# Runtime Verification for Wireless Sensor Network Applications

Usa Sammapun[†], John Regehr[*], Insup Lee[†], Oleg Sokolsky[†]
[†]Department of Computer and Information Science
University of Pennsylvania
[*]Department of Computer Science
University of Utah

## Abstract

*Wireless sensor networks are widely used to detect environment information that is not accessible by human. Developing such networks however requires low-level programming. The lack of sophisticated debugging tools for sensor networks makes it difficult to make the connection between a high-level functional or performance requirement and low-level implementation. This paper investigates a high-level approach by examining low-level execution data from a simulator and checking the data against specification written in formal logics. Such a technique raise the development level for wireless sensor network and providing a mechanism for understanding high-level behaviors of the system in terms of low-level observation. This study can become a road map for wireless sensor network application development.*

## 1 Introduction

A wireless sensor network usually comprises of a collection of tiny devices with built-in processors that can gather physical and environment information such as temperature, light, sound, etc., and communicate with one another over radio. Wireless sensor network applications sit on top of an operating system called TinyOS [2] and are mostly written in nesC [4], an extension of C that provides a component-based programming paradigm. Most of wireless sensor network applications are developed and tested on a simulator before they are deployed in the environment because testing and debugging directly on physical devices are very difficult, especially when the network consists of many nodes, and may not provide enough information for debugging. A simulator usually produces detailed execution information and can help find errors. However, even with the simulator and nesC, the current state of development tools for wireless sensor network still requires very low-level programming, which makes it hard for the developers to maintain a high-level view of the system operation. During the validation stage, lack of sophisticated debugging tools for sensor networks makes it difficult to make the connection between a high-level functional or performance requirement and a particular aspect of system implementation.

This paper investigates a high-level approach to examine execution data from a simulator and analyze it using formal methods. The technique 1) identifies and formally specifies high-level requirements for the system under development, 2) monitors a distributed wireless sensor network application using data provided by the simulator, and 3) checks for timing and dynamic properties to gain understanding of the relevant behaviors of wireless sensor nodes and to provide a systematic approach in finding bugs and errors. This paper uses a monitoring and checking technique, called *runtime verification* [3, 5, 1], which observes program executions at runtime and checks whether the behavior complies with a given formal specification. A particular runtime verification used in this paper is MaC or Monitoring and Checking [5, 7, 8]. MaC provides specification languages capable of expressing functional, timing, and probabilistic properties to specify requirements or patterns of errors. Properties can be examining periodic behaviors or identifying a faulty node. MaC then monitors and checks a wireless sensor network application against its specification by observing data produced by a simulator. In this paper, a simulator called Avrora [9] is used.

Contributions for applying the monitoring and checking technique to check wireless sensor network applications are threefold: 1) raising the development level for wireless sensor network, 2) providing a mechanism for understanding high-level behaviors of the system in terms of low-level observation, and 3) providing a tool based on the acceptance of the state of the art development tool for sensor networks. The result of this paper can provide a road map for wireless sensor network development.

Dagstuhl Seminar Proceedings 07011
Runtime Verification
http://drops.dagstuhl.de/opus/volltexte/2008/1371

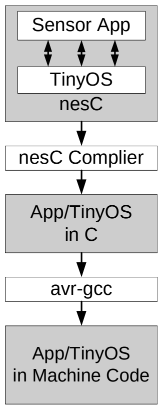


Figure 1: Wireless sensor application and TinyOS

## 2 TinyOS and Avrora

This section provides some background on TinyOS and Avrora.

### 2.1 TinyOS

A TinyOS [2] is a component-based operating system with a simple event-based concurrency model. Its modest power load and small size make it suitable for the tiny sensor devices. TinyOS has a small scheduler and provides many reusable system components such as timers, LEDs, and sensors which can be excluded when not in used. These components are either software modules or thin wrappers around hardware components. A component is described in terms of implementation and the interconnection between its lower-level subcomponents. A component consists of *commands*, *tasks*, and *events*. Commands are synchronous non-blocking requests made to lower-level components and usually return immediately. Tasks in TinyOS are computational units that run to completion without preempting any computation but may be postponed. Thus, tasks are more suitable for non time-critical computation. Events are asynchronous computational units that run to completion but may preempt other events or tasks. Events are for time-critical computations such as low-level communication.

Figure 1 shows the relationship between TinyOS and TinyOS wireless sensor applications. TinyOS itself and TinyOS applications are written in nesC [4]. Before applications can be run on hardware, TinyOS itself and applications are compiled into C programs by a nesC compiler and then compiled again into specific hardware instructions by a compiler `avr-gcc`.

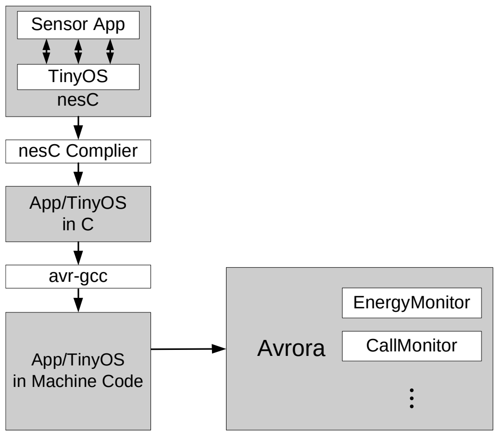


Figure 2: Wireless sensor application, TinyOS, and Avrora

### 2.2 Avrora

Avrora [9], written in Java, is a simulator for wireless sensor network applications. Figure 2 illustrates Avrora. Avrora takes as an input a program in machine code such as Berkely motes or Mica2Dot and then simulates the program. Each node is represented by one Java thread. Each instruction is represented by a Java object with its operands stored in the object. The instruction object also has several utility methods such as `getCycles`() and `getSize`(), which returns corresponding information about the instruction. Avrora also provides state information such as the values of different registers, memory, and program counter. Avrora executes instruction via an interpreter specific each hardware such as Berkely motes or Mica2Dot.

Avrora also provides several `Monitor` modules for probing applications such as an `EnergyMonitor` module that monitors energy levels of an application and a `CallMonitor` that monitors any TinyOS events, tasks and commands, shown in Figure 2. A custom `Monitor` can also be added. Avrora provides methods such as `insertProbe`() and `removeProbe`() to insert or remove a probe at a particular point in a program. The probes also have methods `fireBeforeCall`() and `fireAfterCall`() to be executed before or after a particular instruction. Memory can also be examined through watchpoints using methods `insertWatchpoint`(), `fireBeforeRead`(), `fireBeforeWrite`(), `fireAfterRead`(), and `fireAfterWrite`(). The progress of time in Avrora is driven by the execution cycles of instrunctions. Avrora represents multi-nodes using multi-thread instances of the `Simulator` class. Each node runs its own code with

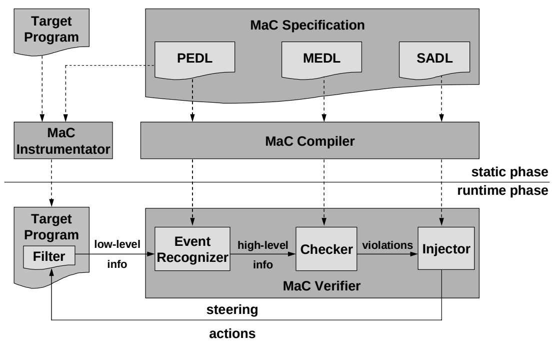


Figure 3: Overview of the MaC framework

its own view of environment and local simulation time. They are synchronized periodically to a global clock.

## 3 Monitoring and Checking (MaC)

Monitoring and Checking or MaC [5, 7, 8] can provide a sophisticated debugging tool for wireless sensor networks. MaC, illustated in Figure 3, ensure that a program is executing correctly with respect to its formal specification. It has two phases: static phase and runtime phase. During static phase, formal requirements are used to generate runtime components, and a program is instrumented with probes, called *filter*, to extract low-level information. During runtime phase, a program execution is monitored and checked with respect to the MaC specification. An *event recognizer* detects low-level information specific to program implementation from the *filter*, transforms the low-level information into high-level information, and forwards it to a *checker*. The *checker* then determines whether the high-level information satisfy the high-level requirement. If the checker detects any violations, an alarm is raised and may be handled by an *injector*, which attempts to steer the program back to its safe states.

**Specification Language.** The main aspect of MaC is to specify formal requirements. MaC provides three specification languages. The low-level monitoring specification or Primitive Event Definition Language (PEDL), defines which low-level application-dependent information is extracted, and how the information is transformed into high-level information. The high-level requirement specification or Meta-Event Definition Language (MEDL), based on Linear Temporal Logic (LTL) [6] and regular expressions, allows one to specify safety properties based on the high-level information. PEDL is tied to a particular implementation while MEDL is independent of any implementation. The steering specification or Steering Action Definition Language (SADL) specifies how violations of safety properties should be handled. Only MEDL is presented in this paper. See MaC [5] for PEDL and SADL.

$$\begin{array}{lcl} E & ::= & e \mid (E) \mid E||E \mid E\&\&E \mid start(C) \\ & \mid & end(C) \mid E\ when\ C \mid E+d \\ & \mid & E\ pr(<p_0, E) \mid E\ pr(>p_0, E) \\ C & ::= & c \mid (C) \mid !C \mid C||C \mid C\&\&C \mid C \rightarrow C \\ & \mid & [E, E) \mid [E, E)_{<d} \end{array}$$

Figure 4: Syntax of events and conditions

The high-level information can be distinguished into two forms: events and conditions. Events occur instantaneously during execution, whereas conditions represent system states that hold for a duration of time and can be $true$ or $false$. For example, an event denoting a call to a method $init$ occurs at the instant the control is passed to the method, while a condition $v < 5$ holds as long as the value $v$ does not exceed $5$. The syntax of events and conditions are given in Figure 4. (Please note the overloaded term, *events*, used in different contexts in MaC and TinyOS. Here, events in TinyOS are called *TinyOS events* and events in MaC are called *MaC events* to avoid confusion. When the context is clear, only *events* will be used.)

The models for MEDL are sequences of worlds, similar to those used for LTL. Each world represents one time instant containing event and condition values. A world is changed from one to aother when the event recognizer forwards events and conditions to the checker. Events are present only at an instant in a specific world whereas conditions retain their values between worlds. The semantics of events and conditions are given as follows. $e$ and $c$ are an event and a condition forwarded from the event recognizer. Disjunction $E_1||E_2$ and conjunction $E_1\&\&E_2$ are defined normally. $start(C)$ is an instant when a condition $C$ becomes $true$, and similarly, $end(C)$ is an instant when a condition $C$ becomes $false$. An event $(E\ when\ C)$ is present if $E$ occurs at a time when a condition $C$ is $true$. A timing event $E + d$ occurs $d$ time units after an event $E$ occurs. $d$ is a non-negative constant and is counted starting from the most recent occurrence of $e$. Probabilistic events $E\ pr(<p_0, E_0)$ and $E\ pr(>p_0, E_0)$ are present when an event $E$ occurs with probabability $< p_0$ or $> p_0$, respectively, given that an event $E_0$ has occurred.

For conditions, negation $(!C)$, disjunction $(C_1||C_2)$, conjunction $(C_1\&\&C_2)$, and implication $(C_1 \rightarrow C_2)$ are also interpreted classically. Any pair of events define an interval forming a condition $[E_1, E_2)$ that is $true$ from an event $E_1$ *until* an event $E_2$. The time-bound interval $[E_1, E_2)_{<d}$ holds $true$ from an event $E_1$ until an event

$E_2$ and from an event $E_1$ until $d$ time units after $E_1$. This time-bound interval is a generalization of $[E_1, E_2)$, which can be written as $[E_1, E_2)_{<\infty}$.

MEDL also allows variables to store histories of events, for instance, to count how many events have occurred. These variables are updated in respond to event occurrences. Events have two attributes associated with them. An attribute $time$ refers to a timestamp of the most recent occurrence of an event while an attribute $value$ is a tuple of application-dependent values. MEDL distinguishes special events and conditions that denote system specification. Safety properties are conditions that must *always* be true during an execution. Alarms, on the other hand, are events that must *never* be raised during an execution. From the viewpoint of expressiveness, both safety properties and alarms correspond to the safety properties [6].

Events and conditions are assigned to names and can be indexed by node identifiers. For example,

$$\texttt{event send[i] = func[i] when}$$
$$\texttt{value(func,0) == 'SurgeM\$sendData';}$$

An event `send` for a node `i` is triggered when a function `SurgeM$sendData` from node `i` is invoked. This function corrensponds to either a TinyOS event, command, or task. An event `func` carries its name as its first value, which can be retrieved from an attribute `value` using an index 0.

## 4 Integrating Avrora and MaC

While an application execution can be examined via Avrora, its detailed information may be too low level making it difficult to make a connection between the gap of low-level implementation and high-level specification, and consequently, errors are hard to locate. MaC closes the gap by checking low-level implementation against high-level specification written in formal logic. This section describes how MaC can be integrated into Avrora to monitor and check wireless sensor network applications.

**Connecting Avrora and MaC.** Recall that Avrora allows a developer to write a custom `Monitor` module to probe wireless sensor applications. A new custom `Monitor` module, called `Logger`, is added to Avrora to log TinyOS events, tasks, and commands. `Logger`, adapted from examples available on Avrora website [9], calls methods `fireBeforeCall()` and `fireAfterCall()` to log all calls to execute TinyOS events, tasks, and commands and also uses memory watchpoints to monitor BASE address, for instance, to log function parameters. to log function parameters. MaC reads these logs of function calls and memory via TCP/IP sockets.

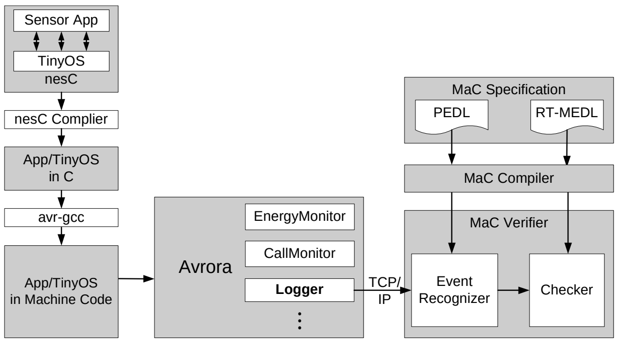


Figure 5: Integrating Avrora and MaC

| Optimize | Flags | Code size (Byte) | Run on Avrora | Function Detected |
|---|---|---|---|---|
| None | `-o0` | 828471 | No | Yes |
| Low | `-o1` | 424450 | No | Yes |
| Balanced | `-o1/o2` | 408597 | Yes | Yes |
| High | `-o2` | 401750 | Yes | No |
| For size | `-os` | 390740 | Yes | No |

Table 1: How optimization affects code size

The integration of Avrora and MaC is shown in Figure 5. Here, the new monitor, `Logger` is added to Avrora. `Logger` sends information about wireless sensor applications and TinyOS to MaC via TCP/IP. An event recognizer in MaC receives the information and forwards it to the checker which checks the application against its specification written in MEDL. Since the feedback is not used in here, the injector and SADL are not present.

**Low-Level Information Extraction.** For MaC to check wireless sensor applications, the monitor `Logger` must at least be able to detect TinyOS events, tasks, and commands. These TinyOS events, tasks, and commands, originally written in nesC, are actually C functions after their nesC implementation is compiled into C. A compiler `avr-gcc` compiles these C functions into labels and jump instructions in machine code. Avrora simulates the machine code and detects TinyOS events, tasks, and commands by looking for appropriate jump instructions. These functions' invocation is a good low-level monitoring point for checking properties using MaC.

Detecting the function invocation can be difficult because of optimization in compilers. When a nesC program is compiled into C, `inline` keywords are inserted into some functions. The `inline` keyword tells `avr-gcc` to embed the function code into its caller. The function thereby no longer exists and cannot be detected by Avrora. If inline functions are essential for checking properties, these `inline` keywords need to be manually removed from the C program. without the `inline` keywords, `avr-gcc` still inlines some functions when some optimization flags are present. Optimization are however necessary to keep TinyOS applications small enough to run on Avrora. Therefore, the right combination of optimization is needed to keep the functions not inlined and still keep the application size small.

Table 1 shows how optimization affects machine code size and inlined functions. When optimization is low, code size is large freezing Avrora but functions are not inlined and can be detected. When optimization is low, code size is small but functions are inlined and cannot be detected. One exception is when the flag is `-o3`, which tries to inline all "simple" functions and may produce inverse effect that increases code size rather than decreases. To get the right combination of optimization, the `-o1` flag is used with some of `-o2` optimization.

```
-o1 -fforce-mem -foptimize-sibling-calls
-fstrength-reduce -fcse-follow-jumps
-frerun-loop-opt -fgcse -fgcse-lm-fgcse-sm
-fdelete-null-pointer-checks -fpeephole2
-fexpensive-optimizations-fregmove
-freorder-functions -fstrict-aliasing
```

With these combination of optimization, the size of the machine is small enough to run on Avrora and the functions are also logged appropriately.

## 5 Checking TinyOS Applications

After the connection between Avrora and MaC is established, TinyOS applications can be checked by MaC via Avrora. This section presents a wireless sensor network application called Surge, describes how to specify its properties in MEDL, and provides results.

### 5.1 Surge

Surge [4] is an application that periodically samples a sensor to obtain environment information such as light or temperature and reports its readings to a base station. Before the sampling is done, multi-hop routing is discoverd in terms of spanning tree where the base station is the root. Each node discovers the route by sending messages to its neighbor and then establishes an appropriate node as its *parent*. Each node also maintains an address of this parent and a depth of the spanning tree. The parent and depth information may be updated periodically afterward. After the route is discovered, each node samples the environment and sends data to its parent, which then forwards to its parent until the data arrives at the base station.

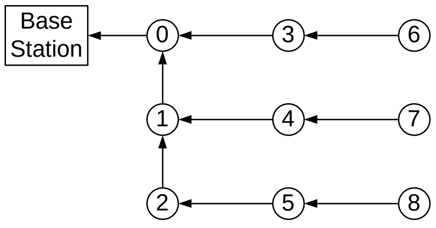

$$\begin{tikzcd}
\boxed{\text{Base Station}} & 0 \arrow[l] & 3 \arrow[l] & 6 \arrow[l] \\
 & 1 \arrow[u] & 4 \arrow[l] & 7 \arrow[l] \\
 & 2 \arrow[u] & 5 \arrow[l] & 8 \arrow[l]
\end{tikzcd}$$

Figure 6: Surge topology used in this case study

When the Surge application is run on Avrora, the topology of nodes must be supplied to Avrora. Figure 6 presents the topology used in this case study. There are nine nodes in total. All the data collected at each node must be sent to the base station by sending and forwarding via its parent. Here, Node 0 is connected a base station via UART port while other nodes send and forward message via radio. Node 3 sends data to the base station via Node 0. Node 6 sends data to the base station via Node 3, which then forwards the data of Node 6 to Node 0. Other nodes send and forward message similarly according to their topology.

Surge consists of different components: system boot code (`Main`), a timer (`Timer`), multi-hop message routing (`Multihop`) a sensor (`Photo`), and LEDs (`Leds`). These components are provided by TinyOS. Surge can just wire these component appropriately and implements necessary operations such as a task `SendData` that reads a sensor and sends data. Although Surge has only a few tasks or events, it generates many other tasks and events via its TinyOS subcomponents. These tasks and events are logged and sent to MaC to be checked against Surge's specification, described in the next subsection.

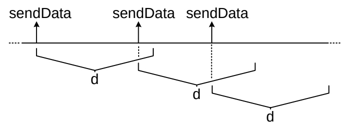
### 5.2 Specifying High-Level Properties

This subsection identifies possible properties for the Surge application. Since Surge nodes should send data periodically, possible properties are 1) examining periodic behaviors, 2) identifying a faulty node, and 3) analyzing send and forward behaviors.

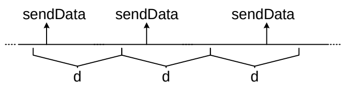
**Examining periodic behaviors.** Each node in Surge periodically reads and sends environment data to a base station. We may want to identify its period. Figure 7 shows MEDL properties for finding Surge's period. A variable `prevSend` stores the timestamp of previous send

```
 1 var int prevSend[i] = 0;
 2 var int avg = 0;
 3
 4 event send[i] = func[i] when
 5   value(func, 0) == 'SurgeM$sendData';
 6
 7 send -> {
 8   avg = (avg == 0) ? time(send) :
 9    (avg + (time(send) − prevSend[send.i]))/2;
10   prevSend[send.i] = time(send);
11 }
```

Figure 7: Properties for identifying its period



(a) Relative



(b) Absolute

Figure 8: Periodic behaviors

for each node i while avg keeps the current average of a period (lines 1-2). An event send, described earlier in Section 3, specifies an instant when a function SurgeM$sendData from node i is invoked where a MaC event func is triggered when any tasks, commands, or TinyOS events are called (lines 4-5). Whenever a send occurs, avg and prevSend are updated accordingly (lines 7-11). After running the Surge application with 200 sends, the average period in Avrora simulation time unit is 14750917.

Periodic behaviors can be thought of as being either relative or absolute. For relative periodic behaviors shown in Figure 8(a), a period starts right after a send occurs, and the next send must occur within $d$ time unit after a current send occurs. On the other hand, a period for absolute periodic behaviors shown in Figure 8(b) starts at specific absolute time. The next send does not have to occur within $d$ time unit after a current send occurs as long as the next send occurs within the next period frame. Both periodic behaviors can be specified in RT-MEDL. An excerpt below shows MEDL for relative periodic behaviors.

```
alarm missRel[i] = end(
  [send[i], send[i] + 14700000) );
```

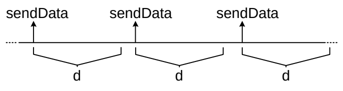
An alarm missRel for node i is defined by an end of a condition [send[i], send[i]+ 14700000). The condition becomes true when a MaC event send occurs and stays true as long as send occurs periodically updating the time of the most recent occurrence of send and preventing the MaC event send[i] + 14700000 from occurring. When send does not occur within 14700000 time units after its last occurrence, the condition [send[i], send[i] + 14700000) becomes false and triggers an alarm missRel, which indicates that a corresponding mote fails to send current periodic data.

Specifying absolute periodic behaviors in MEDL is also possible albeit more complicated as shown in Figure 9. Lines 1-4 declare four variables: 1) period keeps an absolute period, 2) prevTime stores timestamps of the previous send for each mote, 3) nextPeriod specifies the absolute time where the send for each mote must occur before, and 4) countPeriod keeps counts of how many time an end of a period has occurred without any send.

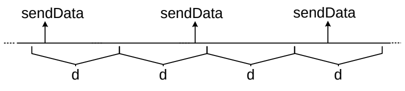
```
 1  var int period = 14700000;
 2  var int prevTime[i] = 0;
 3  var int nextPeriod[i] = 14700000;
 4  var int countPeriod[i] = 0;
 5
 6  event endPeriod = func when
 7     (prevTime[func.i] < nextPeriod[func.i] &&
 8     nextPeriod[func.i]) <= time(func);
 9  alarm missAbs[i] = start(countPeriod[i] >= 2);
10
11  endPeriod -> {
12    nextPeriod[func.i] = nextPeriod[func.i] + period;
13    countPeriod[func.i] = countPeriod[func.i] + 1;
14  }
15  send -> {
16    countPeriod[func.i] = 0;
17  }
```

Figure 9: MEDL properties for checking absolute periodic behaviors

An MaC event endPeriod, specified in lines 6-8, indicates an end of an absolute period, which occurs when current absolute period is between the timestamp of a previous call of a TinyOS event, tasks, or command and the timestamp of the current call. Because MaC cannot trigger events at a certain Avrora's absolute time, this specification of the MaC event works sufficiently to detect an end of a period. When an end of period is reached, the variable nextPeriod for each mote is updated to a new absolute period by adding a period to its current value and the variable countPeriod is also incremented. (lines 11-14). Line 15-17 resets the variable countPeriod to zero. An alarm missAbs raised when the variable countPeriod is



(a) Relative



(b) Absolute

Figure 10: Periodic behaviors when periods are too small

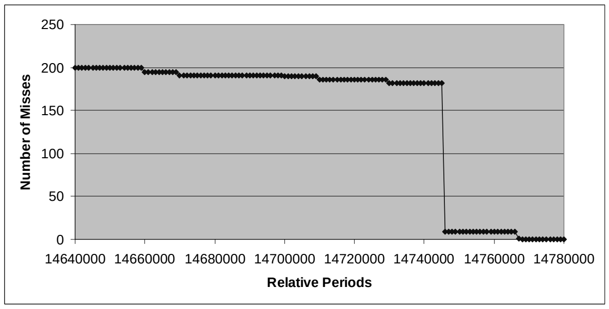


Figure 11: Number of nodes that did not send data within its relative periods

equal or greater than 2, which indicates that a corresponding mote has failed to send data within its absolute period.

To examine Surge periodic behaviors, Surge is run on Avrora and checked against periodic behavior properties by MaC. Different values for both relative and absolute periods are collected to see the affect of different periods to the number of sends that fail to occur within corresponding periods. We call a send that does not occur within corresponding periods as a "*miss*". For relative periods, when the period is too small, there are many misses because the period ends before another send occurs shown in Figure 10(a). Thus, when there are 200 sends for instance, the number of misses can go up to the maximum of 200. When the absolute periods are too small causing many misses, send would occur within some periods but not others shown in Figure 10(b). In this case, if there are 200 sends, the number of misses would never get to the maximum of 200 because a send must occur at least in one period.

Figure 11 presents experimental data for different values of relative periods to see how these different periods affect the number of sends that do not occur within corresponding relative periods. In this experiment, the total number of sends is 200. Figure 12 shows similar data for absolute periods. Please notice the different scales on the two figures. The data for relative periods illustrates a sudden drop of numbers of misses from a period of 14745000 to a period of 14746000 and becomes zero at 14768000. The sudden drop infers that the time duration between two sends are very precise between 14745000 and 14746000.

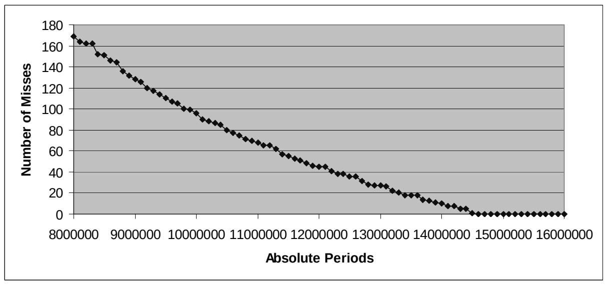


Figure 12: Number of nodes that did not send data within its absolute periods

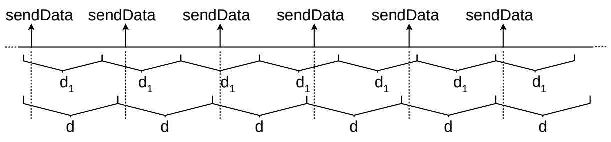


Figure 13: Slightly small absolute periods may not produce misses for small number of trials

The data for absolute periods shows a linear decrease in the number of misses. It becomes 0 at a period of 14600000. This infers that as the period increases and gets close to 14600000, the period becomes large enough to include at least one send in its period and when it reaches zero, the period is very close to its actual period. Since the data from relative periods clearly shows that the time duration between two sends are very precise between 14745000 and 14746000, 14600000 seems too small to be an absolute period. It is possible that even when the number of misses is zero, the specified period might still be too small but just so slightly that 200 sends cannot catch it. Figure 13 illustrates such an example. Here, $d$ is the actual period and $d_1$ is the slightly smaller specified period. With only five sends, the smaller period $d_1$ produces 0 miss. With six sends, however, there is a miss. In our data, it is possible that the difference between 14600000 and the actual period is so small that 200 sends cannot detect a miss.

Nonetheless, because of the sudden drop in the relative data, it clearly shows that the periodic behaviors are relative. This is not surprising however since the nodes are run on a simulator that most likely simulates the perfect environment.

| Node | True $p$ | # Misses | $\bar{p}$ | $z$ | Alarm |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 0.0 | -2.981 | No |
| **1** | **0.05** | **5** | **0.0625** | **-1.118** | **No** |
| 2 | 1 | 0 | 0.0 | -2.981 | No |
| 3 | 1 | 0 | 0.0 | -2.981 | No |
| **4** | **0.3** | **19** | **0.2375** | **4.099** | **Yes** |
| 5 | 1 | 0 | 0.0 | -2.981 | No |
| 6 | 1 | 0 | 0.0 | -2.981 | No |
| 7 | 1 | 0 | 0.0 | -2.981 | No |
| **8** | **0.2** | **13** | **0.1625** | **1.863** | **Yes** |

Table 2: Probabilistic send in Node 4 and Node 8

**Identifying a faulty node.** After a period is identified, it can be used to pinpoint a faulty node that may stop sending data a few times. An excerpt of MEDL below specifies how to identify a faulty node using probabilistic properties.

```
alarm faultyNode[i] =
  missRel[i] prob(> 0.3, send[i]) ||
  end( [send[i], send[i] + 50000000) );
```

Here, each node is monitored to see if the periodic send has stopped for sometime or delayed for too long. An alarm `faultyNode` of a node `i` is triggered when a faulty node `i` is found. `faultyNode` is triggered when a MaC event `missRel` occurs with probability more than 0.3 with respect to a MaC event `send` or the node has not sent data for more than 50000000 time units after its last occurrence.

This faulty node error however is a physical error rather than a software error. Because the environment simulated by Avrora is perfect, it is impossible to detect this error on Avrora unless an artificial bug is introduced into Avrora to simulate the unpredictable environment of sensor nodes. In this case study, using the java class `Random`, the artificial bug using is introduced into Node 1, Node 4, and Node 8, shown in Figure 6. Node 1 would produce a send that fails to execute within its period with only a probability 0.05 while Node 4 and Node 8 would fail with higher probabilities of 0.3 and 0.2, respectively. Table 2 shows the result of probabilistic properties. The sends by other nodes are still simulated perfectly by Avrora. When the node fails to send data in time with probability greater than 0.1, MaC raises an alarm appropriately. Otherwise, MaC does not raise an alarm.

**Analyzing send and forward behaviors.** Change the topology to linear, increase the number of nodes to see if the forwarding behaviors intefere with the send of environment data.

## 5.3 Limitation

Monitoring and checking wireless sensor network applications using MaC and Avrora are somewhat limited. Because it is done through Avrora which always simulates perfect environment, it can only detect software errors but not hardware errors. The distributed nature of TinyOS applications also posts a problem. MaC can check propoties for multiple nodes but still limited. In the actual world, data received from multiple nodes will not be complete nor in order, and thus only local properties but global can be checked. As our future work, MaC needs to be extended to be able to handle global properties when checking sensor nodes in their physical worlds.

## 6 Related Work

related work for programming support for wireless sensor network, mac. could be here or right before conclusion?

John says: I prefer right before conclusion

`http://www.cs.ucla.edu/~kohler/pubs/sympathy-emnets04.pdf`

`http://www.cs.virginia.edu/~control/docs/papers/infocom06-envirolog.pdf`

`http://www.vs.inf.ethz.ch/publ/papers/mringwal-monito-2005.pdf`

`http://www.eecs.harvard.edu/~mdw/papers/motelab-spots05.pdf`

## 7 Conclusion

Low-level programming for wireless sensor network applications can be difficult. Although Avrora already provides a tool for to examine a program, Avrora data may be too detailed and overwhelming making the process of finding errors difficult. Monitoring and checking using MaC can aggregate Avrora data and provide a higher overview of the application. The result of the monitoring and checking allows us to gain some understanding of relevant behaviors of wireless sensor devices and can narrow the gap between the high-level requirement and the implemention of an application. Future works include extensions to check sensor devices in their physical worlds to be able to detect physical errors since Avrora's perfect simulation of environment allows only software errors to be checked. MaC should also be extended to handle wider variety of distributed global property checking.

## References

[1] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of the 5th Conference on Verification, Model Checking and Abstract Interpretation*, Vanice, Italy, 2004.

[2] D. E. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo. A network-centric approach to embedded software for tiny devices. In *Proceedings of the ACM Conference on Embedded Systems Software (EMSOFT)*, Tahoe City, California, October 2001.

[3] D. Drusinsky. Monitoring temporal logic specifications combined with Time Series constraints. *Journal of Universal Computer Science*, 9(11), November 2003.

[4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[5] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a runtime assurance approach for Java programs. *Formal Methods in Systems Design*, 24(2):129–155, March 2004.

[6] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[7] U. Sammapun, I. Lee, and O. Sokolsky. RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties. In *Proceedings of the 11th IEEE Conference of Embedded and Real-Time Computing Systems and Applications*, 2005.

[8] O. Sokolsky, U. Sammapun, I. Lee, and J. Kim. Runtime checking of dynamic properties. In *Proceeding of the 5th Workshop on Runtime Verification (RV'05)*, Edinburgh, Scotland, UK, July 2005.

[9] B. L. Titzer. Avrora: The AVR simulation and analysis framework. Master's thesis, University of California, Los Angeles, June 2004.