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*

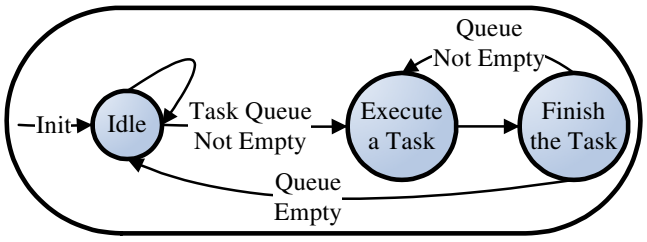

Figure 1. The Execution Model of Task Scheduler

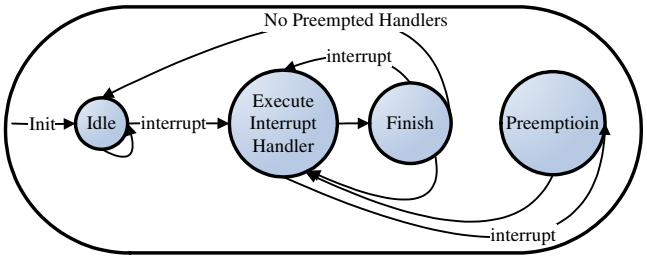

Figure 2. The Execution Model of Interrupt Manager

(i.e. hierarchical real-time components) while we uses *RTS* (i.e. process algebra with real-time extensions).

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

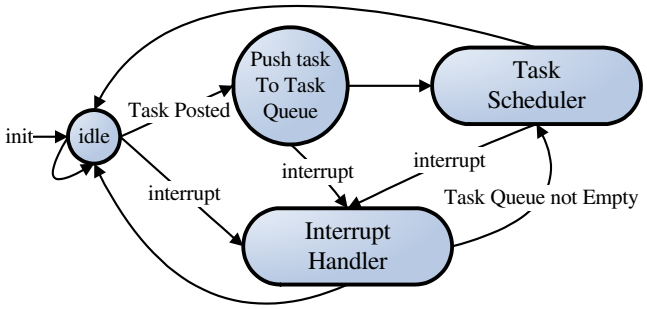

Figure 3. The Execution Model of TinyOS

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 $\mathcal{R} = (Var, init, P)$. *Var* 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 *sdl* of TinyOS is modeled as an *RTS* model $\mathcal{R}_{sdl}$, where $Var = \{Q_t, Ch(sdl,0), Cont(EOT,-1)^1, tsk_{id}\}$ (Q_t is a queue for deferred tasks, $Ch(sdl,0)$ 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_{id} is the id of the executing task), *init* is the initial valuation of *Var* such that Q_t and channel *sdl* are empty, and $P = if(Q_t \neq \varnothing)\{getTask_{sdl}\{tsk_{id} = Q_t.Pop()\} \rightarrow sdl!tsk_{id} \rightarrow sdl?EOT \rightarrow P\}else\{P\}$.

Definition 2 (Interrupt Manager): The interrupt manager *imr* of TinyOS is modeled as an *RTS* model $\mathcal{R}_{imr}$, where $Var = \{Q_i, Ch(imr,0), Cont(EOA,-1)^1, async_{id}\}$ (Q_i is a stack for preempted executions of asynchronous functions, $Ch(imr,0)$ 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 $async_{id}$ is the id of the executing function), *init* is the initial

[1]Cont() is a function to define constants.

valuation of *Var* such that Q_i and channel *imr* are empty, and $P = imr?EOA \rightarrow if(Q_i \neq \varnothing)\{getAsync_{imr}\{async_{id} = Q_i.Pop()\} \rightarrow imr?EOA \rightarrow P\} else\{P\}$.

The following presents a formal definition of *TinyOS* execution model.

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

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 $s = (V, p)$ where *V* is the valuation of *Var*, Q_t and Q_i, and *p* identifies the current execution: *o*(idle), *t*(task) or *i*(interrupt handler).

The firing rules for *nesC* operations, such as *post*, *interrupt*, *return* and so on, are defined as below.

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

$$\frac{}{(V,\mathrm{o}) \overset{post(tk)}{\rightarrow} (V, t\{tk\})} [\ post1\ ]$$

$$\frac{p \neq \mathrm{o}}{(V,p) \overset{post(tk)}{\rightarrow} (V\prime\{Q_t.add(tk)\}, t)} [\ post2\ ]$$

In rule *post*1, the execution is idle, therefore, a newly posted task can be executed immediately and the execution becomes *t*. In rule *post*2, a task or an interrupt handler is running. Since tasks can not preempt other executions, the newly posted task *tk* is added to Q_t.

The following are the firing rules for $interrupt(ih)$ (i.e. the occurrence of an interrupt with the handler *ih*).

$$\frac{}{(V,o) \overset{interrupt(ih)}{\rightarrow} (V, i\{ih\})} [\ interrupt1\ ]$$

$$\frac{}{(V, t\{tk\}) \overset{interrupt(ih)}{\rightarrow} (V\prime\{Q_t.push(tk)\}, i\{ih\})} [\ interrupt2\ ]$$

$$\frac{}{(V, i\{ih_0\}) \overset{interrupt}{\rightarrow} (V\prime\{Q_i.push(ih_0)\}, i\{ih\})} [\ interrupt3\ ]$$

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

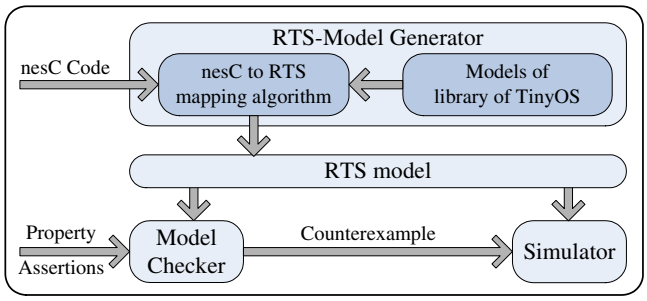

Figure 4. Overview of the framework

Table I
THE MAPPING ALGORITHM

nesC Elements	RTS Constructs
Split-phase operations	Processes communicating via synchronous channels.
Component	Global variables, processes modeling behaviors of commands, events, and tasks.
System	The interleaving composition of all processes.

$$\frac{Q_t = \varnothing,\ Q_i = \varnothing}{(V,p) \overset{return}{\rightarrow} (V,o)} [\ return1\ ]$$

$$\frac{Q_t \neq \varnothing,\ Q_i = \varnothing}{(V,p) \overset{return}{\rightarrow} (V\prime\{tk = Q_t.pop()\}, t\{tk\})} [\ return2\ ]$$

$$\frac{Q_i \neq \varnothing}{(V,p) \overset{return}{\rightarrow} (V\prime\{ih = Q_i.pop()\}, i\{ih\})} [\ return3\ ]$$

B. *Extracting* RTS *model from* nesC

Fig. 4 reflects the architecture of our framework. The input of the framework includes *nesC* source code and assertions representing verification goals. A self-contained parser for the *nesC* language is implemented to generate corresponding *RTS* models from source code. The generated models are then passed to the existent model checker of *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 *PAT*. *PAT* [10], [11] supports a wide range of modeling languages including *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 *PAT* (i.e. 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 *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.

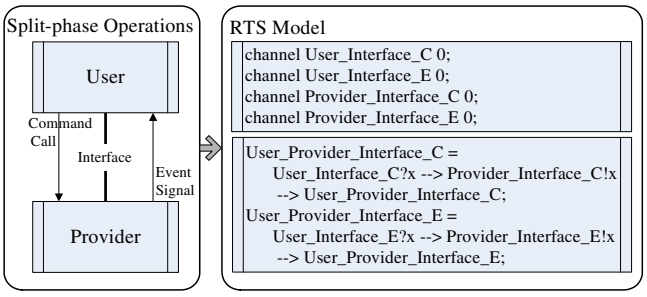

Figure 5. Modeling Split-phase Operations

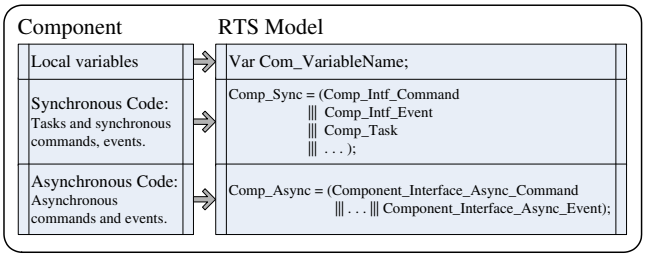

Figure 6. Modeling a component

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 $user \rightarrow prov$ is modeled as an RTS model $\mathcal{R}_{user \rightarrow prov}$ where $Var = \{Ch(user_c, 0), Ch(user_e, 0), Ch(prov_c, 0), Ch(prov_e, 0)\}$, *init* is the valuation such that channels $user_c$, $user_e$, $prov_c$ and $prov_e$ are empty, and $P = (CommandCalls \ ||| \ EventSignals)$. Further, $Ch(c, n)$ defines a channel named c with a buffer of size n (a channel with 0 buffer size requires synchronous input and output), $CommandCalls = (user_c?x \rightarrow prov_c!x \rightarrow CommandCalls)$, and $EventSignals = (prov_e?x \rightarrow user_e!x \rightarrow EventSignals)$.

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.

Table II
EXAMPLE: RTS PROCESSES FOR *nesC* STATEMENTS

nesC statements	**RTS processes**
call intf.cmd()	$prov_c!id_{cmd}$
signal intf.evt()	$user_e!id_{evt}$
post tsk()	$Q_t.Push(tsk)$
if(B){A}else{C}	IF = if (B){A}else{C};
while(B){A}	WHILE = if(B){A;WHILE}else{Skip};
do{A}while(B)	WHILE = A; if(B){WHILE}else{Skip};
for(A;B;C){D}	FOR = A; ReFor; ReFor = if(B){D;C;ReFor }else{Skip};

Definition 6 (Command Implementation): A command implementation *cmd* is modeled as a RTS model $\mathcal{R}_{cmd}$ where $Var = \{id_{cmd}, v1_{cmd}, v2_{cmd}, \ldots\}$ (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 $P = prov_c?id_{cmd} \rightarrow Cmd$. Further, *Cmd* is an RTS process constructed by translating the statements of the command implementation.

Definition 7 (Event Implementation): An event implementation *evt* is modeled as a RTS model $\mathcal{R}_{evt}$ where $Var = \{id_{evt}, v1_{evt}, v2_{evt}, \ldots\}$ (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 $P = user_e?id_{evt} \rightarrow Evt$. Further, *Evt* is an RTS process constructed by translating the statements of the event implementation.

Definition 8 (Task): A task *tsk* is modeled as a RTS model $\mathcal{R}_{tsk}$ where $Var = \{id_{tsk}, v1_{tsk}, v2_{tsk}, \ldots\}$ (id_{tsk} is a unique id for the task, and $v1_{tsk}$, $v2_{tsk}$, . . . , are the variables defined i n the task), *init* is the initial valuation of the variables, and $P = sdl?id_{tsk} \rightarrow Tsk$; $sdl!EOF$. Further, *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.

$$System \mathrel{\widehat{=}} TaskSdl \ ||| \ Comp\_Sync \ ||| \ \ldots \ ||| \ Comp\_Sync \ ||| \ Comp\_Async \ ||| \ \ldots \ ||| \ Comp\_Async;$$

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

$$\begin{aligned} TaskSdl \ = \ & if(Qt.Count() \neq 0)\{ \\ & getTask\{IDtsk \ = \ Qt.First()\} \ \rightarrow \ sdl!IDtsk \\ & \rightarrow \ sdl?EOT \ \rightarrow \ deTask\{Qt.Dequeue()\} \\ & \rightarrow \ TaskSdl \\ & \} \end{aligned}$$

Process *TaskSdl* is blocked until the task queue Q_t 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 Q_t.

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.

[2] $ledon_s$ is a state defined as $LedC.led == 1$, i.e. led is on.

[3] P_1 is a specification process, eg. $P_1 \equiv BlinkC.call.leds.ledOn \rightarrow LedC.leds.ledOn \rightarrow P_1$.

[4] P_2 is a specification process, eg. $P_2 \equiv Led.leds.ledtoggle; Wait[5]; P_2$.

Table III
SOME EXAMPLES OF ASSERTIONS

Type	Assertion	Property
Deadlock Freeness	*#assert System deadlockfree*	The system is deadlock free.
Divergence Freeness	*#assert System divergencefree*	The system is divergence free.
	#assert System divergencefree $< T >$	The system is timed divergence free.
Reachability	*#assert System reaches* $ledon_s$[2]	The system reaches the state $ledon_s$.
Temporal Properties	*#assert System* $\models \Box\Diamond$ *BlinkC.Timer*0*.fired*	*Timer0* is fired infinitely often.
	#assert System $\models \Box$(*BlinkC.Timer*0*.fired* $\rightarrow$ ($\Diamond$*LedsC.Leds.led*0*Toggle*))	*led0* should eventually be toggled whenever *Timer0* is fired.
Refinement	*#assert System refines* P_1[3]	The traces of the system is a subset of those of P_1.
	#assert System refines$_{<T>}$ P_2[4]	The timed traces of the system is a subset of those of P_2.

Table IV
VERIFICATION RESULTS

System	Assertion	Result	Visited States	Time (second)
BlinkTask	P1	True	397	0.18
	P2	True	1,926	0.50
	P3	True	1,875	0.55
BlinkTask'	P1'	True	158,668	78.27
	P2'	True	1,397,580	1,420.72
	P3'	True	1,238,588	1,039.30

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 explore state reduction 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

[1] J. Hill, R. Szewczyk, A. W. an S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *PLOS'00*, 2000, pp. 93–104.

[2] 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'03*, 2003, pp. 1–11.

[3] J. Sun, Y. Liu, J. S. Dong, and H. H. Wang, "Verifying stateful timed CSP using implicit clocks and zone abstraction," in *ICFEM'09*, 2009.

[4] N. S. Rosa and P. R. F. Cunha, "Behavioural specification of wireless sensor network applications," in *GIIS'07*, 2007, pp. 66–72.

[5] A. I. McInnes, "Using CSP to model and analyze TinyOS applications," in *IEEE ECBS'09*, 2009, pp. 79–88.

[6] Y. Hanna and H. Rajan, "Slede: framework for automatic verification of sensor network security protocol implementations," in *ICSE Companion'09*, 2009, pp. 427–428.

[7] Y. Hanna, H. Rajan, and W. Zhang, "Slede: a domain-specific verification framework for sensor network security protocol implementations," in *WISEC'08*, 2008, pp. 109–118.

[8] G. J. Holzmann, "Software model checking with SPIN," *Advances in Computers*, pp. 78–109, 2005.

[9] A. Basu, L. Mounier, M. Poulhiès, J. Pulou, and J. Sifakis, "Using BIP for modeling and verification of networked systems – a Case study on TinyOS-based networks," in *NCA'07*, 2007, pp. 257–260.

[10] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "PAT: towards flexible verification under fairness," in *CAV*, 2009, pp. 709–714.

[11] J. Sun, Y. Liu, J. S. Dong, and H. H. Wang, "Specifying and verifying event-based fairness enhanced systems," in *ICFEM*, 2008, pp. 5–24.

[12] B. P. Mahony and J. S. Dong, "Timed communicating Object Z," *IEEE Trans. Software Eng.*, vol. 26, no. 2, pp. 150–177, 2000.

[13] ——, "Blending Object-Z and Timed CSP: an introduction to TCOZ," in *ICSE*, 1998, pp. 95–104.

[14] "PAT website," http://www.comp.nus.edu.sg/∼pat/.

[15] J. Sun, Y. Liu, J. S. Dong, and J. Sun, "Bounded model checking of compositional processes," in *TASE'08*, 2008, pp. 23–30.

[16] Y. Liu, W. Chen, Y. A. Liu, and J. Sun, "Model checking linearizability via refinement," in *FM'09*, 2009, pp. 321–337.