2010 International Conference on Distributed Computing Systems

Sentomist: Unveiling Transient Sensor Network Bugs via Symptom Mining

Yangfan Zhou* Xinyu Chen* Michael R. Lyu* Jiangchuan Liu†
*Dept. of Computer Science & Engineering, The Chinese Univ. of Hong Kong, Shatin, Hong Kong
†School of Computing Science, Simon Fraser Univ., Burnaby, BC, Canada

***Abstract*—Wireless Sensor Network (WSN) applications are typically event-driven. While the source codes of these applications may look simple, they are executed with a complicated concurrency model, which frequently introduces software bugs, in particular, transient bugs. Such buggy logics may only be triggered by some occasionally interleaved events that bear implicit dependency, but can lead to fatal system failures. Unfortunately, these deeply-hidden bugs or even their symptoms can hardly be identified by state-of-the-art debugging tools, and manual identification from massive running traces can be prohibitively expensive.**

In this paper, we present `Sentomist` (`Sen`sor application `anatomist`), a novel tool for identifying potential transient bugs in WSN applications. The `Sentomist` design is based on a key observation that transient bugs make the behaviors of a WSN system deviate from the normal, and thus outliers (i.e., abnormal behaviors) are good indicators of potential bugs. `Sentomist` introduces the notion of event-handling interval to systematically anatomize the long-term execution history of an event-driven WSN system into groups of intervals. It then applies a customized outlier detection algorithm to quickly identify and rank abnormal intervals. This dramatically reduces the human efforts of inspection (otherwise, we have to manually check tremendous data samples, typically with brute-force inspection) and thus greatly speeds up debugging.

We have implemented `Sentomist` based on the concurrency model of TinyOS. We apply `Sentomist` to test a series of representative real-life WSN applications that contain transient bugs. These bugs, though caused by complicated interactions that can hardly be predicted during the programming stage, are successfully confined by `Sentomist`.

I. Introduction

Wireless Sensor Networks (WSNs) have been advocated as a promising tool for environmental data collection and monitoring [1]. Recent publications however report that existing WSN applications frequently encounter failures due to various software bugs [2, 3], posing a major barrier to their extensive deployments. In fact, potential industrial customers have ranked software reliability as the most critical concern toward adopting WSNs [4].

Given the limited computation and storage capacity of wireless sensors, existing WSN applications look short and simple [5]. For example, with TinyOS [6], we just need to customize less than 100 lines of codes to allow a sensor node to sense data and forward packets. Why do such simple codes still inject bugs? A key reason is that the simple codes are in fact executed with a complicated concurrency model. As an energy-aware embedded device, a sensor generally works in an *event-driven* mode. Specific event-handling logic (*i.e.*, *event procedure*) is activated by its corresponding event (*i.e.*, a hardware interrupt) [6, 7, 8]. For example, when receiving a packet, the wireless interface chip will issue an interrupt, activating its corresponding event procedure to perform such actions as retrieving the packet content. Internal procedures (*e.g.*, sampling sensor data regularly) are also activated by hardware timer interrupts. During system runtime, events may occur randomly, and instances of event procedures may therefore start at any time and even interleave with each other.

Such interleaved executions, together with the interactions among multiple sensor nodes and multi-tasking in the latest WSN operating systems (*e.g.*, TinyOS), can be too complicated to be comprehended even by the original system designers. Software bugs therefore become inevitable. They are also difficult to be identified by state-of-the-art software testing tools for commercial software [9, 10]. This is particularly true for *transient bugs*, *i.e.*, buggy logics that may only be triggered by some *occasionally* interleaved event procedures that bear implicit dependency [11]. Such bugs in WSNs can lead to fatal system failures. One famous example is a bug in the widely used Collection Tree Protocol (CTP) in the TinyOS distribution [12]. The bug, once triggered, makes a WSN stop data reporting. Unfortunately, due to their ephemeral nature, the symptoms of these transient bugs are deeply hidden. Even identifying the buggy symptoms becomes extremely labor intensive, not to mention correcting the bugs [11, 13].

In this paper, we present `Sentomist` (`Sen`sor application `anatomist`), a novel tool for identifying transient bugs in WSN applications. The `Sentomist` design is based on a key observation that transient bugs make the behaviors of a WSN system deviate from the normal, and thus outliers (*i.e.*, abnormal behaviors) are good indicators of potential bugs [14]. To this end, `Sentomist` introduces the notion of *event-handling interval* to systematically anatomize the long-term execution history of an event-driven WSN system into groups of intervals, during which the same event type is being handled. Such a semantic partition can exploit the similarity of system behaviors when the same event procedure runs. `Sentomist` then applies a customized outlier detection algorithm to quickly identify and rank abnormal intervals. As a result, WSN application developers and testers just need to inspect the suspicious short-time

1063-6927/10 $26.00 © 2010 IEEE
DOI 10.1109/ICDCS.2010.75

event-handling intervals, instead of the whole execution trace of a WSN system.

We have implemented `Sentomist` based on the concurrency model of TinyOS. The latest release can be found online [15]. We apply `Sentomist` to test a series of representative real-life WSN applications. Their bugs, though caused by complicated interactions that can hardly be predicted during the programming stage, are successfully located by `Sentomist`. In particular, we show three case studies that cover a wide range of interrupts in WSN applications, and `Sentomist` confides the bugs by automatically ranking the intervals containing their abnormal symptoms as the first several ones for manual inspection in all the three cases. This dramatically reduces the human efforts of inspection (otherwise, we have to manually check tremendous data samples, typically with brute-force inspection) and thus greatly speeds up debugging.

The rest of the paper is organized as follows. Section II presents the related work. In Section III, we introduce some preliminary knowledge on the concurrency model of WSN applications. Section IV illustrates a motivating example. In Section V, we elaborate how we mine the symptoms of bugs. Three case studies are discussed in Section VI. Section VII finally concludes this paper.

II. Related Work

As more and more experimental WSN systems have been field-deployed, software bug also starts its notorious role [2, 3]. Conventional software testing tools (*e.g.*, [16, 17]) are not adequate to test the interleaved executions of event procedures. They generally consider sequential programs, where functions call one another in a sequential manner. Existing concurrent program testing tools (*e.g.*, [18, 19]) do not work well for WSN applications, either, for the number of possible interleaved executions in WSN applications are generally so large that a complete coverage is impractical [20]. To enhance the reliability of WSN systems, many troubleshooting techniques, debugging tools, testing methodologies, and compile-time checking schemes for WSNs applications are proposed, which are surveyed in what follows.

Cooprider *et al.* [21] suggested adding data type and memory safety checks for applications running over TinyOS. NodeMD [22] extended compilers by inserting checking codes into WSN applications. These preventive tools have yet to be widely incorporated into WSN programming, and they are not able to eliminate all possible bugs.

Marionette [23] and Clairvoyant [24] are two recently-proposed debugging tools that provide interactive remote debugging interfaces for sensor nodes. Adding declarative tracepoints [25] has also been suggested for extracting program runtime information after observing abnormal behaviors. These tools facilitate fixing the *already-seen* bugs, but the identification of faulty behaviors depends on manual efforts inevitably.

Dustminer is the most recent troubleshooting approach for identifying bugs in WSN applications [11]. Based on a function-level logging engine [26], Dustminer checks discriminative log patterns, assuming that a bad behavior interval can be identified from a good one. It finds the differences between the logs of the good behavior interval and those of the bad one so that the root of the problem can be located. However, such identification of bad-behavior interval generally causes extensive manual efforts, especially when a bug is transient in nature.

Regehr [13] discussed how to make random interrupt test possible for WSN applications running over earlier versions of TinyOS. Though the technique does not necessarily work for the current TinyOS release, an important notion is that WSN applications are interrupt-driven and hence testing them needs to schedule random artificial interrupts. Based on this notion, Lai *et al.* [20] studied the test adequacy criteria for WSN applications. Unfortunately, in their framework, the size of test suites is large in nature and an automatic mechanism to verify the pass/fail of a test case is impossible. Moreover, a test case is a long sequence of interrupts which incurs a long runtime trace. All these make human inspection of the results impractical. `Sentomist` addresses this challenge by anatomizing the program runtime trace to a reasonable granularity and automatically picking up the most suspicious time intervals for manual inspection, which greatly reduces human efforts.

Finally, anomaly detection has long been proposed for test case selection and bug localization (*e.g.*, [27, 28]), shedding light on solving our problem. But existing approaches generally focus on unit testing for sequential programs [9] given a set of test cases. They are not applicable to our problem due to the complicated concurrency model of WSN applications and the lack of explicit test cases (instead, we just have a long-term trace of system behaviors). `Sentomist` closes this gap by anatomizing the trace into a set of samples based on the concurrency model and the semantics of WSN applications. Thus, anomaly detection can eventually be applied for locating bug symptoms.

III. Concurrency Model of WSN Applications

As energy is a critical resource for sensor nodes, WSN applications are generally event-driven so as to save energy: When no events are to be handled, the sensor node hardware can go into a power-conserving sleeping mode. An event in this design paradigm is actually an asynchronous hardware interrupt, *e.g.*, one that indicates a packet arrival or a timer timeout [6, 7, 8].

After an interrupt is triggered, the microcontroller unit (MCU) will automatically call its corresponding *interrupt handler*. It may be straightforward to implement the entire

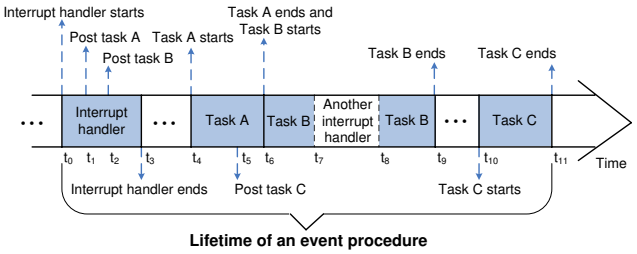


Figure 1. A sample event procedure instance.

```
...
 1: // This function is called by the handler for
 2: // ADC data-ready interrupt.
 3: event void Read.readDone(error_t error, uint16_t data)
 4: {
 5:     packet->data[dataItem] = data;
 6:     dataItem ++;
 7:     // After 3 data items have been collected, it
 8:     // posts a task, which will send out the items.
 9:     if(dataItem == 3)
10:     {
11:         dataItem == 0;
12:         post prepareAndSendPacket();
13:     }
14: }
...
```

Figure 2. A buggy function in an ADC event procedure, where `packet->data` will be polluted if the function is called again before the task `prepareAndSendPacket` runs.

event procedure (*i.e.*, the specific application logic for handling the event) in the interrupt handler. However, an event procedure may involve extensive computational efforts and thus need to be executed for a long time. Hence, to avoid the monopolization of MCU resource and to minimize the usage of function-call stack [29], WSN operating systems (*e.g.*, TinyOS) typically implement an event procedure as two separated parts: an asynchronous interrupt handler and some deferred synchronous procedure calls, namely, *tasks*.

A task is posted by an interrupt handler or another task. The operating system maintains one single *task queue*, where a task is posted and executed in a first-in-first-out (FIFO) manner. Moreover, tasks are executed only when there is no running interrupt handler. Finally, when there is a new interrupt arriving during the execution of a task, the task can be preempted by the interrupt handler. Otherwise, it will run till completion. We summarize three rules of the concurrency model of WSN applications as follows:

Rule 1: An interrupt handler is triggered only by its corresponding hardware interrupt;□

Rule 2: Interrupt handlers and tasks all run to completion unless preempted by other interrupt handlers;□

Rule 3: Tasks are posted by interrupt handlers or other tasks and executed in an FIFO manner.□

Figure 1 demonstrates an example of how a typical event procedure instance runs in a sensor node. Besides the interrupt handler, the instance of the event procedure contains three tasks: A, B, and C. The interrupt handler is activated at time t_0 by its corresponding hardware interrupt event. Before it exits at t_3, it posts two tasks A and B at t_1 and t_2, respectively. Later, after the MCU finishes performing the tasks posted previously by other event procedures, task A will be executed at t_4. It defers some application logic by posting another task C at t_5. After task A ends at t_6, task B runs, during which a new interrupt handler preempts it at t_7. After the preempting interrupt handler exits at t_8, task B continues till completion. Task C is then the last task to be executed. Hence, the event procedure instance starts at time t_0 and ends at time t_{11}, during which its logic is executed in the time intervals denoted by the shaded areas in the figure. Note that different event procedure instances for the same event type may not run in the same pattern. For example, task B in another instance may not be interrupted.

We formally define the *tasks* of an event procedure instance in a recursive way as follows:

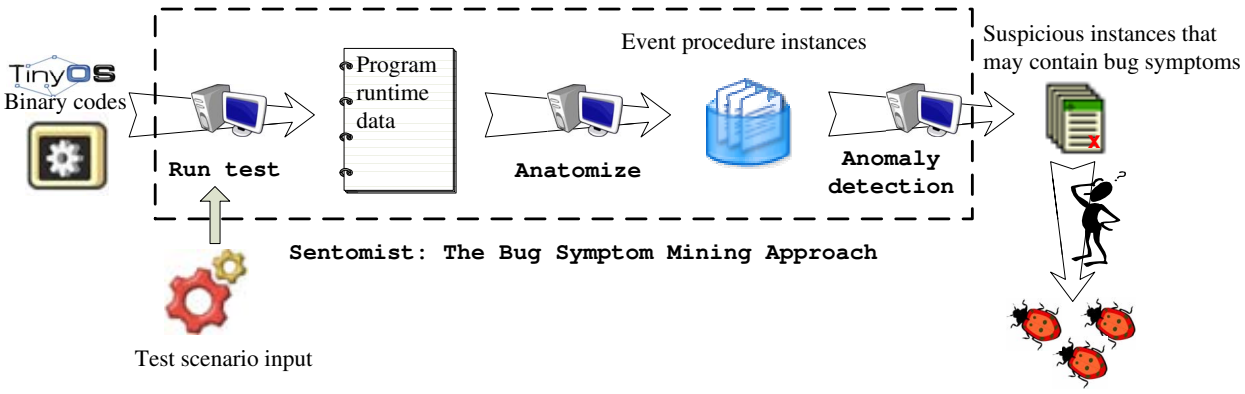
Definition 1: The tasks of an event procedure instance include those posted by its corresponding interrupt handler and those posted by other tasks of the same event procedure instance.□

IV. A Motivating Example: Data Pollution

The above concurrency model inevitably results in complicated interleaving executions of event procedures. Consequently, even simple codes may potentially contain transient bugs, which are particularly hard to be detected by current testing techniques. We demonstrate this by a simple example in Figure 2.

The nesC [30] code segment is adapted from the `Oscilloscope` application [6], a representative WSN application where a node senses and sends environmental data. Function `Read.readDone` in this figure is called by an ADC (Analog-to-Digital Converter) interrupt handler. The ADC interrupt is one issued by an ADC chip when the chip is ready to provide its data (*i.e.*, a sensor reading) [5]. The program intends to collect every three sensor readings and send them in one data packet. Sending a packet is a non-trivial procedure since the network communication protocols may spend some time preparing and sending the packet, and possibly resending the packet if it is lost. It is therefore implemented as a task, *i.e.*, a deferred procedure call to `prepareAndSendPacket()`, instead of being part of the interrupt handler.

This simple function contains a transient bug. Let us consider that the ADC interrupt has been issued for three times and the `data` collected so far are 100, 24, and 22, respectively. Then lines 11 and 12 are executed (see Figure 2). As a result, `dataItem` is now 0 and a task is posted and expected to send out "100, 24, 22" in a packet later. The behaviors of this event procedure may be normal most of the time. But under certain situations, *e.g.*, when the sensor



`Sentomist` takes the binary WSN application codes and the test scenarios as inputs and run the tests. A program runtime trace is then obtained. `Sentomist` anatomizes the program runtime into a set of time intervals, during each of which an event procedure instance runs. It then calls a plug-in outlier detection algorithm to locate suspicious instances which may contain bug symptoms. Finally, `Sentomist` generates a ranking of the instances, indicating the priority order by which we should perform a manual inspection to check the correctness of the WSN application.

Figure 3. Overview of our bug symptom mining framework.

node is performing another heavy-weighted event procedure, problems may occur. If there are many tasks queuing before this packet-sending task, it will take a relatively longer waiting time for the task to be executed. It is then possible that the fourth ADC interrupt arrives with new `data`, say 23, before the task runs. `packet->data[0]` is then modified to 23. Consequently, the task later will send out "23, 24, 22" instead. The original value 100 are polluted by the new reading 23.

The bug is a typical data-race bug caused by the interleaving of event procedures. It is however difficult to be unveiled. First, the bug is not easy to be triggered unless we generate a variety of random interleaving scenarios to hit the triggering condition. Second, even if the bug is triggered, the symptom of the bug is not obvious. The system will not crash. Also, the polluted data are not senseless, and thus a sanity check does not work. Consequently, it is hard to figure out the data pollution automatically. However, such correctness information of the test scenarios is generally required by either generic testing approaches [9] or specific testing/troubleshooting approaches for WSN applications [11, 20]. Hence, they do not work well in fighting against the transient bugs in WSN applications.

V. Automatic Bug Symptom Mining via Machine Learning

To cope with transient bugs, a brand new systematic method for testing WSN applications is desired. One important notion adopted in `Sentomist` is that the transient nature of such bugs is not always a curse to testing. Rather, this property can facilitate testing. Although tremendous testing scenarios are needed to trigger a bug, the application, however, behaves normally in most testing scenarios. This turns out to be a favorable characteristic: We can therefore summarize the normal behaviors, since they are dominant features. For example, in Figure 2, most execution patterns should be either "ADC interrupt, interrupt exit" or "ADC interrupt, posting a task, interrupt exit, running the task". In contrast, when a bug is triggered, the pattern would be something like "ADC interrupt, posting a task, interrupt exit, ADC interrupt, interrupt exit, running the task", which is obviously an outlier.

This observation is essentially the key for us to crack the challenging WSN testing problem. Through a proper decomposition of the program runtime, `Sentomist` can obtain a lot of time intervals. In most intervals the program would run in a similar manner, while in just a few intervals its behaviors are strange, where a bug might be manifested. With a set of quantified attributes assigned to each interval, these intervals can be abstracted as a set of vectors. `Sentomist` can then apply an outlier detection algorithm to find out which intervals are most suspicious of containing bug symptoms. The results can direct us to where we should conduct a thorough manual inspection. Such a bug symptom mining framework adopted in `Sentomist` is described in Figure 3.

We can see that three critical issues naturally arise in implementing `Sentomist`. The very first problem we need to address is how to decompose the program runtime into a set of time intervals, where the program behaviors of the majority of the intervals can exhibit certain statistical similarity. A natural granularity is the runtime of an event procedure instances. But given the complicated concurrency model, identifying the runtime of an event procedure instance is not trivial. Second, it is critical to select a set of good attributes so that each interval can be well featured. Finally, we need a generic outlier detection algorithm in processing these intervals. We illustrate how we attack these

three challenging problems in what follows.

A. Anatomizing program runtime

Let us first discuss how we get a set of time intervals from the entire program runtime, during each of which an event procedure instance runs. Although an event procedure instance always starts with an interrupt handler, when it terminates is not straightforward to identify. It is hard to tell how many tasks an event procedure instance may post, when a task is posted, and when it is executed. Moreover, the complicated interleaving executions of event procedure instances make it more difficult to track task posting and executions. We have to find a generic method to determine when an event procedure ends.

With the term *task* defined in Definition 1, let us first precisely describe *event-handling interval* as follows.

Definition 2: An event-handling interval is the lifetime of an event procedure instance. It starts at the entry of its corresponding interrupt handler. It ends when its last task has been executed if the interrupt handler posts tasks; otherwise, it ends when the interrupt handler exits.□

For the example shown in Figure 1, the corresponding event-handling interval is that from t_0 to t_{11}.

Although the WSN application concurrency model causes complicated interleaving executions of event procedures, we find that, by tracking a smaller number of function calls, it is enough to identify the concurrency of all event procedures, *i.e.*, each event-handling interval. Details are illustrated as follows.

In TinyOS, tasks are always posted via a postTask function and executed via a runTask function. The handler for each interrupt is also unique. During the system runtime, these two functions and the interrupt handlers will be called in sequence. We name such a sequence the system *lifecycle sequence*, which consists of four items: postTask, runTask, int(n), and reti. A postTask item and a runTask item indicate the calling of the postTask function and the runTask function, respectively. An int(n) item denotes the entry of the interrupt handler for interrupt number n, while a reti item indicates the exit of an interrupt handler. We will show how to identify each event procedure instance with only such a lifecycle sequence.

Based on $Rule$ 3 of the TinyOS concurrency model, we instantly have the following criterion since the tasks are scheduled and executed in a FIFO queue.

Criterion 1: The task posted via the ith postTask is executed via the ith runTask.□

Now we define an *int-reti string* as follows.

Definition 3: An int-reti string is a subsequence of the lifecycle sequence collected during the runtime of an interrupt handler. It starts with an int(n) item and end with a reti item which indicates the exit of the int(n) item's corresponding interrupt handler.□

```
// INPUT:  A lifecycle sequence and one of its int(n) items.
// OUTPUT: The index of the last item of the corresponding
//         event procedure instance.
 1: S ← The corresponding int-reti string of the int(n) item
 2: loc ← The index in the lifecycle sequence of the
 3:       last reti item in S
 4: Remove the substrings of S which are int-reti strings
 5: P ← S     // P contains only postTask items.
 6: loop
 7:   if P has no postTask items
 8:     output loc
 9:     break loop
10:   else
11:     T ← ∅
12:     for each postTask item p in P
13:       r ← p's corresponding runTask item
14:       loc ← r's index in the lifecycle sequence
15:       Q ← The string between r and the next runTask
16:       Remove substrings of Q which are int-reti strings
17:       // Q contains only postTask items.
18:       T ← TQ  // Concatenate T and Q.
19:     end for
20:     P ← T
21:   end if
22: end loop
```

Figure 4. Algorithm to identify an event procedure instance.

Also for the example in Figure 1, the items collected from t_0 to t_3 (*i.e.*, a int(n), followed by two postTasks and a reti) form an int-reti string.

According to $Rule$ 3, an int-reti string can contain an arbitrary number of postTask items. Because an interrupt handler may be preempted by other interrupt handlers according to $Rule$ 2, an int-reti string may also contain an arbitrary number of other int-reti strings. Furthermore, it must not contain runTask items since an interrupt handler cannot be preempted by a task. Hence, we get the following criterion.

Criterion 2: All items in an int-reti string, except those in its substrings which are also int-reti strings, must be postTask items. The tasks posted via the postTasks are those posted by the starting int(n) item's corresponding interrupt handler.□

Since tasks cannot be preempted by other tasks while can be preempted by other interrupt handlers, the following criterion is also true for the lifecycle sequence.

Criterion 3: All tasks posted via postTasks between two consecutive runTask items, except those in int-reti strings between these two runTask items, are posted by the task which is executed by the first runTask.□

With these three criteria, `Sentomist` can then parse the lifecycle sequence to determine when an event procedure instance starts and ends. An algorithmic description is shown in Figure 4, which in essential employs a breadth-first search. Since each event procedure for a particular event, *i.e.*, a hardware interrupt, starts with the interrupt handler, `Sentomist` will begin parsing each event procedure when its corresponding interrupt handler is called. First according to Criterion 2, a sequence of postTasks called by an interrupt handler is identified. Then with Criterion 1, we know which runTasks execute the tasks posted via the postTasks. Criterion 3 further tells us which postTasks are called by

these runTasks. Following Criterion 1 again, we know the runTasks that execute these tasks. Thus in a recursive way, we can work out which runTask executes the last task of the event procedure instance. When the runTask returns, it actually indicates the end of an event procedure instance.

Note that Sentomist requires to identify int-reti strings in the lifecycle sequence (Lines 1, 4, and 16 of the algorithm in Figure 4). We now discuss how this is done. Based on Criterion 2, with the formal-language conventional symbols [31], an int-reti string S can hence be described by a grammar G with the following formation rules:

$$\begin{aligned} S &\rightarrow \text{int(n)}\ R\ \text{reti} \\ R &\rightarrow P \mid P\ S\ R \\ P &\rightarrow \text{postTask}\ P \mid \varepsilon \end{aligned}$$

Here ε denotes an empty string. We can see that G is a context-free grammar which can be recognized by a pushdown automaton (*i.e.*, a recognizing algorithm) [31]. Moreover, for an int-reti string, none of its prefix substrings can belong to this grammar, since int(n) and reti are nested. Therefore, when reading an int(n) item from the lifecycle sequence, Sentomist can start feeding the item and the rest of the sequence to the recognizing algorithm until the algorithm reads a subsequence which is a valid string of the grammar. This string is the int-reti string we need.

In this way, Sentomist can obtain groups of event-handling intervals, during each of which an event procedure instance for handling the same event type runs. Finally, note that an event-handling interval may *overlap* with another one due to the interleaving executions of event procedure instances, since another instance may start before the current instance ends. This property is exactly what we need since we want to capture the program behaviors during such an overlap.

B. Featuring instances of event procedures

Given an event-handling interval when an event procedure instance runs, we should find a set of attributes to feature the behavior of the WSN application during this interval. We name the behavior of the WSN application during such an interval a *sample*. Our aim of such an abstraction is to quantify the samples so that they can be more tractable for automatic bug symptom mining. Hence, one instant requirement is that normal samples should still be distinguishable from abnormal samples where bugs are triggered via this abstraction. In other words, the abstraction must not eliminate the differences between normal samples and abnormal ones.

There are many straightforward candidates for featuring the samples. Examples include memory usage, number of calls to a specific function, sequence of function calls, and number of packets transferred. However, most of these attributes are only suitable for specific applications. For example, the number of packets transferred may only be applicable for testing certain network protocols. Moreover, some attributes are hard to be quantified although they may to some extent help distinguish normal samples from abnormal ones. The function call sequence of a sample is an example: It can help capture the situation that an event procedure instance is interrupted by another one since in that case the function call sequence will be changed. But it is hard to quantify such a complicated sequence, making it inappropriate for an automatic bug symptom mining algorithm. Based on these considerations, the attributes should be adequately chosen so that they are easy to quantify and suitable for generic WSN applications.

Sentomist adopts *instruction counter* as a metric to feature a sample, where the meaning of instruction is its conventional meaning, *i.e.*, a sentence of machine codes denoting a single operation of the MCU. The metric is formally described as follows.

Definition 4: An instruction counter of an interval during a WSN program runtime is a vector of N elements, where N is the total number of instructions of the program's corresponding machine codes. The ith element of the vector denotes the execution number of the ith instruction during the interval.□

Instruction counter is similar to source code coverage, a measure largely adopted in software testing for determining the testing completion condition by recording whether a line of source code has been executed [9]. One reason we adopt instruction counter is that such an abstraction scheme can be applicable for all WSN applications: Any event procedure instance can be mapped to a program runtime interval with the approach described in Section V-A, and can thus be abstracted as an instruction counter without any manual adaptation. Most importantly, this simple abstraction can accurately discriminate normal samples from abnormal ones since it can capture the symptoms of a transient bug, whose manifestation causes program instruction execution to deviate from normal patterns.

Let us again see the example in Figure 2. In the time interval during a normal ADC event procedure instance, since it does not interleave with another ADC event procedure instance, the instructions generated from lines 5, 6, and 9 would be executed once. In contrast, for the interval during which an ADC event procedure instance interleaves with another one, these instructions would be executed twice. The additional execution is contributed by another instance: Buggy interleaving executions can thus be captured.

Note that in this example, as the program sends a packet after every three readings, one third of the normal instances will execute the instructions generated from lines 11 and 12, while the rest will not. The differences between these two cases should not be considered as a bug symptom since the number of samples in both cases are large, which does not imply a transient bug. To this end, we need an

algorithm that can determine outliers with sophisticated statistical inferences so that the samples in neither case will be classified as abnormal samples.

C. Symptom mining: A one-class SVM approach

Given a set of instruction counters, the question now is how to find out the outliers where bugs are potentially triggered. Specifically, `Sentomist` needs to model the majority characteristic of the samples and quantify the difference between such a characteristic and that of a particular sample. The difference is instantly a metric to determine whether a sample is an outlier. To this end, `Sentomist` adopts a variant of the state-of-the-art Support Vector Machine (SVM) classification algorithm, called one-class SVM, to achieve this distinction.

1) One-class SVM: SVM is a statistics-based method that infers how *two* classes of points are different from each other [32]. The input of an SVM is a set of points in a d-dimensional space $\mathbb{R}$, where each point is designated to either one of the two classes. The algorithm finds a hyperplane that best separates these points into the two different classes. The hyperplane is considered as the boundary of the two classes. With such a boundary, any unlabeled sample can then be labeled to a class according to which side of the boundary it locates.

SVM has long been proven successful in many classification applications[1]. But it is a so-called supervised learning algorithm since we need to manually label each input sample to a class. In our problem settings, we do not have two sets of labeled samples. On the contrary, what we have is a set of unlabeled ones. To handle this problem, we should exploit the known statistics based on the ephemeral feature of transient bugs: Most samples are normal, while just a few are abnormal ones, if any. A trick is therefore to assume that all input samples belong to one class, *i.e.*, the *normal* class, which however contains some misclassified ones. Also consider that there is a virtual *outlier* class, which naturally contains the origin of $\mathbb{R}$ and some samples that are misclassified to the normal class[2]. We can then apply SVM to find a boundary to separate these two classes. Such a variant of SVM is called one-class SVM [33].

One-class SVM can thus model the majority characteristic of a set of unclassified samples and determine whether a sample is an outlier based on the decision boundary. Most input samples should be on one side (namely, the normal side) of the boundary. But note that such a boundary is not a *hard* classification boundary that can strictly discriminate normal samples from outlier samples since we allow misclassification errors. Consequently, we have the following heuristic: If a sample is on the normal side, the closer it is to the boundary, the more suspicious it is as an outlier. Otherwise, the farther it is away from the boundary, the more certain it is as an outlier. Considering that the distance between a sample to this boundary is positive if the sample is on the normal side, and negative otherwise, we can then take such a distance as a score to rank the samples. The lower the score of a sample is, the more possible that the sample should contain bug symptoms.

2) Why one-class SVM: There are many outlier detection algorithms [34] that can be employed in our framework. We decide to apply one-class SVM as it can well exploit the fact that the majority samples are normal and it can statistically infer a boundary that surrounds most of the samples without manual labeling. Consequently, it is a suitable technique for automatic outlier detection, which best matches our problem settings.

Moreover, the kernel method [32] can be seamlessly applied in one-class SVM. As such, it can find a nonlinear boundary that best surrounds the majority samples. This nonlinear property is critical to mine abnormal samples, since the discriminations between normal samples and abnormal ones in terms of instruction counter is nonlinear in nature.

Finally, one-class SVM can score all samples conveniently according to their distances to the boundary it finds. The ranking can instantly show how suspicious a sample is in a comparative way. We can thus select top k suspicious samples to perform careful manual inspection, where k can be flexibly chosen. This can accommodate different testing requirements based on how critical a WSN application is. In other words, k can be set according to the efforts we plan to put in manual inspections of the WSN application.

VI. Evaluation

To show the effectiveness of `Sentomist` in testing WSN applications, we discuss three representative case studies in this section, showing how we detected suspicious event procedure instances and unveiled subtle WSN bug symptoms. The three case studies are all based on real WSN applications distributed with TinyOS [6]. The symptoms of the bugs in these case studies are all difficult to unveil. The first bug is in a data collection application, which is caused by the interleaving of the event procedures triggered by internal events (those indicating the sensor readings are ready). The second bug resides in a multi-hop communication protocol triggered occasionally by the arrivals of packets from another sensor node (external events). The last is due to the coexistence of two communication protocols, when they race for the same hardware resource (wireless interface chip). The bugs in these case studies reside in the event procedures corresponding to different hardware interrupts, namely, ADC interrupt, Serial Peripheral Interface (SPI) interrupt, and timer interrupt, respectively. These cover various types of events that a typical WSN application needs to handle.

[1] A list of examples can be found in http://www.clopinet.com/isabelle/Projects/SVM/applist.html.

[2] We can deem that the origin of $\mathbb{R}$ corresponds to an instruction counter with all items being 0, which is trivially an outlier.

Instance Index	Score
$[1, 76]$	-1.5554
$[1, 176]$	-0.5291
$[1, 198]$	-0.2462
$[1, 239]$	-0.1541
$[1, 251]$	-0.0815
$[1, 9]$	-0.0313
$[1, 81]$	-0.0313
$\vdots$	$\vdots$
$[3, 12]$	0.9921
$[1, 153]$	1.0000

(a) Case study I

Instance Index	Score
20	-0.0827
108	-0.0827
157	-0.0519
21	-0.0519
58	0.0213
109	0.0825
158	0.1048
$\vdots$	$\vdots$
38	0.9862
86	1.0000

(b) Case study II

Instance Index	Score
$[8, 2]$	-0.0891
$[6, 22]$	-0.0881
$[8, 10]$	0.0205
$[8, 20]$	0.0205
$[8, 15]$	0.0232
$[8, 1]$	0.0239
$[6, 5]$	0.0239
$\vdots$	$\vdots$
$[2, 3]$	1.0000
$[2, 4]$	1.0000

(c) Case study III

Figure 5. Ranking results for the three case studies.

A. Data acquisition

`Sentomist` conducts WSN system testing over `Avrora`[3], a state-of-the-art emulator for real WSN applications [35]. `Avrora` can run a binary WSN application in the instruction level, which provides a cycle-accurate emulation of the sensor node hardware functionalities and their interactions. It can thus achieve nice fidelity for emulating real WSN applications. This property exactly meets our requirements since we aim at the transient bugs caused by interleaving executions of event procedures, where timing accuracy is of a critical concern.

The front-end data acquisition approach of `Sentomist` is a module we implemented to extend `Avrora`, which contains around $3,000$ lines of Java codes. It can be loaded as a *monitor* [35] of `Avrora`. `Sentomist` obtains the program runtime information during each testing run based on the built-in support for profiling and instrumentation in `Avrora`, anatomizes it into a set of event-handling intervals through the scheme described in Section V-A, and samples them to a set of instruction counters as described in Section V-B. After the testing runs stop, the back-end outlier detection approach of `Sentomist` will input the instruction counters to an outlier detection plug-in algorithm named `LIBSVM`, a well-adopted one-class SVM implementation[4], to rank the samples. The ascending ranking then indicates the priority order of the runtime intervals by which we should perform a manual inspection to check the correctness of the application. The source codes of `Sentomist`, including the front-end data acquisition approach and the back-end outlier detection algorithm, and all the case studies are available online [15].

B. Case study I: Data pollution in a single-hop data collection WSN

We first provide our experience on testing a WSN data collection application where several sensor nodes monitor temperature and report the readings to a data sink in a single hop manner, which includes the example codes described in Section IV. The program is adapted from `Oscilloscope` [6], where each sensor node requests its sensor readings periodically with a hardware timer whose timeout latency D is an application-specific parameter. This means the sensor nodes will collect the temperature every D seconds. When a reading is ready upon request, an ADC interrupt will be issued so that the program can get the reading via the ADC event procedure. After collecting three sensor readings, a node will post a task to send the three readings in a data packet to the sink. The core part of the ADC event procedure is shown in Figure 2.

Note that when testing a WSN application, the parameters of the application (*e.g.*, D in this case study) should be set according to the application specification. Because the environmental data sampling frequency is not large in general, in this study, we set D to be $20ms$, $40ms$, $60ms$, $80ms$ and $100ms$, respectively, in five testing runs indexed from 1 to 5, each of which lasts for $10s$.

With the front-end data acquisition approach of `Sentomist`, the program runtime is anatomized into a set of event-handling intervals, each corresponding to an ADC event procedure instance. In total, we have collected 1099 samples, giving us 1099 instruction counters. Each is indexed by $[r, s]$ where r is its testing run index and s is its chronological order in the test run. `Sentomist` then feeds the counters into the one-class SVM outlier detection algorithm. Part of the ranking scores are shown in Figure 5(a) in an ascending order[5]. It shows that the behaviors of top-ranked instances are suspicious to contain bug symptoms compared with the others. It thus directs us to a manual inspection of the system behavior: According to the ranking, we can inspect these instances (*i.e.*, the 76th, 176th, and 198th instances in testing run 1, and so on) one by one to check the correctness of the application.

We check the application behaviors during any of the

[3]http://compilers.cs.ucla.edu/avrora.

[4]http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[5]We normalize the scores so that the largest positive score is 1 for easy comparison purpose in all the three case studies.

top three instances of the ADC event procedure and they all confirm the data pollution bug discussed in Section IV. This shows that our testing paradigm can recommend just a few short program runtime intervals for manual inspection, where bug symptoms do present in these recommended short intervals. In contrast to inspecting the long-term system behaviors, our bug symptom mining approach can greatly save the human efforts in test inspection. It is worth noting that even for this simple application, the program trace of each testing run is very long (*e.g.*, when $D = 20ms$, the size of the function-level log can reach tens of megabytes). It is thus labor-intensive to manually inspect whether the WSN application runs correctly in each testing run without our proposed bug symptom mining approach.

C. Case study II: Packet loss in a multi-hop data forwarding WSN

We next consider multi-hop packet transmissions where intermediate nodes serve as relays between the source and sink. We test a typical multi-hop packet forwarding protocol based on `BlinkToRadio` distributed with TinyOS [6].

When a wireless interface chip (*e.g.*, CC1000 for Crossbow Mica2) receives a packet from its antenna, it will issue an SPI interrupt to the MCU. The packet-arrival event procedure is designed to process such an SPI interrupt for obtaining the packet content. A key function here is `Receive.receive`, which directly calls another function `AMSend.send` to forward the packet to next hop.

Our testing target in this case study is the packet forwarding operation, and we therefore consider a three-node setting in `Avrora`: node 0 as the sink, node 1 as the intermediate node, and node 2 as the source node. We are particularly interested in the behaviors of the intermediate node, *i.e.*, node 1. The test inputs are naturally the packet arrival events at node 1. By randomizing the packet sending ratio of node 2, we can inject a random sequence of packet arrival events for node 1 to handle. `Sentomist`runs the test for $20s$ and obtains a system lifecycle sequence. The program runtime is then anatomized into 195 event-handling intervals indexed by a chronological order of their starting time. During each interval an instance of the packet-arrival event procedure runs[6]. We therefore obtain 195 instruction counters.

Figure 5(b) shows the ranking results from one-class SVM outlier detection algorithm. Again, the top-ranked samples (*i.e.*, samples 20, 108, and so on) correspond abnormal symptoms, *i.e.*, those possibly with transient bugs, that are worth further investigation.

Instantly, we find that the system behaviors when processing packet 20 contain a bug symptom: Though node 1 has received the packet and calls `Receive.receive`, it actively drops the packet in `AMSend.send` due to a busy flag. An in-depth analysis reveals that the busy flag is set when node 1 is in the process of sending a data packet. Since WSNs generally adopt a carrier sense multiple access (CSMA) protocol to avoid packet collision, the process of sending a data packet involves several control packet exchanges. Specifically, a node needs to send an RTS packet and wait for a CTS packet before it can actually send the data packet. The busy flag is set during the whole process and cleared only if it is done when a corresponding ACK packet arrives (See `CC1000SendReceiveP.nc` in TinyOS 2.1.0 distribution [6]). Active packet drop due to the busy flag thus happens when the time interval between two packet arrivals is too short.

This bug is actually due to an improper design: the protocol should queue up a received packet and send it when the busy flag is cleared, instead of sending the packet immediately. The bug is triggered occasionally after certain complicated executions. Unfortunately, in practice it is difficult to identify such a tricky packet loss from other common wireless losses. As a result, it can be labor-intensive to justify the correctness of the protocol. Moreover, even we know the packet loss results from a bug, we have to check 195 event procedure instances one by one without `Sentomist`. Actually, we verify that only three of them contain bug symptoms, which are successfully ranked by `Sentomist` as the top three instances. Therefore, we can easily unveil such a bug symptom with `Sentomist` and human inspection efforts are greatly saved.

D. Case study III: Unhandled failure caused by two co-existing WSN protocols

In the third case study, we test a WSN application implemented for event detection application, where an event of interest lasts for a random interval. During the event interval, a sensor node will report its sensor readings to a sink periodically. We employ the Collection Tree Protocol (CTP) [12] implementation distributed with TinyOS 2.1.0 to transport sensor readings. We also implement a heartbeat message exchange protocol for monitoring the life conditions of sensor nodes, where a sensor node sends a heartbeat message to its neighbors every $500ms$. We deploy 9 nodes in `Avrora`, and randomly select sensor nodes as sources. Each source will randomly start reporting packets with CTP in a time interval with a random length.

As both the heartbeat protocol and CTP are driven by timer timeout events, we focus on the corresponding timeout event procedure at each sensor node. We run the test for $15s$. With a similar process described in the previous two case studies, `Sentomist` obtains 95 timer event-handling intervals, which are sampled as 95 instruction counters, for the timer to report sensing data in 4 sensors. The one-class SVM outlier detection algorithm provides their ranking results shown in Figure 5(c), in which each sample is indexed by $[n, s]$ where n is the node ID and s is the

[6]Since each of the instances corresponds to a packet arrival event, we also index the packets being forwarded with the same order.

chronological order of the corresponding instance of the timeout event procedure running on each node.

The results show that the behaviors of the WSN application running on node 8 during its 2nd event procedure instance is the most suspicious one to contain bug symptoms. We inspect the system behaviors; however, it seems fine. So do the second and the third ones. Yet we quickly find that a bug symptom exists in the rank-4 instance $[8, 20]$, where CTP cannot send a packet out. It reaches a status of `FAIL` due to resource contention with the heartbeat protocol. The failure status, however, is not properly handled in the current implementation of CTP. As a result, the corresponding mark that indicates the busy status of the underling communication chip is not reset. Hence, all the following packets are not sent out and the CTP protocol at the node hangs.

The failure status is unhandled in CTP because it assumes that it is the sole protocol responsible for transmitting all messages in a sensor node and the failure should never happen. However, multiple protocols with different purposes may co-exist in a senor to fulfil different tasks. Hence, uncoordinated resource contention among different protocols may be transiently triggered and the system eventually fails.

This bug is due to certain assumption a TinyOS component makes, which is however violated by other co-existing components. This is by no means an occasional situation. TinyOS applications are component-based and different components are typically coded by different developers around the world. They may not share the same design assumptions, resulting in tricky bugs when wiring the components into one WSN application. Note that the bug has been extensively discussed in the TinyOS mail list. It had long been causing confusing problems until it was discovered recently[7]. But with `Sentomist`, we can easily identify its symptoms and quickly locate it.

E. Discussions

Emulation is a popular way to check the functionality and performance of a WSN application [35]. In this paper, the system behaviors are collected based on emulation over `Avrora` by `Sentomist`. We do not rely on real experiments to capture the system runtime behaviors because the transient nature of WSN bugs caused by the random interleaving requires a long-term system execution. It is generally not cost-effective, if not infeasible, for a real system to explore a variety of system states (*e.g.*, different parameters) to hit the trigger condition of a transient bug. Hence, resorting to emulation is more efficient for fighting against transient bugs. We choose `Avrora` since it demonstrates high fidelity to the real world. `Avrora` models hardware behaviors and their interactions with high timing accuracy, and thus supports interrupt preemptions and network communications, which are of the most concerns in WSN application development. With `Avrora`, we can thus explore the interleaving executions of event procedures that may occur in practice. It is worth noting that another widely-adopted simulator TOSSIM [36] cannot provide such a high timing accuracy in hardware behaviors, since it simulates event in a consequential manner which will fail to capture the interleaving executions of event procedures [35].

`Sentomist` adopts one-class SVM as its plug-in anomaly detection approach. It is noted that one-class SVM is not the sole option. There are many other outlier detection algorithms [34] that can be incorporated into `Sentomist` such as Principal Component Analysis and one-class Kernel Fisher Discriminants. `Sentomist` can actually plug in these outlier detection algorithms conveniently. A further comparison study can be conducted in our future work.

[7]https://www.millennium.berkeley.edu/pipermail/tinyos-devel/2009-March/003735.html.

VII. Conclusion

WSN applications are fault-prone. Testing WSN systems is however a very challenging task, far from simply applying existing software testing techniques. Many WSN bugs are subtly caused by random interleaving executions of event procedures. It is extremely hard to handle such bugs since their symptoms are transient in nature, which are deeply hidden in tremendous system runtime data. It is labor-intensive, if not impossible, to examine whether a system behaves correctly or not. This paper presents an effective tool `Sentomist` for testing WSN systems. `Sentomist` divides the long-term system runtime data in a proper granularity, *i.e.*, the event-handling intervals. It captures the system behaviors of each interval with an instruction counter profile. Anomaly is then detected with a plug-in outlier detection algorithm. The symptoms of potential bugs are thus exposed for human inspections. We apply `Sentomist` to testing transient bugs in several representative real-life WSN applications. Our experiments demonstrate that `Sentomist` can greatly save manual efforts in testing WSN applications. Finally, in our future work, we are interested in extending `Sentomist` for achieving bug localization, *i.e.*, locating bugs in source code level, by adopting the symptom-mining approach to correlate bug symptoms with source codes.

Acknowledgement

The work described in this paper was supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4154/09E). The work of J. Liu was supported by the Natural Sciences and Engineering Research Council of Canada under a Discovery Grant and a Strategic Project Grant. We would like to thank Mr. Wujie Zheng for his various suggestions and kind aids that help shape this work.

References

[1] J. Kahn, R. Katz, and K. Pister, "Next century challenges: Mobile networking for "smart dust"," in *Proc. of the ACM MOBICOM*, Seattle, Washington, Aug. 1999, pp. 271–278.

[2] K. Langendoen and A. B. O. Visser, "Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture," in *Proc. of the International Workshop on Parallel and Distributed Real-Time Systems*, Apr. 2006.

[3] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proc. of the USENIX OSDI*, Seattle, USA, Nov. 2006, pp. 381–396.

[4] ON World Inc., "WSN for smart industries: A market dynamics report," Sep., 2007.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proc. of the ACM ASPLOS*, 2000.

[6] TinyOS Community Forum, "TinyOS: An open-source OS for the networked sensor regime," http://www.tinyos.net.

[7] E. Cheong, J. Liebman, J. Liu, and F. Zhao, "TinyGALS: A programming model for event-driven embedded systems," in *Proc. of the ACM SAC*, Mar. 2003, pp. 698–704.

[8] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proc. of the ACM MobiSys*, 2005, pp. 163–176.

[9] P. Jalote, *An Integrated Approach to Software Engineering*, 3rd ed. Springer, 2005.

[10] M. R. Lyu, *Handbook of Software Reliability Engineering*. McGraw-Hill, 1996.

[11] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han, "Dustminer: Troubleshooting interactive complexity bugs in sensor networks," in *Proc. of the ACM SENSYS*, Nov. 2008, pp. 99–112.

[12] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection tree protocol," in *Proc. of the ACM SENSYS*, Nov. 2009, pp. 1–14.

[13] J. Regehr, "Random testing of interrupt-driven software," in *Proc. of the ACM EMSOFT*, Sep. 2005, pp. 290–298.

[14] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*, 2nd ed. Elsevier Science, 2009.

[15] Y. Zhou, X. Chen, M. R. Lyu, and J. Liu, "Sentomist: Unveiling transient sensor network bugs via symptom mining," http://www.cse.cuhk.edu.hk/~yfzhou/Sentomist.

[16] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, Oct. 1988.

[17] M. R. Lyu, J. R. Horgan, and S. London, "A coverage analysis tool for the effectiveness of software testing," *IEEE Transactions on Reliability*, vol. 43, no. 4, pp. 527–535, Dec. 1994.

[18] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for GUI-based software applications," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, Feb. 2007.

[19] Y. Lei and R. H. Carver, "Reachability testing of concurrent programs," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 382–403, Jun. 2006.

[20] Z. Lai, S. C. Cheung, and W. K. Chan, "Inter-context control-flow and data-flow test adequacy criteria for nesc applications," in *Proc. of the ACM FSE*, Nov. 2008, pp. 94–104.

[21] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr, "Efficient memory safety for TinyOS," in *Proc. of the ACM SENSYS*, Sydney, Australia, Nov. 2007, pp. 205–218.

[22] V. Krunic, E. Trumpler, and R. Han, "NodeMD: Diagnosing node-level faults in remote wireless sensor systems," in *Proc. of the ACM MobiSys*, Jun. 2007, pp. 43–56.

[23] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler, "Marionette: Using RPC for interactive development and debugging of wireless embedded networks," in *Proc. of the ACM IPSN*, 2006.

[24] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse, "Clairvoyant: A comprehensive source-level debugger for wireless sensor networks," in *Proc. of the ACM SENSYS*, Nov. 2007.

[25] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo, "Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks," in *Proc. of the ACM SENSYS*, Nov. 2008.

[26] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic, "Achieving repeatability of asynchronous events in wireless sensor networks with EnviroLog," in *Proc. of the IEEE INFOCOM*, Apr. 2006, pp. 1–14.

[27] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proc. of the ACM/IEEE ICSE*, 2002.

[28] W. Zheng, M. R. Lyu, and T. Xie, "Test selection for result inspection via mining predicate rules," in *Proc. of the ACM/IEEE ICSE Companion*, May 2009, pp. 219–222.

[29] P. Levis and D. Gay, *TinyOS Programming*. Cambridge University Press, 2009.

[30] D. Gay, M. Welsh, P. Levis, E. Brewer, R. von Behren, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proc. of the ACM PLDI*, Jun. 2003, pp. 1–11.

[31] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Addison-Wesley, 2007.

[32] V. N. Vapnik, *The Nature of Statistical Learning Theory*. Springer, 2000.

[33] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural Computation*, vol. 13, no. 7, pp. 1443–1471, Jul. 2001.

[34] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, pp. 15:1–15:58, Jul. 2009.

[35] B. Titzer, D. Lee, and J. Palsberg, "Avrora: Scalable sensor network simulation with precise timing," in *Proc. of the IEEE IPSN*, May 2005.

[36] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in *Proc. of the ACM SENSYS*, Nov. 2003, pp. 126–137.