An Automatic Approach To Verify Sensor Network Systems

Man Chun Zheng
School of Computing
National University of Singapore
Singapore
zmanchun@comp.nus.edu.sg

Abstract—The programming language nesC for TinyOS applications supports special features of sensor network systems by providing a component-oriented programming model which is flexibly concurrent/reactive and event-driven. Sensor network systems are correctness critical since they are expected to work autonomously. Formal verification techniques such as model checking have been successfully applied to assure the reliability and correctness of concurrent systems and real-time systems. However, manually constructing a formal model is always a non-trivial task. We develop a lightweight framework for sensor network systems which automatically extracts real-time models from nesC implementations and verifies them against goals using model checking techniques. We believe that our approach contributes to systematically improving the quality of sensor network systems, with little overhead or cost caused by applying verification techniques.

Keywords-Sensor Network System; Model Checking; RTS;

I. INTRODUCTION

A sensor network consists of a large number of tiny, low-power sensor nodes, each of which executes concurrent, reactive programs that must operate with severe memory and power constraints. To deal with such constraints, TinyOS [1] is proposed as an operating system for sensor network systems. Moreover, nesC was proposed by Gay et al. [2] as a programming language for developing TinyOS applications.

The well-designed mechanisms of TinyOS and the expressiveness of nesC have attracted a lot of research users. Although nesC has provided advanced program analysis such as compile-time data race detection, it cannot assure complete correctness of a given nesC application. Since the execution model of TinyOS is concurrent and based on tasks and interrupt handler, the correctness of concurrency (such as absence of race conditions) is critical. Furthermore, the correctness of real-time constraints, power managements, sensing and communicating behaviors of sensor network systems have motivated researchers to explore formal verification techniques for sensor network systems.

In this paper, we propose to automatically generate real-time system (RTS) models [3] from sensor network systems implemented in nesC. Model checking techniques are also applied to the RTS models for verifying properties such as deadlock freeness, divergence freeness, state reachability, temporal properties and non-timed/timed refinement. Currently, we have built a lightweight framework for this approach. We believe that our approach will contribute to both eliminating the extra efforts for formally verifying nesC applications and increasing the correctness and reliability of nesC applications.

II. RELATED WORK

Rosa and Cunha propose an approach for formalizing TinyOS applications with a formal specification language LOTOS [4] based on process algebra. Main concepts of nesC, namely interfaces, modules and configurations, are specified as synchronization ports, processes with choices and compositional processes. This approach emphasizes mostly on modeling the interaction between components and introduces no verification techniques. McInnes presents a technique for modeling the concurrency of TinyOS applications using another process based specification language, i.e. CSP, and the model checker FDR is introduced to analyze the applications [5]. Both approaches contribute to applying formal methods to sensor network systems for enhanced analysis and early error detection. However, these approaches cannot yet avoid the trouble caused by constructing formal models manually.

The framework SLEDE [6], [7] is proposed for automatic verification of sensor network security protocols implemented in nesC. SLEDE extracts Promela models from protocol implementations in nesC, generates intrusion models and uses the model checker SPIN [8] to verify security properties of the models. The contribution of SLEDE is that it decreases the cost of verification and improves the quality of nesC security protocol implementations. However, this framework focuses on security protocol implementations and is not suitable for other nesC applications. Our approach differs from SLEDE in that we consider real-time behaviors during model generations and our approach is feasible for various applications besides protocol implementations.

Reference [9] introduces a model construction methodology for TinyOS-based networks using Behavior-Interaction-Priority (BIP) component framework. This methodology includes a model generation method for nesC. The networked system is specified as a global model, which is the composition of models of nodes. This framework allows behavioral verification and simulation of sensor network systems. This work differs from ours in that it uses BIP.
III. METHODOLOGY

Our approach is to automatically extract RTS models from TinyOS applications (implemented in nesC), and then to apply verification techniques to the generated RTS models. The verification results of the RTS models reflect the correctness of the original TinyOS applications. The TinyOS execution model and related RTS semantics are described in Section III-A. Section III-B explains in detail how our approach is designed and implemented in a framework. In Section III-C, an assertion annotation language is presented and examples are provided to demonstrate verification goals.

A. The Execution Model of TinyOS

TinyOS’s execution model is based on split-phase operations, run-to-completion tasks and interrupt handlers.

In effect, a task is a deferred procedure call. Tasks can be posted at anytime and are executed later one by one, managing by the TinyOS scheduler. A task runs atomically with respect to others. At runtime, TinyOS maintains a scheduler for managing the executions of tasks, using a queue for running tasks in FIFO order. The task scheduler can be represented as the state machine shown in Fig. 1.

In contrast to tasks, an interrupt handler can preempt tasks or other interrupt handlers. The interrupt manager is introduced to manage the executions and interactions between interrupt handlers. In the interrupt manager, a new interrupt preempts the execution of the current one. The state machine of the interrupt manager is shown in Fig. 2.

Since interrupts can preempt the execution of tasks, the global state machine of the execution model of TinyOS should capture such features, as presented in Fig. 3. Initially, the state machine is idle. When a new task is posted, it will be pushed into the task queue, and the task scheduler begins to execute. Similarly, initially an interrupt will begin the execution of the interrupt handler. Since interrupts preempt tasks, an interrupt can be accepted during the execution of the task scheduler, and the execution of the interrupt handler will begin, blocking the execution of the task scheduler. After the execution of the interrupt handler, the task scheduler will be resumed if the task queue is not empty otherwise the execution model will become idle again. The executions of the task scheduler and the interrupt handler in Fig. 3 follow the state machines in Fig. 1 and Fig. 2, respectively.

The formal definition of RTS models have been previously presented in [3], where an RTS model is defined as a 3-tuple $\mathcal{R} = (\mathcal{V}_r, init, P)$. $\mathcal{V}_r$ is a set of global variables, $init$ is the initial valuation of the variables and $P$ is a process. Based on this definition, the RTS semantics of the task scheduler and of the interrupt manager are defined as the following.

Definition 1 (Task Scheduler): The task scheduler $sd$ of TinyOS is modeled as an RTS model $\mathcal{R}_{sd}$, where $\mathcal{V}_r = \{Q, Ch(sd, 0), Cont(EOT, −1), tsk\}$ ($Q$ is a queue for deferred tasks, $Ch(sd, 0)$ is a synchronous channel for notifying a certain task to execute, $EOT$ is a constant variable with the unique value $−1$ denoting the end of a task, and $tsk$ is the id of the executing task), $init$ is the initial valuation of $\mathcal{V}_r$ such that $Q$, and channel $sd$ are empty, and $P = if (Q \neq \emptyset) \{getTask_{sd}(tsk) = Q.Pop() \rightarrow sdl!tsk \rightarrow sd?EOT \rightarrow P\} else\{P\}$.

Definition 2 (Interrupt Manager): The interrupt manager $im$ of TinyOS is modeled as an RTS model $\mathcal{R}_{im}$, where $\mathcal{V}_r = \{Q, Ch(im, 0), Cont(EOA, −1), async\}$ ($Q$ is a stack for preempted executions of asynchronous functions, $Ch(im, 0)$ is a synchronous channel for interrupting an asynchronous execution and notifying preempted executions to resume, $EOA$ is a constant variable with the unique value $−1$ denoting the end of an asynchronous execution, and $async$ is the id of the executing function), $init$ is the initial

$^{1}$Cont() is a function to define constants.
valuation of \( \text{Var} \) such that \( Q_i \) and channel \( \text{imr} \) are empty, and 
\[
P = \text{imr'}\text{EOA} \rightarrow \{\text{if}(Q_i \neq \emptyset)\{\text{getAsync}_\text{imr}\{\text{async}_\text{id} = Q_i.Pop()\} \rightarrow \text{imr'}\text{EOA} \rightarrow P\}\}param\{P\}.
\]

The following presents a formal definition of TinyOS execution model.

**Definition 3 (Execution Model):** A TinyOS execution model is a 7-tuple \((\text{Var}, Q_i, Q_0, S, A, s_0, \delta)\), consisting of:
- a set of variables declared in all components \((\text{Var})\),
- a queue for scheduling tasks \((Q_i)\),
- a stack for managing interrupt handlers \((Q_t)\),
- a set of reachable states \((S)\),
- a set of possible actions \((A)\),
- an initial state \((s_0)\),
- and a transition function \((\delta : S \times A \rightarrow S)\).

A state \(s\) is defined as the following definition.

**Definition 4 (State \(s\)):** A state \(s\) of the TinyOS execution model is a 2-tuple \((V, p)\) where \(V\) is the valuation of \(\text{Var}\), \(Q_i\), and \(Q_0\), and \(p\) identifies the current execution: \(o\)(idle), \(t\)(task) or \(i\)(interrupt handler).

The firing rules for \(\text{nesC}\) operations, such as \text{post}, \text{interrupt}, \text{return} and so on, are defined as below.

Below are the firing rules for a statement \text{post}(tk) (i.e. to post a task \(tk\)). \(t\{tk\}\) means that the current execution is in the task scheduler with task \(tk\).

\[
\begin{align*}
(V, o) & \xrightarrow{\text{post}(tk)} (V, t\{tk\}) \quad \text{[post1]} \\
p \neq o & \xrightarrow{} (V, p) \xrightarrow{\text{post}(tk)} (V, \{Q_i.\text{add}(tk)\}, t) \quad \text{[post2]}
\end{align*}
\]

In rule \text{post1}, the execution is idle, therefore, a newly posted task can be executed immediately and the execution becomes \(t\). In rule \text{post2}, a task or an interrupt handler is running. Since tasks can not preempt other executions, the newly posted task \(tk\) is added to \(Q_i\).

The following are the firing rules for \text{interrupt}(ih) (i.e. the occurrence of an interrupt with the handler \(ih\)).

\[
\begin{align*}
(V, o) & \xrightarrow{\text{interrupt}(ih)} (V, i\{ih\}) \quad \text{[interrupt1]} \\
(V, t\{tk\}) & \xrightarrow{\text{interrupt}(ih)} (V, \{Q_i.\text{push}(tk)\}, i\{ih\}) \quad \text{[interrupt2]} \\
(V, i\{ih_0\}) & \xrightarrow{\text{interrupt}} (V, \{Q_i.\text{push}(ih_0)\}, i\{ih\}) \quad \text{[interrupt3]}
\end{align*}
\]

Below are the firing rules for \text{return} (i.e. the completion of a task or interrupt handler). \(i\{ih\}\) means that the current execution is the interrupt handler \(ih\).

\[
\begin{align*}
Q_i & = \emptyset, Q_0 = \emptyset \quad \text{[return1]} \\
(V, o) & \xrightarrow{\text{return}} (V, o) \\
Q_i & \neq \emptyset, Q_0 = \emptyset \quad \text{[return2]} \\
(V, p) & \xrightarrow{\text{return}} (V, t\{tk = Q_i.pop()\}, i\{tk\}) \\
Q_i & \neq \emptyset \quad \text{[return3]} \\
(V, p) & \xrightarrow{\text{return}} (V, i\{ih = Q_i.pop()\}, i\{ih\})
\end{align*}
\]

### B. Extracting RTS model from nesC

Fig. 4 reflects the architecture of our framework. The input of the framework includes \(\text{nesC}\) source code and assertions representing verification goals. A self-contained parser for the \(\text{nesC}\) language is implemented to generate corresponding RTS models from source code. The generated models are then passed to the existing model checker of \(\text{PAT}\) to verify whether the goals are verified and counterexamples would be provided for unsatisfied goals. Executing graphs of the models and counterexamples can be viewed via the simulator of \(\text{PAT}\). \(\text{PAT}\) [10], [11] supports a wide range of modeling languages including \(\text{RTS}\) (Real Time System, a process algebra with timed operators), which shares similar design principles with integrated specification languages like TCOZ [12], [13]. Our framework is implemented as a module of \(\text{PAT}\) (i.e. \(\text{nesC}\) module) and is available at [14].

Since TinyOS system library (which encapsulates operating system environment and hardware functionalities as predefined interfaces and components) is required when running \(\text{nesC}\) applications, a set of RTS models representing the TinyOS library is statically provided (such as Timer, Sensor, Led, etc.). These environmental models are used when applying the mapping algorithm for model generation.

**Table I**

<table>
<thead>
<tr>
<th>nesC Elements</th>
<th>RTS Constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split-phase operations</td>
<td>Processes communicating via synchronous channels.</td>
</tr>
<tr>
<td>Component</td>
<td>Global variables, processes modeling behaviors of commands, events, and tasks.</td>
</tr>
<tr>
<td>System</td>
<td>The interleaving composition of all processes.</td>
</tr>
</tbody>
</table>
As shown in Table I, the mapping algorithm for extracting RTS models from nesC code consists of three steps: modeling split-phase operation, modeling components, and building a system-level process to model the application as a whole. Each step is explained in detail as follows.

**Step 1: model split-phase operations**

As shown in Fig. 5, components interact with each other via interfaces. A wiring in nesC "wires" a component to another by an interface and the two components are required to use and provide the interface respectively. Wiring is modeled by defining two processes transferring command calls and event signals between the user and the provider.

**Definition 5 (Wiring Expression):** A wiring expression \( \text{user} \rightarrow \text{prov} \) is modeled as an RTS model \( \mathcal{R}_{\text{user} \rightarrow \text{prov}} \) where \( \text{Var} = \{ \text{Ch(user},0),\text{Ch(user},0),\text{Ch(prov},0),\text{Ch(prov},0) \} \), init is the valuation such that channels \( \text{user}, \text{user}, \text{prov}, \text{prov} \) are empty, and \( P = (\text{CommandCalls} \mid \mid \text{EventSignals}) \). Further, \( \text{Ch}(c,n) \) defines a channel named \( c \) with a buffer of size \( n \) (a channel with 0 buffer size requires synchronous input and output), \( \text{CommandCalls} = (\text{user} \rightarrow \text{prov} \mid \text{CommandCalls}) \), and \( \text{EventSignals} = (\text{prov} \rightarrow \text{user} \rightarrow \text{EventSignals}) \).

**Step 2: model a component**

A component is separated into three parts: component variables, synchronous code, and asynchronous code. According to the execution model of TinyOS, the execution is either a task or a interrupt handler. The execution of a function runs to completion until it is interrupted. Each function (a task, command or event) is specified as a process, and a component is the interleaving composition of all the processes representing its implemented functions, as shown in Fig. 6. The statements in a function are modeled as RTS events or processes, as shown in Table II. To emphasize, tasks are scheduled in a task scheduler and do not execute immediately once it is posted as commands or events. A queue is defined for posted tasks to wait for being scheduled, and a channel is used for communication between tasks and the task scheduler.

**Definition 6 (Command Implementation):** A command implementation \( \text{cmd} \) is modeled as a RTS model \( \mathcal{R}_{\text{cmd}} \) where \( \text{Var} = \{ \text{id}_{\text{cmd}}, \text{v}_{\text{cmd}}, \text{v}_{\text{cmd}}, \ldots \} \) (i.e. the unique id of the command and the variables defined in the command implementation), init is the initial valuation of the variables, and \( P = \text{prov} \mid \text{id}_{\text{cmd}} \rightarrow \text{Cmd} \). Further, \( \text{Cmd} \) is an RTS process constructed by translating the statements of the command implementation.

**Definition 7 (Event Implementation):** An event implementation \( \text{evt} \) is modeled as a RTS model \( \mathcal{R}_{\text{evt}} \) where \( \text{Var} = \{ \text{id}_{\text{evt}}, \text{v}_{\text{evt}}, \text{v}_{\text{evt}}, \ldots \} \) (i.e. the unique id of the event and the variables defined in the event implementation), init is the initial valuation of the variables, and \( P = \text{user} \mid \text{id}_{\text{evt}} \rightarrow \text{Evt} \). Further, \( \text{Evt} \) is an RTS process constructed by translating the statements of the event implementation.

**Definition 8 (Task):** A task \( \text{tsk} \) is modeled as a RTS model \( \mathcal{R}_{\text{tsk}} \) where \( \text{Var} = \{ \text{id}_{\text{tsk}}, \text{v}_{\text{tsk}}, \text{v}_{\text{tsk}}, \ldots \} \) (id\(_{\text{tsk}}\) is a unique id for the task, and \( v_{\text{tsk}}, v_{\text{tsk}}, \ldots \), are the variables defined in the task), init is the initial valuation of the variables, and \( P = \text{sd} \mid \text{id}_{\text{tsk}} \rightarrow \text{Tsk} \mid \text{sd} \mid \text{EOF} \). Further, \( \text{Tsk} \) is an RTS process constructed by translating the statements of the task.

**Step 3: build the system-level process**

After specifying all components and their wirings, a top-level process is defined as the interleaving of all existent processes in runtime to represent the whole system, including the process modeling the task scheduler.

\[
\text{System} \triangleq \text{TaskSdl} \mid \mid \text{Comp}_\text{Sync} \mid \ldots \mid \text{Comp}_\text{Sync} \\
\mid \mid \text{Comp}_\text{Async} \mid \ldots \mid \text{Comp}_\text{Async};
\]

The task scheduler is modeled as a process (TaskSdl) as follows.

\[
\text{TaskSdl} = \text{if}(\text{Qt}.\text{Count}() \neq 0)\{
\text{getTask}[\text{ID}_{\text{Disk}} = \text{Qt}.\text{First}()] \rightarrow \text{sd} \mid \text{ID}_{\text{Disk}} \\
\rightarrow \text{sd} \mid \text{EOF} \rightarrow \text{deTask}[\text{Qt}.\text{Dequeue}()] \\
\rightarrow \text{TaskSdl}
\}
\]
Process TaskSdl is blocked until the task queue Qt is not empty, and then a task id is fetched from the queue and the corresponding task is informed to run via channel sdl. As shown in Definition 9, process Tsk (modeling a task) is blocked until the channel sdl has a value equaled to the task’s ID. After the execution of the task, it writes channel sdl with EOT to inform the process TaskSdl of its completion, and its id will then be dequeued from Qt.

Process System reserves the semantics of synchronous code and asynchronous code of nesC, i.e. a synchronous function runs atomically if not preempted by asynchronous functions and an asynchronous function can be preempted by other asynchronous function. The process System is then verified against assertions to prove whether the original application satisfies the defined properties. This process System is then verified with assertions to prove whether the original application satisfies the defined properties.

Modeling timing constraints TinyOS maintains middlewares such as Timer and Alarm to support the real-time features for developing nesC applications. The timing middlewares of TinyOS are modeled as timed processes with timed operators including Wait, Deadline, WaitUntil, Interrupt and so on, as illustrated in [3]. These models are implemented statically as part of the runtime environmental library when automatically constructing the RTS models in our framework.

C. Verification

1) Assertion Annotation Language: To ease the procedure of defining verification goals, an expressive and user-friendly assertion annotation language is defined. The annotation language includes assertions for defining deadlock freeness, divergence freeness, state reachability, temporal properties (as LTL formula) and timed refinement. It is used to define verification goals for checking data races, recursion of operation calls, existence of failures, timing requirements, safety and so on. Examples are given in Table III.

2) Verification Techniques: Various techniques are applied for different kinds of properties, such as deadlock freeness, divergence freeness, state reachability, temporal properties, and non-timed/timed refinement. Temporal properties are written in Linear Temporal Logic (LTL) formulae, which are converted into Büchi automata. As presented in [15], on-the-fly model checking approach and a depth-first search algorithm are used to search for a counterexample. Non-timed refinement technique has been introduced to model check linearizability in [16], which adopts partial order reduction for a modified on-the-fly refinement checking algorithm. Timed refinement checking is proposed in [3], using zone abstraction to preserve timed traces and developing model checking algorithm based on Difference Bound Matrix.

IV. EXPERIMENTS

In this section, we illustrate the execution of the current framework with the application BlinkTask, in which a led is turned on and off periodically. In its configuration, BlinkC is wired to TimerC, LedsC and MainC via interface Timer, Leds and Boot, respectively. We also run another application BlinkTask’ for comparison, in which 2 timers are used to toggle 2 leds periodically respectively. Our testbed is a PC with Intel Core2 CPU at 2.33GH and 3.25GB RAM.

During model extraction, timed operator Wait is used when modeling the command startPeriodic implemented by TimerC. The generated RTS model consists of variables, channels and processes. Three kinds of verification goals are verified against for both applications. P1(P1′) is the goal of deadlock freeness; P2(P2′) is the goal that the timer(s) is(are) fired infinitely often; P3(P3′) is the goal that the led should eventually be toggled whenever its corresponding timer is fired. The results of verification of both applications are shown in Table IV. These examples show that our approach is useful and efficient for verifying TinyOS applications. However, we still need to improve the current framework to support more complex applications without decreasing the efficiency, as discussed in next section.

V. DISCUSSIONS

Currently, our framework supports a large subset of the nesC syntax, and we are still working to support advanced
syntax such as multiple wiring and hierarchical components. Besides, we are facing challenges in the following aspects.

First, timing middlewares (such as Timer and Alarm) in TinyOS allow the absolute time to be accessed. A simple mechanism can be maintaining a global variable to represent the system time and increasing it periodically. This method is problematical because introducing an unbounded global variable will make the state space infinitely large. An alternative is to model absolute time as relative time, which will introduce more complexity.

Second, state explosion is a common problem while applying model checking techniques. As shown in Table IV, during the verification of desirable properties for BlinkTask’, more than 1 million states are visited. One solution is to refine the model extraction methodology for generating more abstract models. An alternative is to apply verification techniques such as symmetric reduction to reduce the state space. A second alternative is to apply verification techniques directly on nesC instead of any formal models.

Third, sensor networks are often required to work in unreliable environments, with lossy channels, unreliable data reading, etc. Events like loss of data are randomized, however, our current approach takes into account no probabilistic behaviors for simplicity.

In the near future, we will enhance the current framework to support the complete syntax of nesC. Meanwhile, the methodology will be refined for extracting models at different levels of abstraction and state reduction techniques will also be explored. The current framework models applications on one sensor node, and we will work on composing a model of the sensor networked system based on separate nodes. Moreover, we will improve the model extraction with randomized behaviors (such as data loss and communication failure) of sensor network systems, and then probabilistic model checking techniques can be applied to verify to our approach. Finally, we will improve our framework to be feasible for verifying various applications of sensor networks (such as security protocols, medium access control, power managements, sensing and communicating, data aggregation, radio control, etc.).

Due to the gap between the semantics of nesC and those of formalisms, formal models extracted from nesC are inevitably large, complex and redundant. Intuitively, direct verification is straightforward and more efficient as specialized optimization techniques are possible. However, direct verification is infeasible without a complete operational semantics of nesC. We expect to define the operational semantics of nesC, and generate Labeled Transition Systems (LTS) based on them. Verification techniques will be applied directly by exploring the LTS. However, to define the operational semantics is not an easy task, since currently there exists no formal description for nesC. Moreover, the diversity of nesC semantics makes defining its operational semantics more difficult. At present, we notice that the execution model of TinyOS application is well-defined. Therefore, we intend to develop a formal description of this execution model and then operational semantics can be defined based on it.

VI. Conclusion

This paper explains our recent work on automatic formal verification of sensor network systems. Currently, we have developed a framework for automatic generation of RTS models from nesC implementations. Our ultimate goal is to develop direct verification of nesC applications. We believe that our approach contributes to the robustness of sensor network systems because it allows nesC programmers to model check their code without being troubled by manually constructing formal models.

References