Compositional Verification of Sensor Software Using Uppaal

Mustafa Hammad  
Department of Computer Science  
Mu'tah University  
Jordan

Jonathan Cook  
Department of Computer Science  
New Mexico State University  
Las Cruces, NM 88003  
Email: joncook@nmsu.edu

Abstract—Verification of wireless sensor networks has long been performed for communication protocols and for network-level behavior over multiple nodes, but not for the basic properties that should hold at a single node. Testing sensor networks, however, is extremely hard due to the lack of controllability, and complex simulation setups are often too expensive to undertake. Thus, verification of properties for a sensor node is desirable. We created a verification methodology that extracts timed models of the high-level behavior of a wireless sensor and then uses UPPAAL to verify both functional and non-functional (timed) properties for the sensor. This verification capability will enhance the trustworthiness of deployed sensor networks.

Keywords—verification, wireless sensor networks, compositional analysis

I. INTRODUCTION

Wireless sensor networks (WSNs) are becoming a common type of embedded computing system, where small computers with local sensors actively sense their environment and communicate this information to the larger system. Much research has been focused on the WSN behavior at the network level, including designing and verifying various communication and data reduction protocols (e.g., [1]), but less has been pursued at the individual node level (e.g., [2], [3]). While per-node behavior can sometimes be a relatively simple wakeup-sense-send cycle, message forwarding and routing and more complicated behavior schemes can combine to make per-node behavior less straightforward. Testing WSNs is quite hard because setting up real-world conditions in a lab, or even in a simulator, can be tedious and error-prone. Testing WSN software usually does not go beyond sanity checks and initial deployment check-outs.

While this is often suitable for the straightforward repetitive functional behavior of the WSN nodes as they operate under normal, average conditions, it is doubtful to provide much help in understanding whether or not under abnormal conditions more subtle functional requirements might be satisfied or not (e.g., a data measurement be lost), nor much help for non-functional requirements (e.g., can the measurement interval drift beyond a given duration). To address these question we have constructed a methodology for verifying these types of properties using the UPPAAL real-time model checker [4]. Our methodology is compositional so that it can be reused by others to create new verifications for future property checking needs.

Specifically, in this work we target wireless sensor applications written on the TinyOS platform. TinyOS and its associated nesC programming language is an open-source WSN platform that is commonly used for low-resource WSN devices [5], [6]. The semantics of nesC and TinyOS create interesting aspects that must be handled when doing verification.

Our main contributions in this paper include: an UPPAAL definition of the semantics common to all TinyOS/nesC applications, which can be reused; a methodology for extracting relevant application semantics suitable for a variety of verification needs; and a methodology for extracting real timing data from the application and including it into the UPPAAL model for purposes of real-time property verification. With these capabilities, WSN applications can be verified to ensure they satisfy desirable properties even under hard-to-test abnormal conditions.

Section II presents background information on nesC, TinyOS, and the UPPAAL verification tool. Section III presents the detail of our verification methodology, and Section IV presents examples of its use and an evaluation of its capabilities and performance. Section V presents related work, and Section VI concludes with final insights and ideas for future work.

II. BACKGROUND

A. NesC and TinyOS

NesC is a programming language designed for small embedded computers needing event-driven, reactive software [5], and is integral with the TinyOS runtime environment [6], which itself is mostly written in nesC. NesC/TinyOS applications are made up of asynchronous and synchronous code, where time-sensitive code that is executed as the result of a hardware interrupt is asynchronous, and other code is synchronous. Asynchronous code is invoked beginning at asynchronous event handlers. For synchronous code the task is the main unit of scheduled execution, and is always the starting point for synchronous code. Tasks can call commands, invoke functions, and signal synchronous events which invoke synchronous event handlers.
Synchronous code is non-preemptive and executes as it is invoked, other than the act of posting a task. A task posting, whether executed from asynchronous or synchronous code, simply places the task on a task queue for later execution. In TinyOS 2 each unique task has one slot in the scheduler. Thus a task posting succeeds if there is no current outstanding posting of that task or if the task is currently running; but if a previous posting of the task has not yet executed then the new posting will fail.

Asynchronous event handlers are most often low-level routines that respond to hardware events, and typically just post a task that will generate a synchronous version of the event. Thus applications are generally almost all synchronous code. The overall nesC/TinyOS execution model is one of quiescent waiting for events, which then trigger some computation, starting at the event handler and possibly invoking commands and posting tasks; once all tasks are done, execution then returns to the quiescent state, waiting for the next event.

B. Timed Automata and Uppaal

UPPAAL is a finite-state verification toolset that includes the ability to perform real-time property verification [4]. We only provide a very brief description here; for more information please refer to the citation or the many other resources for UPPAAL.

UPPAAL’s fundamental modeling formalism is timed automata, which have clock variables and constraints over the clocks, in addition to the standard automata formalisms of states and guarded transitions with actions. UPPAAL models also include synchronization channels, which are used to compose complete models out of independent automata. Transitions which produce an event on a named channel (synchronized with a !) are synchronized (they fire simultaneously with) all enabled transitions that are waiting for the event on the channel (synchronized with a ?). This composition allows UPPAAL to model communicating concurrent processes, and also allows models to be built compositionally, which we take advantage of in order to model the generic parts of TinyOS/nesC separately from the application-specific parts. This separation also enables a form of compositional verification where one model component, once verified, can be replaced with a simplified version that, e.g., only models a minimum and maximum delay between its interactions with the rest of the model. UPPAAL in general verifies complete models, but the parts can be composed in different ways to target specific property verification.

UPPAAL states marked with a ‘U’ or a ‘C’ are “urgent” and “committed” states, respectively. These states do not consume time and must be transitioned out of before any time-consuming states are, with committed states having the highest priority. Initial states are marked with a double circle; there are no final states.

Properties are specified in a subset of TCTL (timed computation tree logic), which supports the typical types of properties that most verification systems support, examples of which will be seen in Section IV.

III. Modeling Methodology

To construct a verification model from a TinyOS application, the A nesC/TinyOS application program is first built normally into a native sensor node executable—for our present analysis the framework requires the Atmel AVR platform, but the approach is generic. Our Time Extraction step uses the Avrora cycle-accurate AVR simulator [7] to extract timing information for application code blocks. Then, the Model Generation component uses the application source code and the timing information, along with pre-defined model templates to create a complete UPPAAL model. This is used by UPPAAL to verify user-defined properties. The models can also be used in simulation mode in the UPPAAL framework, and can be used to simulate counter-examples if verification fails.

A. Modeling TinyOS and NesC Semantics: Overview

Part of the complete UPPAAL model must capture the general semantics of TinyOS and the nesC language. TinyOS and the application it runs is one single-process entity; it is not a multi-threaded operating system. It implements at least part of the semantics of the nesC programming language, along with providing an abstraction layer of the hardware and a place to connect in device drivers and other hardware interfaces.

The most important part of the nesC semantics that relate to modeling are the task and event concepts (described in detail in Section II-A). Tasks are run at some point later than they are posted, and this requires TinyOS to have at least a rudimentary task scheduler in it. Tasks can also be interrupted by asynchronous events, and so must be able to be stopped and restarted.

External events that result from some hardware condition (e.g., a timer expiring or a radio message received) can occur at any time relative to the program and are at some level input-dependent, though that “input” could be as simple as a hard-coded timer interval.

Figure 1 shows the general idea of our modeling approach; the essential program components of event handlers, tasks, and commands are modeled as individual timed automata, along with a task scheduler automaton. These automata trigger each other using UPPAAL synchronization channels, named as the transitions in the figure. The asynchronous event models are automata that model external events that trigger application behavior. As will be seen, we also use some generic automata for the purposes of keeping track of generic execution attributes that are useful for some types of property verification.
B. Modeling TinyOS and NesC Semantics: Detail

Figure 2 is the most generic component of our model. It represents the overall duty cycle of the WSN node; that is, the amount of time the node is idle, and the amount of time it is active (performing computation). Real-time properties relating to the duty cycle can be verified with this component, although since UPPAAL does not support arithmetic on clock values, a property over the actual fraction of active versus idle time cannot be specified. The most straightforward properties supported by this automaton are those relating to the minimum and maximum active and idle periods.

Transitions from idle to active are triggered by hardware interruptions (stop?) and task executions (run[t]?), while transitioning back to idle is accomplished if a task finishes and no other tasks are left to execute (done[t]?), or if an interruption finishes and no tasks are running (run_task?). The clock y counts time in the active state and the clock x counts time in the idle state.

Figure 3 shows the two-level modeling of tasks, with (a) being the generic task model and (b) being the application-specific task model, where the state marked “task body” represents a single-entry, single-exit automaton that is extracted from the application specific code (as described in the following subsection). One automaton of each is instantiated for each task in the application.

The generic model exists so that generic properties can more easily be formulated and verified. For example, one can check whether a task is ever interrupted, or whether its posting ever fails, or if the delay between posting and
execution is ever above a threshold. These types of questions have nothing to do with the internals of the specific task. The generic model has three failure states that capture when the posting of a task fails; since they are marked with a ‘C’ they are committed zero-duration states that do not affect the model timing but only exist to aid in property verification. Recall that each unique task has one slot in the scheduler and if the slot is already occupied then the posting fails. This can occur if the waiting task has one slot in the scheduler and if the slot is already occupied then the posting fails.

The specific task automaton will contain timing constraints and the generation of command invocations, event signals, or postings of other tasks. The generic and specific task automata synchronize on the shared named channels run[taskId] (produced by the task scheduler) and done[taskId] (produced by the specific task automaton). They also both respond to the stop/go interruption of asynchronous events. Section III-C describes how the task body is generated, and Section IV presents an example of an actual application-specific task automaton, with the task body expanded.

The TinyOS task scheduler is modeled in UPPAAL as shown in Figure 4. Its responsibility is to select a task that has been posted and run it. One should not think of it as a free-running operating system scheduling process. WSN devices need to be able to “shut down” to a very quiescent state in order to conserve power. Essentially, every asynchronous event handler that was triggered from hardware, after it finishes, tells the task scheduler to go ahead and run any tasks it might have to run by using the run_task channel. When tasks finish they also tell the scheduler to run any other pending tasks (done[t]). When no tasks are available to run the scheduler goes back to the idle state.

The UPPAAL model must also contain parts that represent the events in the external world that trigger behavior in the WSN application. WSN nodes typically have only three mechanisms by which they can be kicked out of an idle state and “wake up” to respond to the event. These three are timer events, radio receive events, and sensor events.

Figure 5 shows the UPPAAL representation for timer events. These are instantiated for whatever timers the application program creates, as indicated in the processing of the application source code. Timers are started with set[timerId], and are either one-shot timers, in which case they return to the initial (idle) state, or repetitive “free-running” timers which fire at the regular specified interval. One timer automaton is instantiated for each application timer that is created, and it is connected to the particular task that responds to that timer.

Figure 6 shows the sensor event model. Sensors are not free-running in the sense that the application first asks them to perform a reading, and then at some point later they complete the task and the data is ready, at which they signal an event to the application. Depending on the complexity of the sensor, this may happen on a known, fixed delay, or may be variable. For example, a simple light sensor using an analog-to-digital conversion will have a short, fixed response time, while a GPS unit may take a variable
amount of time depending on its satellite reception status. Figure 6 uses the \texttt{call(commandId)} channel to indicate the application’s request of sensor, and then somewhere between the minimum and maximum delay times the model will generate a \texttt{post[taskId]} to indicate the data is ready.

Figure 6 also shows a common time modeling mechanism we use in several model components. A clock \( x \) is used to model the expected duration of some computation, but since interrupts may occur we cap the clock \( x \) at the expected maximum so that the model can proceed cleanly, and then use a clock \( y \) to capture the actual time elapsed. It is the clock \( y \) that is used in property formulas.

Radio receive events need a slightly more complicated representation, to allow bursts but disallow never-ending continuous network traffic. If we simply modeled a minimum and maximum interval between incoming packets, then UPPAAL would include in the model space a continuous reception of packets at the minimum interval time, and this would vastly skew verification of such properties as duty cycle (since message handling would dominate). In reality, network traffic is bursty. For example, if we have a network of 5 nodes, all that implement a wakeup-sense-send cycle, then we might have bursts of four received packets, but we could not have a continuous reception of packets on the minimum interval.

Thus, we model radio receive events to model this burst capability, as is shown in Figures 7 and 8. We specify a burst min/max rate, a min/max burst size, and a single between-burst rate. Figure 8 models the between-burst rate and the burst size (by setting the \texttt{receive_max} variable), and Figure 7 models the burst itself.

We should note that our sensor model and more especially the radio packet model, while satisfactory for most basic modeling needs, could be easily replaced by a more complex model if a user decided they needed it. Because the entire model is compositional, these components could be substituted with more complex models for a particular verification scenario that required them.

Event handlers are modeled in exactly the same form as task bodies, shown in Figure 3b, except that the triggering channel is \texttt{fire[eventId]}? rather than \texttt{run[taskId]}?. As with tasks, the actual event handler body determines the internal contents of the automaton, and one automaton is instantiated for each event handler. Applications typically (and are encouraged) to have only synchronous event handlers, but TinyOS does provide interfaces that expose asynchronous events. Asynchronous handlers will be connected directly to the asynchronous event models which produce those events.

C. Extracting an Application Model

In our methodology we extract an UPPAAL model representing the high-level behavior of the application. A high level view of a TinyOS/nesC application is one of connected modules composed of tasks, events, commands, and functions. We model the application at the level of these features, and only capture source-level behavior at those points where the execution of the high-level features is controlled. For example, an if-else construct that did not contain any task postings, event signaling, or command or function invocation would not be modeled, but one that did contain at least one of these would be modeled.

Part of the reason to model at this level is to control the model size, but it also reflects what can actually be measured in a compiled TinyOS application. Because TinyOS is included in the application at compile time, many of the “insignificant” calls within the source code disappear due to compiler optimizations, especially inlining. This is a purposely designed feature of TinyOS, but it also means that trying to model low-level features of a TinyOS application would be incompatible with populating the model with actual execution times.

It is over these high-level features that property verification will be formulated; for example, whether a task can ever be interrupted, or if an event signal can occur within some time period. Tasks and events, and to some extent commands, represent the “visible” behavior of the application and are thus that over which properties might be formulated.

One big simplification we make is that we assume loops do not contain these high-level features at all; we simply do not model loops. Our rationale for this simplification is that in our experience, WSN programs are most often essentially composed of constant time functionality that can be modeled with single duration values. In places where the code is written with loops, it most typically is with counting loops that walk over a fixed size data structure. For example, GPS sensors typically return their information in an ASCII string.
format; a routine that parses this string and extracts this information may have several loops and small conditional branches (e.g., handling years beginning with “19” or “20”), but the string is always a fixed size and the data extracted is always the same, and thus the routine runs in essentially a constant time (modulo perhaps a couple of instructions for small branches). Similarly, a routine extracting data from a fixed-size message with a known format will always run in within a small constant time bound.

WSN applications essentially obtain their repetitive high-level behavior from external events (e.g., timer events and radio message receive events), not loops. This event-level repetitive behavior is captured in the structure of our model, and we believe is sufficient for most WSN applications. In any case, some form of a bounded loop assumption is necessary to perform real-time property verification; if a program has the potential to perform unbounded computation at some point, then any real-time constraint that involves that computation will necessarily not be satisfied.

For example, of the 16 applications included in the TinyOS distribution (which represent typical applications), 3 share a “BaseStation” module that contains 2 constant loops, 2 others have a total of 3 short constant loops, and one (TCPEcho) has 3 loops that parse an HTTP message header, which can be reasonably bounded. Since TinyOS itself gets compiled together into a single application binary, we inspected the disassembled code from the “Blink” example (which had no loops in the application code). In the machine code, constant loops can be readily identified by the register usage and the compare instructions that control the loop branches. In the Blink+TinyOS program there are 6 constant loops, 11 3-instruction wait loops that wait for an interrupt to modify a global variable, and then the main unbounded control and scheduling loop that runs the application. While we did not track the exact purpose of each interrupt wait loop, these would be controlled by timers or other external events, and so their behavior will be subsumed by our model’s clocks and state structure. Thus internally, TinyOS seems to satisfy our assumptions as well\(^1\). We should note that if, e.g., a particular application had a task that used a loop to generate a bounded \(N\) radio messages or events, this could be modeled by hand and plugged into the task body part of our model; we just do not currently have the tools to do this automatically, as it seems most applications do not need it.

We use UPPAAL clocks to model the passing of time in the program. To capture the duration of the code that precedes, say, the invocation of a command, we set a clock and then let the model occupy a state for at most \(max\) time and enable the transition at \(min\) time, thus allowing the model to proceed at least at \(min\) time. This method can be seen in Figure 5, which models timer events. Since our model only captures conditional execution as it relates to task/event/command production, for a block of code that does not produce these the minimum and maximum execution durations capture the variability in that code due to taking different execution paths, possibly including loop iterations. Also, for verification purposes it is not important to know the distribution of expected times within that interval, since verification checks whether any possible execution duration can violate the property. This most often involves the duration boundaries and not the statistically average duration somewhere in the middle.

Conditional branches which can result in different sequences of the modeled application structures (commands, events, tasks) are modeled simply as non-deterministic branches; verification will explore both behavioral paths. We use this only when code paths can result in different high-level actions; otherwise the varying behavior is captured in the minimum and maximum time values for the code segment.

D. Extracting Application Timing

Avrora is a cycle-accurate simulator for the Atmel AVR processors, which are used in several wireless sensor devices [7]. It has specific support for simulating WSN-based programs, especially those based on TinyOS. In simulating these programs it produces an accurate count of CPU cycles that the application uses, and thus can produce an accurate timing profile of the application.

Avrora is extensible in that plug-in monitors can be created which perform particular queries and control the simulation. We use this extensibility to capture block-level timing of the application program. By block we do not mean basic blocks, but sections of code from an entry point to when “something significant” happens: a task posting, command execution, or event signaling. In Avrora monitors it is straightforward to access the execution time between two points in code.

In conjunction with the simplifying assumptions presented in Section III-C in extracting the structural model of the application, we do the same for extracting the timings. This means that we obtain single timing durations for blocks of code that could possibly contain loops.

Although the timings of code blocks are constant, the structural model does include execution branches where significant behavior can differ (e.g., sending a message or not), so we still are capturing the higher level time-varying behavior of the application. We are just abstracting out the detailed behavior that in practice is constant time. The real time-varying behavior of a WSN application comes from the generation of the asynchronous events, which model external activations of the node.

These times then are integrated into the UPPAAL model so that the model reflects the real timing behavior of the

\(^1\) The TinyOS thread extension, TOSThreads, does not have this simplified structure that our models capture, and in this work we are making no attempt to address and handle the issues related to TOSThreads.
application. The times extracted from Avrora are measured in cycles, so our models deal with logical “cycle” time, but which is easily converted to real time. Cycle counts map more easily to the integer clocks in UPPAAL than does some arbitrary selection of a minimum unit of time.

IV. EVALUATION

Our application models are suitable for the verification of a wide variety of properties, many of which we probably have not thought of. In addition, because of the models’ modularity it is relatively straightforward to augment a model with some particular counter or clock in order to verify a new property which the model already covers structure-wise. On the other hand, we acknowledge that our models are only capturing the high-level behavior of an application, and skip details which some desirable properties would need. For example, low-level properties such as buffer overflows are only capturing the high-level behavior of an application, and skip details which some desirable properties would need. For example, low-level properties such as buffer overflows are entirely out of the scope of our modeling framework.

A. Example Properties for Verification

Here example properties are presented that the models and methodology support.

From the automaton in Figure 2 we can support properties over the overall activity of the WSN node. For example, with SN being the name of the automaton, the property (written in UPPAAL syntax)

\[ A[] (((SN.activated and SN.idle and SN.x==0) imply (SN.y >= activeMin and SN.y <= activeMax))) \]

verifies that all active periods of the node are not shorter in duration than activeMin and not longer than activeMax. The formula states that the automaton must have been activated at least once, currently in the idle state with the idle clock x reset, and then checks the “final” value of the active clock y that was set from the active state.

Our modeling structure supports verification of many task-related properties. For example, if process P is derived from the model of Figure 3 (generic task), then we can verify the following:

- Does posting the task ever fail?
\[ E<> (T.fail1 or T.fail2 or T.fail3) \]
- Can the task be posted while it is running?
\[ E<> (T.running_posting) \]
- Can the task be posted while it is interrupted?
\[ E<> (T.interrupted_posted) \]

On the other hand, the task body model can be used to verify properties about the internal task code. For instance, Figure 9 shows the model for a specific task that contains two code segments delineated by a call to a send command, which is abstractly modeled as a sensor event automaton (Figure 6) that posts a task that fires a synchronous sendDone event. If T is derived from this model, then we can verify detailed properties such as:

- Can the first code segment be interrupted?
\[ E<> (T.interrupted1) \]
- Can the send command be delayed by X?
\[ A[](T.segment1 imply T.y < T1+X) \]

If we are interested in verifying how many times the task might be interrupted, we can add an integer counter that is incremented whenever an interruption happens; then the maximum number of interruptions can be checked by the formula

\[ A[] (T.end imply T.interruptCounter <= Max). \]

In general, if users need other properties not immediately captured by the existing clocks and variables, the models can be augmented with other clocks and variables to support such properties. With care not to change the behavior of the model, new states can be even added for specific checks. For example, the property of whether a periodic sensor read can ever be skewed by X time units from the designed rate can be supported as follows. The model in Figure 6 is used to represent the sensor command; to verify the above property we add a committed location L before the processing location to which the call[commandId] edge is connected. An edge from location L to processing is created with activated=true and time=0, where ‘activated’ is a boolean variable initialized to false and ‘time’ is a clock. If process P is derived from this model, then the formula for this property is:

\[ E<> (P.L \text{ and } P.activated \text{ and } (P.time - setPeriod) > X). \]

That is, the value of the time clock is the duration from the last sensor event to the current one, and the property is checked just before the clock is reset again. The activated flag simply ensures that the property is checked only on the second and succeeding events, so that there truly is an interval period to check.

If other asynchronous events interrupt timer processing, then an application’s response to a timer could be skewed by some amount. To check if the timer skew (drift) in a timer event model T is less than some maximum allowed:

\[ A[] (T.drifted imply T.x <= Max) \]
The generic task model allows us to ask whether the delay from task posting to task execution in task T is within a maximum bounds:

$$A[(T.waiting \implies T.x \leq Max)]$$

These are examples of properties that can be checked in our methodology.

B. Performance Evaluation

To verify our methodology and its usefulness, we performed some actual field tests to see if the models are indeed accurate. In this evaluation we used a monitoring framework that we have previously created to acquire hardware timing measurements [8]. The platform used was the Crossbow Mica2, which is based on an AVR processor. The clock speed of the processor is 7.3728MHz, which is used to convert from cycles to seconds.

As related earlier, creating actual lab test cases that approach a maximal load (which is where verified properties are likely to fail) is not easy, and so the application described is somewhat artificial in order for us to be able to conduct field tests.

We built three variations of a WSN application to measure the execution time of a large task that toggles three LEDs 1000 times (making it long enough to measure external interruptions). This task is posted from a timer that fires every quarter second. Figure 10 show the task body for this BigTask application, where the time limits are created from Avrora measurements, as our methodology specifies. The rest of the model is not shown due to space limitations.

First, we used our model to verify the maximum time needed to execute this task without any interruptions, which verified to 229,307 cycles. Secondly, we added a radio-receive component to this application and added two other sensors sending dummy packets at a rate of one packet every quarter second. Finally we added four other sending sensors configured the same as the two-sensor case. These two configurations resulted in radio communication automata where packets can be received in bursts of two and four, respectively, with the time between bursts set to 1,843,200 cycles (1/4 second). Within the burst, the packet rate is a minimum of 223 cycles (the fastest that a packet can be responded to, as measured by Avrora), and a maximum of 154,684 cycles for the two-sender case and 38,671 cycles for the 4-sender case (taken from sending times measured in Avrora; however, the maximums do not play a role in the properties we are verifying). For these three applications and models we measured the real task execution time from the hardware and then verified the following properties:

1) the execution time of the task is within the maximum value (229307 cycles):

$$A[\{P.processing \implies P.y \leq 229307\}]$$

2) the task will not be interrupted (property reversed in formula):

$$E<> (P.interrupted)$$

3) the number of possible interruptions is from 1 (Min) to 2 (Max):

$$A[\{P.end \implies (P.intCount<=2 \text{ and } P.intCount>=1)\}]$$

The results are shown in Table I. Also in the table are results from the same applications running on actual hardware; shown are the maximum task durations over ten experiments. What first is seen is that in the verification, both configurations with radio communication fail to satisfy the non-interrupted execution time property of the model, and indeed in the hardware measurements we see examples of the task duration being longer than that time, while in the no-radio version we do not see a violation of the property, in agreement with the verification result. This is at least initial validation of the timing measurements taken from Avrora and put into the model; they match what we observe on the hardware. Secondly, the properties of whether the task is interrupted or not, and then limits on how many interruptions can occur, are consistent with the applications, where only the no-radio version satisfies the no-interrupt property, and the 4-sender version does not satisfy the 2-interrupt property.

To evaluate the efficiency and scalability of our UPPAAL models, we used the Memtime tool [9], which is developed by the UPPAAL team to measure time and memory consumption during verification. Four properties were checked on different sensor sets. These properties are:

1) deadlock freedom,
2) does posting a task ever fail,
3) task execution time is within a given time, and
4) task delay will be within a given time.

The experiments were run on a machine with an AMD Sempron 1.8 GHz CPU running the Linux operating system. Results are shown in Table II. Note that there is a significant increase in memory allocation size when we added the radio models to the basic application model.

The first conclusion from this table is that all of the
analyses are easily doable, not even closely approaching system limits or overly long execution times. And while adding in the radio communication component significantly increases the model size, adding in more potential interrupt components does not significantly increase memory size. The execution time of verification does increase with more interrupt components, since it increases the depth of the state space model. But even here, 16 sending neighbors for a WSN node is modeling a very crowded network, and realistic models will not go much beyond this. Note that the time on the first two properties increases significantly down the columns because those properties do verify as satisfied, so the entire model space is analysed; properties three and four fail at the higher number of interrupt components, and so the verifications stop as soon as a failing counter-example is found.

V. RELATED WORK

Li and Regehr proposed T-Check [10], a tool to verify safety and liveness properties for sensor network applications running under the TinyOS platform. It works as a random tester or a model checker. The model checking is based on Killian et al.’s work [11]. The basic idea is that liveness violations can be detected heuristically by finding sufficiently long violations of the property. T-Check is based on the TOSSIM simulator [12] and Safe TinyOS assertions [3]. T-Check adds non-deterministic events to trigger corner case bugs, and by using TOSSIM they can abstract away some low-level hardware details. T-Check ignores the local execution time behavior and focuses instead on network behavior.

Green et al. [1] used UPPAAL to verify the real-time logic of their sensor network (Smart Dust). They developed a per-node timed automaton, duplicated it for each node in the network, and created another automaton to model the basestation behavior. Communication among nodes is modeled using channels. This model allows them to verify some basic properties about the network, such as whether a node ever communicates with the basestation and whether a sensor eventually goes to sleep. The model they created was for their specific application and they did not propose a general methodology.

Bucur and Kwiatkowska [2] used the CBMC bounded model checker for C to verify TinyOS programs written for the MSP430 platform. They modeled parts of the MSP430 hardware, created an efficient mechanism for integrating into the C code the interrupt handlers that result from the nesC toolchain, and created pre-defined verification properties, such as array bounds checking. Their work does not perform real-time verification, and by using a bounded model checker, properties are proved over a limited system model and thus are highly likely to hold, but not guaranteed.

Coleri et al. [13] verify TinyOS systems by using HyTech, a model checker for linear hybrid automata. The sensor network is modeled to analyze the energy lifetime of the network. Their focus is on power consumption and they only model the local application insofar as they accurately model power consumption in response to external events.

Kothari et al. [14] proposed FSMGen, a tool to generate finite state machines from TinyOS programs using Symbolic Execution. The generated FSMs aid in program understanding, error detection, and program validation. FSMGen employs a form of predicate abstraction on the resulting information from symbolic execution. They do not model real time, only logical program behavior.

Sammapun et al. [15] proposed a runtime verification technique, which consists of Monitoring and Checking (MaC). Their high-level approach includes running and monitoring the target TinyOS application in the Avrora simulator. Then, they check the monitoring data against formal program specifications. This technique is able to check timing and dynamic properties, and it acts as a systematic approach to finding bugs.
VI. Conclusion

In this paper we presented a methodology using the UPPAAL real-time model checker for verifying desirable properties of software for wireless sensors. The methodology combines reusable model components of TinyOS and nesC semantics, and then uses external actions to generate both the structural and temporal components of the application model. To generate the temporal component we use the Avrora simulator to extract timing information for application segments.

Our work has several possible directions in the future. The first is to fully implement the time extraction and model generation components of the methodology. Having worked out and established the efficacy of the methodology, building the components will now be worthwhile. We see no real technological difficulties in their construction; both are a straightforward application of (probably lightweight) parsing and regular template-based output.

Another direction is to extend the work to some of the new semantics introduced in some of the latest TinyOS extensions, including threads. As discussed in the paper, we do take a view of WSN applications that most, but not all, satisfy: that of input-independent constant time sections of code. Extending this to allow more time-varying behavior could allow verification of more complex WSN applications, although the challenge of controlling the model state size will be an issue.

Property verification necessarily focuses attention on the “worst-case” possibilities; if a property fails (especially a temporal property) it is probably under a scenario that is expected to be atypical. The real question then is how likely is that atypical scenario. Using counter-examples and some likelihood estimates and distributions, we may be able to estimate a probability for such an occurring scenario.

Overall, this methodology offers a new way to ensure that WSN applications will perform reliably under the variety of conditions that can occur in the field but that are hard to produce in a lab environment.

References


