set of firing rules associated with each and every NesC programming construct. The firing rules for *post* and *call* are presented in Fig. 5 for illustration purpose. The complete set of firing rules can be found in [1]. The following symbols are adopted to define the firing rules.

- $^\frown$ is sequence concatenation.
- $\checkmark$ simply denotes the termination of the execution of a statement.
- τ is an event label denoting a silent transition.
- $Impl(f, L_{arg})$ returns the the body of function $(\{F\})f$ with arguments L_{arg}.
- $FstFnc(L_{arg})$ returns the first element in L_arg which contains function calls.
- $L_{arg}[a'/a]$ replaces argument a with a' in the argument list L_{arg}.
- I is a status variable in A, denoting whether interrupts are allowed. Interrupts are only disabled within an *atomic* block thus I is set *off* only during an *atomic* block.

Rules *post*1 and *post*2 describe the semantics of the statement *post tsk*(). The task *tsk* will be pushed to the task queue Q if there are no identical tasks pending in Q (rule *post*1), otherwise the task is simply dropped (rule *post*2). Rules *call*1 and *call*2 capture the semantics of a command call *call intf.cmd*(L_{arg}), where L_{arg} is the list of arguments. If the arguments contain no function calls, the execution of a command call will transit directly to the execution of the corresponding function body (rule *call*1). Otherwise, the function calls in the arguments will be executed first (rule *call*2), also step by step.

Apart from the firing rules for NesC structures, we adopt several operators and semantic structures from process algebra community [10] to capture the execution model of TinyOS and hardware behaviors. Some of these firing rules are presented in Fig. 6, and the complete set can be found in [1]. The interrupt operator ($\triangle$) is used to formalize the concurrent execution between tasks and interrupts, and interrupts always preempt tasks, denoted by rules *itr*1. Further, when a task (M) completes its completion, a new task will be fetched from Q for execution (rule *itr*3). Interrupts are always enabled in rules *itr*3 and *itr*4 because no atomic blocks are executing. A sensor will remain *idle* when no interrupts are triggered by devices (i.e. *H is idle*) and no tasks are deferred (rule *itr*4), and it can be activated by an interrupt like the arrival of a new message.

$$\frac{e = s.post.t,\ t \notin Q,\ Q' = Q \frown \langle t \rangle}{(V,\ Q,\ B,\ post\ tsk()) \xrightarrow{e} (V,\ Q',\ B,\ \checkmark)} \quad [\ post1\ ]$$

$$\frac{e = s.post.t,\ t \in Q}{(V,\ Q,\ B,\ post\ tsk()) \xrightarrow{e} (V,\ Q,\ B,\ \checkmark)} \quad [\ post2\ ]$$

$$\frac{\forall a \in L_{arg} : (V,\ Q,\ B,\ a) \xrightarrow{\tau} (V,\ Q,\ B,\ \checkmark),\ F = Impl(intf.cmd, L_{arg})}{(V,\ Q,\ B,\ call\ intf.cmd(L_{arg})) \xrightarrow{e} (V,\ Q,\ B,\ \{F\})} \quad [\ call1\ ]$$

$$\frac{FstFnc(L_{arg}) = a,\ a \neq \epsilon,\ (V,\ Q,\ B,\ a) \xrightarrow{e} (V',\ Q',\ B',\ a')}{(V,\ Q,\ B,\ call\ intf.cmd(L_{arg})) \xrightarrow{e} (V',\ Q',\ B',\ call\ intf.cmd(L_{arg}[a'/a]))} \quad [\ call2\ ]$$

Fig. 5: Firing Rules for NesC Structures

A hardware interrupt is modeled as an atomic action which pushes a task to the top of the task queue, and thus the task has a higher priority than others. This task will signal the corresponding event for handling the interrupt. This is exactly the way that TinyOS deals with hardware interrupts. For example, when an interrupt is triggered by an incoming message, a task (t_{rcv}) will be added at the head of Q for signaling a *receive* event (rule *rcv*). Semantic structures *Send* and *Rcv* are defined to model the behaviors of sending and receiving a message respectively. *Send* is defined as (s, dst, msg), where s is the identifier of the sensor which sends a message, *dst* is the list of receivers and *msg* is the message itself.

Notice that devices such as Timer, Receiver, Reader (Sensor) and so on 'execute' concurrently, because they can trigger interrupts independently. This is captured using an interleave operator $|||$, which resembles the interleave operator in CSP [10].

4 Formalizing Wireless Sensor Networks

In this section, we formalize WSNs as LTSs. A sensor network $\mathcal{N}$ is composed of a set of sensors and a network topology[2]. From a logical point of view, a network topology is simply a directed graph where nodes are sensors and links are directed communications between sensors. In reality, a sensor always broadcasts messages and only the ones within its radio range would be able to receive the messages. We introduce radio range model to describe network topology, i.e. whether a sensor is able to send messages to some other sensor. Let $\mathbb{N} = \{0, 1, \cdots, i, \cdots, n\}$ be the set of the unique identifier of each sensor in a WSN $\mathcal{N}$. The radio range model is defined as the relation $\mathcal{R} : \mathbb{N} \leftrightarrow \mathbb{N}$, such that $(i, j) \in \mathcal{R}$ if and only if sensor j is within sensor i's radio range. We define a WSN model as the parallel of the sensors with its topology, as shown in Definition 4.

[2] We assume that the network topology for a given WSN is fixed in this work.

$$\frac{V(I) = on,\ (V,\ Q,\ B,\ H) \xrightarrow{e} (V',\ Q',\ B',\ H')}{(V,\ Q,\ B,\ M \triangle H) \xrightarrow{e} (V',\ Q',\ B',\ M \triangle H')} \quad [\ itr1\ ]$$

$$\frac{H \text{ is idle or } V(I) = off,\ (V,\ Q,\ B,\ M) \xrightarrow{e} (V',\ Q',\ B',\ M')}{(V,\ Q,\ B,\ M \triangle H) \xrightarrow{e} (V',\ Q',\ B',\ M' \triangle H)} \quad [\ itr2\ ]$$

$$\frac{H \text{ is idle},\ M = Impl(t, \varnothing)}{(V,\ \langle t \rangle^{\frown} Q',\ B,\ \checkmark \triangle H) \xrightarrow{e} (V,\ Q',\ B,\ M \triangle H)} \quad [\ itr3\ ]$$

$$\frac{H \text{ is idle}}{(V,\ \varnothing,\ B,\ \checkmark \triangle H) \xrightarrow{s.idle} (V,\ \varnothing,\ B,\ \checkmark \triangle H)} \quad [\ itr4\ ]$$

$$\frac{B = \langle msg \rangle^{\frown} B',\ t_{rcv} \notin Q,\ Q' = \langle t_{rcv} \rangle\ ^{\frown} Q}{(V,\ Q,\ B,\ Rcv) \xrightarrow{s.rcv\_msg} (V,\ Q',\ B',\ Rcv)} \quad [\ rcv\ ]$$

$$\frac{t_{sendDone} \notin Q,\ Q' = \langle t_{sendDone} \rangle\ ^{\frown} Q}{(V,\ Q,\ B,\ Send(s, msg)) \xrightarrow{s.send.msg} (V,\ Q',\ B,\ \checkmark)} \quad [\ send\ ]$$

Fig. 6: Firing Rules for Concurrent Execution and Hardware Behaviors

Definition 4 (WSN Model). *The model of a wireless sensor network $\mathcal{N}$ is defined as a tuple$(\mathcal{R}, \{\mathcal{S}_0, \cdots, \mathcal{S}_n\})$ where $\mathcal{R}$ is the radio model (i.e. network topology), $\{\mathcal{S}_0, \cdots, \mathcal{S}_n\}$ is a finite ordered set of sensor models, and $\mathcal{S}_i$ ($0 \leqslant i \leqslant n$) is the model of sensor i.*

Sensors in a network can communicate through messaging, and semantic structures *Send* and *Rcv* are defined to model message transmission among sensors. WSNs are highly concurrent as all sensors run in parallel, i.e. the network behaviors are obtained by non-deterministically choosing one sensor to execute at each step.

Definition 5 (WSN Configuration). *Let $\mathcal{N} = (\mathcal{R}, \{\mathcal{S}_0, \cdots, \mathcal{S}_n\})$ be a WSN model. A configuration of $\mathcal{N}$ is defined as the finite ordered set of sensor configurations: $\mathcal{C} = \{C_0, \cdots, C_n\}$ where C_i ($0 \leqslant i \leqslant n$) is the configuration of $\mathcal{S}_i$.*

Definition 5 formally defines a global system state of a WSN. Next, the semantics of sensor networks can be defined in LTSs, as follows.

Definition 6 (WSN Transition System). *Let $\mathcal{N} = (\mathcal{R}, \{\mathcal{S}_0, \cdots, \mathcal{S}_n\})$ be a sensor network model. The WSN transition system corresponding to $\mathcal{N}$ is a 3-tuple $\mathcal{T} = (\Gamma, init, \hookrightarrow)$ where Γ is the set of all reachable WSN configurations, $init = \{C_0^{init}, \cdots, C_n^{init}\}$ (C_i^{init} is the initial configuration of $\mathcal{S}_i$) being the initial configuration of $\mathcal{N}$, and $\hookrightarrow$ is the transition relation.*

$$\frac{C_i \xrightarrow{e} C_i',\ e \neq s_i.send.dst.msg,\ e \neq s_i.idle}{\{C_0,\cdots,C_i,\cdots,C_n\} \overset{e}{\hookrightarrow} \{C_0,\cdots,C_i',\cdots,C_n\}}\ [\ network1\ ]$$

$$\frac{C_i \xrightarrow{e} C_i',\ e = s_i.send.msg,\ \forall j \in [0,i) \cup (i,n] \bullet C_j' = InMsg(Radio(i), msg, C_j)}{\{C_0,\cdots,C_i,\cdots,C_n\} \overset{e}{\hookrightarrow} \{C_0',\cdots,C_i',\cdots,C_n'\}}\ [\ network2\ ]$$

Fig. 8: Firing Rules for Sensor Networks

Example 2. Fig. 7 presents a WSN with three nodes (i.e., $\mathcal{S}_0$, $\mathcal{S}_1$ and $\mathcal{S}_2$), each of which is implemented with application *TrickleAppC* (*Tk* for short) of Example 1. The radio range of each senor is described by a circle around it, and the network topology model can be abstracted as $\mathcal{R} = \{(0,1),(1,2),(2,0)\}$.

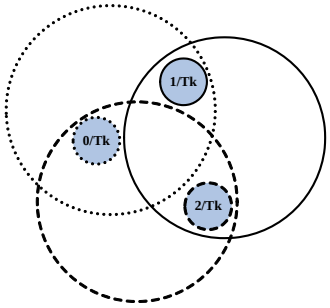

Fig. 7: A WSN Example

A transition of a WSN is of the form $\mathcal{C} \overset{e}{\hookrightarrow} \mathcal{C}'$, where $\mathcal{C} = \{C_0,\cdots,C_i,\cdots,C_n\}$ and $\mathcal{C}' = \{C_0',\cdots,C_i',\cdots,C_n'\}$. The transition relation is obtained through a set of firing rules, which are shown in Fig. 8. Rule *network*1 describes the concurrent execution betweens sensors, i.e. the network non-deterministically chooses a sensor to perform a transition. Rule *network*2 is dedicated for communication. Function $Radio(i)$ returns the set of sensors that are within sensor i's radio range. Function $InMsg(msg, C_i)$ enqueues the message msg to sensor i, i.e. $C_j' = InMsg(dst, msg, C_j) \Leftrightarrow (j \in dst \Rightarrow C_j' = C_j[B_j \frown \langle msg \rangle / B_j]) \wedge (j \notin dst \Rightarrow C_j' = C_j)$. Thus a sensor sending a message will not only change its local state but also change those of the sensors in the destination list, by enqueuing the message to their message buffers.

5 Implementation and Evaluation

Our approach has been implemented in the model checking framework PAT as the NesC module, which is named NesC@PAT. Fig. 9 illustrates the architecture of NesC@PAT. There are five main components, *i.e.* an editor, a parser, a model generator, a simulator and a model checker. The editor allows users to input different NesC programs for sensors and to draw the network topology, as well as to define assertions (i.e. verification goals). The parser compiles all inputs from the editor. The model generator generates sensor models based on the NesC programs and the built-in hardware model collection (i.e. the component model library). Furthermore, it generates WSN models. Both the sensor models and the network models are then passed to the simulator and the model checker for visualized simulation and automated verification respectively.

NesC@PAT supports both sensor-level and network-level verifications, against properties including deadlock-freeness, state reachability, and liveness properties expressed

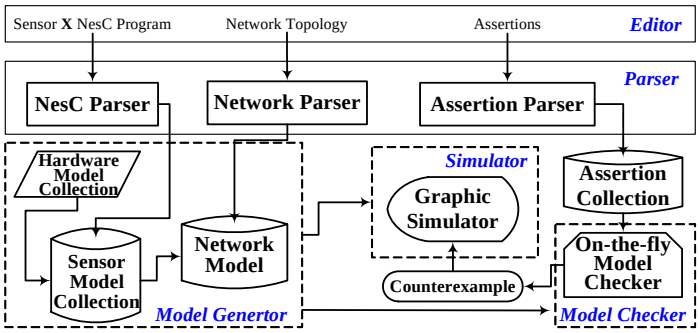

Fig. 9: Architecture of NesC@PAT

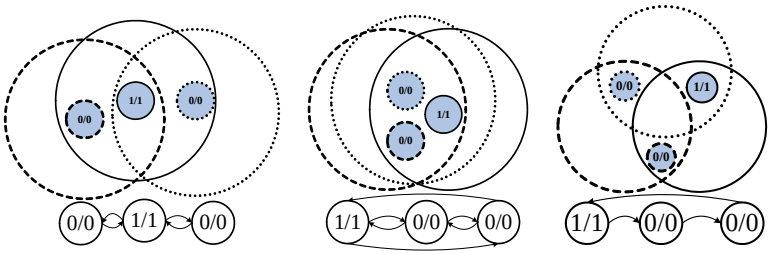

Fig. 10: Network Topology: Star, Ring, Single-track Ring

in linear temporal logic (LTL) [21]. Deadlock-freeness and state reachability are checked by exhaustively exploring the state space using Depth-first search or Breadth-first search algorithms. We adopt the approach presented in [25] to verify LTL properties. First, the negation of an LTL formula is converting into an equivalent Büchi automaton; and then accepting strong connected components (SCC) in the synchronous product of the automaton and the model are examined in order to find a counterexample. Notice that the SCC-based algorithm allows us to model check with fairness [27], which often plays an important role in proving liveness properties of WSNs.

NesC@PAT was used to analyze WSNs deployed with the Trickle algorithm presented in Example 1. We studied WSNs with different topologies including star, ring and single-tracked ring (short for SRing). The settings of WSNs are presented in Fig. 10, where $1/1$ stands for *new* code/*new* summary and $0/0$ stands for *old* code/*old* summary and a directed graph is used to illustrate the logical view for each network. Two safety properties (i.e. *Deadlock free* and *Reach FalseUpdat* $-$ *ed*) and two liveness properties (i.e. $\Diamond$*AllUpdated* and $\Box\Diamond$*AllUpdated*) are verified. Property *Reach FalseUpdated* is a state reachability checking, which is valid if and only if the state *FalseUpdated* where a node updates its code with an older one can be reached. Property $\Diamond$*AllUpdated* is an LTL formula which is valid if and only if the state *AllUpdated* where all nodes are updated with the new code can be reached *eventually*, while $\Box\Diamond$*AllUpdated* requires state *AllUpdated* to be reached *infinitely often*.

(a) Initial Setting (b) Star/Ring network (c) SRing network

Fig. 11: Real Executions on Iris Motes

A server with Intel Xeon 4-Core CPU*2 and 32 GB memory was used to run the verifications, and the results are summarized in Table 2. The results show that the algorithm satisfies the safety properties, i.e. neither a deadlock state nor a *FalseUpdated* state is reachable. As for the liveness property, both ring and star networks work well, which means that every node is eventually updated with the new code. However, the single-tracked-ring (SRing) network fails to satisfy either liveness property. The counterexample produced by NesC@PAT shows that only one sensor can be updated with the new code. By simulating the counterexample in NesC@PAT, we can find the reason of this bug. On one hand, the initially updated node A receives old summary from node C thus broadcasting its code but only node B can hear it. On the other hand, after node B is updated it fails to send its code to node C because node B never hears an older summary from node C. We also increased the number of sensors for SRing networks, and the results remained the same.

We ran the Trickle program on Iris motes to study whether this bug could be evidenced in real executions. The *TrickleAppC* was modified by adding operations on leds to display the changes of code (available in [1]):

1. Initially sensor *A* has the new code $(1/1)$ and has its red led on, while sensor *B* and *C* have the old code $(0/0)$ and have their yellow leds on, as shown in Fig. 11a.
2. A sensor will turns on its red led when it is updated with the new code.
3. Different topologies are achieved by specifying different *AM id* for each sensor's *AMSenderC* and *AMReceiverC*, details of which are in [1].

The revised *TrickleAppC* was executed on real sensors with star, ring and single-tracked ring topologies. Fig. 11b shows that the star/ring network is able to update all nodes, and Fig. 11c shows that the single-tracked ring network fails to update one node, which confirms that the bug found by NesC@PAT could be evidenced in reality.

Discussion The results in Table 2 show that a WSN of Trickle algorithm with three nodes has a state space of $10^6 \sim 10^7$, and the state space grows exponentially with the network topology and the number of nodes. One direction of our future work is to introduce reduction techniques such like partial order reduction [24] and symmetry reduction [5] to improve the scalability of our approach. Hardware-related behaviors are abstracted and manually modeled based on real sensors, i.e. Iris motes. This manual abstraction of hardware is simple to implement, but lacks the ability to find errors due to hardware failures, as the hardware is assumed to be always working.

Network	Property	Size	Result	#State	#Transition	Time(s)	Memory(KB)
Star	Deadlock free	3	✓	300,332	859,115	49	42,936
	Reach FalseUpdated		×	300,332	859,115	47	23,165
	◇AllUpdated		✓	791,419	2,270,243	148	25,133
	□◇AllUpdated		✓	1,620,273	6,885,511	654	13,281,100
Ring	Deadlock free	3	✓	1,093,077	3,152,574	171	80,108
	Reach FalseUpdated		×	1,093,077	3,152,574	161	27,871
	◇AllUpdated		✓	2,127,930	6,157,922	389	78,435
SRing	Deadlock free	3	✓	30,872	85,143	5	19,968
		4	✓	672,136	2,476,655	170	72,209
	Reach FalseUpdated	3	×	30,872	85,143	5	23,641
		4	×	672,136	2,476,655	156	62,778
	◇AllUpdated	3	×	42	73	<1	19,290
		4	×	52	113	<1	19,771
	□◇AllUpdated	3	×	146	147	<1	51,938
		4	×	226	227	<1	51,421
		8	×	746	747	<1	59,900
		20	×	4,126	4,127	2	148,155

Table 2: Experimental Results

6 Conclusion

In this work, we present an initial step towards automatic verifications of WSNs at implementation level. Semantic models, i.e. event-based LTS, of WSNs are generated directly and automatically from NesC programs, avoiding manual construction of models. To the best of our knowledge, our approach is the first complete and systematical approach to verify networked NesC programs. Moreover, our approach is the first to model the interrupt-driven execution model of TinyOS. This is important since it allows concurrency errors at sensor level to be detected. Model checking algorithms have been implemented for verifying various properties.

Our work currently adopts a non-threaded execution model of TinyOS. Recently, new models have been proposed, e.g., TOSThread [12] has been proposed to allow user threads in TinyOS. Our future work thus includes designing approach for modeling TOSThread. Moreover, the current component model library of NesC@PAT only models a subset of the TinyOS component library. Another future direction is to generate comprehensive timing models using techniques [26,20] and to take failures into account by probabilistic modeling techniques [28]. We also plan to apply reduction techniques [29] for optimizing the usage of time and memory at verification phase.

References

1. NesC@PAT. http://www.comp.nus.edu.sg/~pat/NesC/ .
2. I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38:132–138, 2001.
3. W. Archer, P. Levis, and J. Regehr. Interface contracts for TinyOS. In *IPSN*, pages 158–165, 2007.
4. D. Bucur and M. Z. Kwiatkowska. Software verification for TinyOS. In *IPSN*, pages 400–401, 2010.

5. E. A. Emerson, S. Jha, and D. Peled. Combining Partial Order and Symmetry Reductions. In *TACAS*, pages 19–34, 1997.
6. D. Gay, P. Levis, and D. E. Culler. Software design patterns for TinyOS. *ACM Trans. Embedded Comput. Syst.*, 6(2), 2007.
7. 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 *PLDI*, pages 1–11, 2003.
8. Y. Hanna and H. Rajan. Slede: Framework for automatic verification of sensor network security protocol implementations. In *ICSE Companion*, pages 427–428, 2009.
9. Y. Hanna, H. Rajan, and W. Zhang. Slede: a domain-specific verification framework for sensor network security protocol implementations. In *WISEC*, pages 109–118, 2008.
10. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
11. G. J. Holzmann. Design and Validation of Protocols: A Tutorial. *Computer Networks and ISDN Systems*, 25(9):981–1017, 1993.
12. K. Klues, C.-J. M. Liang, J. Paek, R. Musaloiu-Elefteri, P. Levis, A. Terzis, and R. Govindan. TOSThreads: thread-safe and non-invasive preemption in TinyOS. In *SenSys*, pages 127–140, 2009.
13. N. Kothari, T. D. Millstein, and R. Govindan. Deriving State Machines from TinyOS Programs Using Symbolic Execution. In *IPSN*, pages 271–282, 2008.
14. P. Levis and D. Gay. *TinyOS Programming*. Cambridge University Press, 1 edition, 2009.
15. P. Levis, N. Lee, M. Welsh, and D. E. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *SenSys*, pages 126–137, 2003.
16. P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*. Springer Verlag, 2004.
17. P. Levis, N. Patel, D. E. Culler, and S. Shenker. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *NSDI*, pages 15–28, 2004.
18. P. Li and J. Regehr. T-check: bug finding for sensor networks. In *IPSN*, pages 174–185, 2010.
19. Y. Liu, J. Sun, and J. S. Dong. An Analyzer for Extended Compositional Process Algebras. In *ICSE Companion*, pages 919–920. ACM, 2008.
20. Y. Liu, J. Sun, and J. S. Dong. Developing Model Checkers Using PAT. In *ATVA*, pages 371–377, 2010.
21. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems:Specification*. Springer-Verlag, 1992.
22. V. Menrad, M. Garcia, and S. Schupp. Improving TinyOS Developer Productivity with State Charts. In *SOMSED*, 2009.
23. N. T. M. Nguyen and M. L. Soffa. Program representations for testing wireless sensor network applications. In *DOSTA*, pages 20–26, 2007.
24. D. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. *Formal Methods in System Design*, 8(1):39–64, 1996.
25. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, pages 709–714, 2009.
26. J. Sun, Y. Liu, J. S. Dong, and X. Zhang. Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In *ICFEM*, pages 581–600, 2009.
27. J. Sun, Y. Liu, A. Roychoudhury, S. Liu, and J. S. Dong. Fair model checking with process counter abstraction. In *FM*, pages 123–139, 2009.
28. J. Sun, S. Song, and Y. Liu. Model Checking Hierarchical Probabilistic Systems. In *ICFEM*, pages 388–403, 2010.
29. S. J. Zhang, J. Sun, J. Pang, Y. Liu, and J. S. Dong. On Combining State Space Reductions with Global Fairness Assumptions. In *FM*, pages 432–447, 2011.