Lightweight Tracing for Wireless Sensor Networks Debugging*

Vinaitheerthan Sundaram[a], Patrick Eugster[b], Xiangyu Zhang[b]
School of Electrical and Computer Engineering[a], Department of Computer Science[b]
Purdue University, West Lafayette, Indiana, U.S.A.
{vsundar,peugster,xyzhang}@purdue.edu

ABSTRACT

Wireless Sensor Networks(WSNs) are being increasingly deployed in the real world to monitor the environment and large industrial infrastructures. The extreme resource constraints inherent to WSNs, the in situ deployment in harsh environments and the lack of run-time support tools make debugging and maintaining WSN applications very challenging. In particular, run-time debugging tools are required to detect and diagnose complex run-time faults such as race-conditions, which occur due to unexpected interaction with the real-world environment.The ability to repeatedly reproduce the failure by replaying the execution from the trace of events that took place can play a crucial role in debugging such faults. Obtaining such a trace is made difficult due to tight resource constraints. In this paper, we propose a lightweight tracing tool for WSNs which uses a novel control flow tracing and encoding scheme to generate a highly compressed control-flow trace. In addition to the construction of the trace, our tracing tool supports storing the trace in non-volatile memory and querying interface that allows base station to retrieve the trace when needed. We show the effectiveness of our tracing tool through a case-study and illustrate its low overhead through measurements.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Testing and Debugging—*tracing*; C.3 [**Special-Purpose and Application-Based Systems**]: embedded networked systems

General Terms

Debugging Wireless Sensor Networks

Keywords

program tracing, trace encoding, wireless sensor networks, NesC, control flow

*Financially supported by NSF under grant 0834529.

MidSens'09, Novemeber 30 - December 4, 2009 Urbana Champaign, Illinois, USA

1. INTRODUCTION

Wireless Sensor Networks(WSNs) are being increasingly deployed in the real world to monitor the environment and large industrial infrastructures[14]. However, the extreme resource constraints inherent to WSNs, the in situ deployment in harsh environments and the lack of run-time support tools make debugging and maintaining WSN applications very challenging. In particular, run-time debugging tools are required to detect and diagnose complex run-time faults such as race-conditions, which occur due to unexpected interaction with the real-world environment. Therefore, run-time debugging tools are required to detect and diagnose these defects in the post- deployment phase.

There have been several debugging solutions proposed for WSNs. (1) Automated debugging tools mostly monitor the network which doesn't provide much help for programmer errors as causes and symptoms may be far apart, or impose expensive synchronization to achieve such resilience; alternatively, programmer are required to express invariants to guide runtime monitoring or use SQL-based queries for debugging. (2) "Remote control" approaches try to provide insight into — and control of — remote sensor nodes. These incur however a substantial overhead and put much load on the programmer who has to manually navigate though the program at runtime. Finally, (3) program-analysis based tools provide higher-level understanding of programs but fail to quickly pinpoint the exact causes of faults or suffer from high complexity making them only applicable to very small programs.

The ability to repeatedly reproduce the failure by replaying the execution from the trace of events that took place can play a crucial role in debugging complex run-time faults in distributed systems[3]. Such a replay requires the trace of the ordered sequence of events, which may not present insurmountable issues in a wired setting; obtaining it in a WSN can however be prohibitively expensive in terms of storage and bandwidth required to send the trace from a node to the base station. Tight restrictions do not only exist on bandwidth, but similarly also on computation and storage resources, which makes it challenging to devise an efficient tracing scheme altogether. In summary, the key challenge in enabling replay-based debugging tool is to devise a lightweight tracing tool.

In this paper, we propose a lightweight tracing tool for WSNs which uses a novel control flow tracing and encoding scheme to generate a highly compressed control-flow trace. In addition to the construction of the trace, our tracing tool supports storing the trace in non-volatile memory and a

querying interface that allows base station to retrieve the trace when needed.

The control-flow trace constructed contains the trace of all interleaving concurrent events, thus allowing faithful replay of the original execution. To be able to deal with long-running real-life programs, our tracing tool incurs very low overhead. This is achieved by statically analyzing the program and instrumenting every procedure call to record a few bytes (3 bytes in our example) per procedure such that all the interleaving concurrent events are captured. At runtime, the trace is recorded in memory and compressed before being written into non-volatile external flash memory. Since WSN applications typically execute the same sequence of events repeatedly, these traces can be compressed quite well. When the allocated flash memory is full, the least recent trace is overwritten with a new trace.

Since WSN program executions are event-driven, every event handler is a starting point in the program. Therefore, the trace since the application start is not needed. When a fault is detected, upon request from a system manager, or at predefined points in the program, the trace is sent to the base station for analysis. The programmer can replay the trace in a controlled environment such as a simulator or debugger to reproduce the fault. In fact, the trace enables reverse debugging of the program. Thus, the programmer will be able to identify the defect effectively. We illustrate the effectiveness of our approach through a case study and demonstrate its low overhead through performance measurements.

More precisely, we make the following contributions in this paper:

- A novel control flow tracing and encoding scheme is presented for WSNs
- We introduce a tracing method that satisfies the stringent resource constraints of WSNs, making use of compression techniques to that end.

Roadmap

Section 2 presents the design and implementation of our encoding and tracing techniques. Section 3 illustrates the effectiveness of our approach through cases study, and dissects its overhead. Section 4 presents related work. Section 5 concludes with final remarks and an outlook on future work.

2. DESIGN AND IMPLEMENTATION

In this section, we first explain the challenges in lightweight tracing and our approach to solve those challenges. We next present our algorithms for lightweight tracing. We implemented our solution in TinyOS, a popular WSN Operating System and the important implementation details are discussed next.

2.1 Challenges And Solution Overview

We observe that complex failures are often due to unexpected interaction of different interrupt handlers. For example, the receiver buffer being simultaneously modified by an application and the network layer is noted as a common error in WSN programs but difficult to diagnose[5]. In order to diagnose these complex failures, we note that it is sufficient to record the program's control flow.

Interrupts And Concurrent Execution.

In WSN applications, there are two types of concurrency model, one thread based and the other event-based. Operating systems such as Contiki, Mantis use thread-based model while TinyOS uses event-based concurrency model. Both these models are interrupt-driven so as to keep the system responsive. The presence of concurrent executions and being interrupt driven make statically enumerating all possible program paths prohibitively expensive as the combination of paths grows at an exponential rate. Futhermore, sensors features very limited computation resource, which renders fine-grained tracing extremely challenging. Simply recording every executed instruction is not possible.

Control Flow Path Encoding.

In the seminal paper [1], Ball and Larus proposed an efficient algorithm (referred as BL algorithm) to represent each possible intra-procedural control flow path with an integer between zero and the total number of paths less one. The BL algorithm features translating acyclic path encoding to instrumentations on control flow edges. At runtime, a sequence of these instrumentations are executed following a control flow path, resulting in the path identifier being computed. The idea can be illustrated by the example in Figure 1. The code is shown on the left and the control flow graph is shown on the right. The instrumentations are marked on control flow edges. Before the first statement, `lcount` is initialized to 0. If the false branch is taken at line 1, `lcount` is incremented by 2. If the false branch is taken at line 5, `lcount` is incremented by 1. As shown on left bottom, execution taking different paths leads to different values in `lcount`. In other words, `lcount` encodes the path. This encoding has been shown to be minimal in terms of number of bits used to represent a control flow path. We observe that this control-flow path encoding technique is particularly attractive to WSNs because of the inherent repetition of computation in WSNs and small code size of WSN applications.

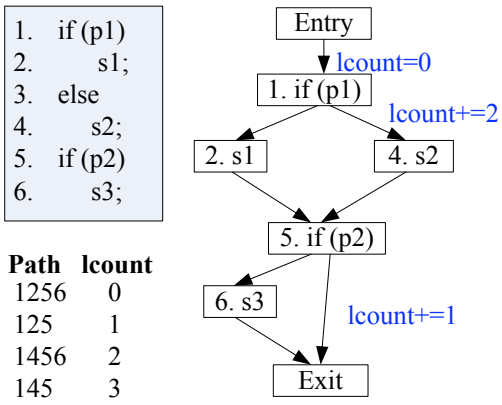

Figure 1: Example for Ball-Larus path encoding. Instrumentations are marked on control flow edges

Basic Idea.

The basic idea of the proposed work is presented as follows. First, we instrument the control flow graph (CFG) with the BL algorithm increments to compute path encoding. Then, we instrument the code to record the start and

end and the identifier of the control-flow path taken for all procedure. Note, the end of the procedure and identifier for the control flow path taken are recorded *atomically.* When there are multiple concurrent threads of execution, recording the start and end of procedure allows us to infer the interleaving among these threads.

Note that it is possible to have concurrent invocations of a procedure, which have to be distinguished at run-time. For example, the event handler for a timer can be invoked before an earlier invocation completes execution. To handle this case each procedure label recorded will also contain session identifier in addition to procedure identifier.

The trace recorded in the memory is compressed and written to non-volatile flash memory often. We use three simple compression techniques based on the following three observations.

- when a procedure's start and end appear back to back in the trace, meaning no interruptions occurred in between, it is enough to record just the procedure name and the path information. This implies start and end identifier can be combined to one symbol.
- WSN applications tend to repeat the same sequence of procedures most of the time and therefore, representing these sequences of procedures with one symbol can significantly reduce the trace.
- when there are loops inside a procedure, the Ball and Larus algorithm will generate an entry in the trace for every loop iteration. Since a loop internal path often repeats itself for many times, we can represent the repeating path with a path id and its repitition count.

The tracing tool supports a simple query interface that allows retrieval of traces from the base-station. When a trace request is received, it formats a reply packet containing most recent compressed trace information and sends it to the base station using the underlying multi-hop protocol.

2.2 Algorithms

First, we present the instrumentation algorithm, which explains how each procedure is instrumented with increments and what is recorded to the trace. Next, the compression and storage algorithm explains the compression techniques used and storing the trace in non-volatile memory. In the interest of space, we do not show our simple query interface implementation.

Instrumentation.

The instrumentation algorithm is shown in algorithm 1. The nodes in the CFG are annotated with the number of paths rooted from that node using BL algorithm. The entry node is instrumented with initialization of "count" and the identifiers "procedure label" and "session identifier". For each edge in the control flow graph , the edge increment is calculated and corresponding incrementing counter code is inserted. At the exit, the procedure label and the path taken as stored in the count are recorded in the trace. Since multiple invocations of a procedure is possible, a session identifier is used to uniquely identify a procedure invocation at runtime.

Compression And Storage.

The compression and storage implements the three levels of compression explained before. We define the procedure label (`procLabel`) such that the compression and decompression is simple. A procedure label contains procedure identifier bits, compressed flag bit, session bits and start or end identifier bit. For example, assume a procedure label has 8 bits, then the first 4 bits represent procedure identfier, the next 2 bits represent session identifier and the next bit represents compression and the last bit represents the start/end of the procedure. If the compression bit is on, then the procedure identifier is followed by the path taken and the start/end bit is ignored. The session bits represent the number of concurrent invocation or session of the same procedure in progress. The number of session bits is configurable. Since it is very rare to have multiple concurrent invocations of the same procedure, two bits are usually enough and if there are more than expected concurrent sessions of the same procedure, it is a strong indicator of an error such as indefinite recursion. The start/end bit is valid when the compression bit is off and it indicates the start or end of a procedure. A unique procedure identifier is assigned for every procedure in the instrumented component. Additional unique identifiers are used to represent sequences (of procedures). Suppose there are 8 procedure and we use 4 bits to represent them, there are 8 free procedure identifiers that can be used for representing sequences.

Algorithm 1 Instrumenting a procedure P whose CFG is denoted by G

annotate each node in G with the number of paths rooted at the node.
remove the backedges due to loops from G
add dummy edges for loops to G as expected by BL algorithm
instrument the entry node in G with "`lcount=0; procid= getProcedureID(P); sid = getUniqueSessionID (procid); procLabel = createProcLabel(procid, sid) Tracer.ProcBegin(procLabel)`"
for each edge $n \rightarrow m$ in G **do**
  $s \leftarrow 0$
  for each edge $n \rightarrow t$ and t precedes m in n's edge list **do**
    $s \leftarrow s$+t's annotation
  end for
  instrument $n \rightarrow m$ with "`lcount+=s`"
end for
for each backedge due to loops $m \rightarrow n$ **do**
  instrument the loop exit node m with "`Tracer.ProcEnd(procLabel, lcount); lcount=0`"
end for
instrument the exit node with "`Tracer.ProcEnd(procLabel, lcount)`"

Decompression can be done by a simple parsing of the procedure label. If the compression bit is turned on, then the decompression algorithm checks if it is a sequence. If it is not a sequence, the next byte is the path taken in that procedure. If the compress bit is turned off, the algorithm simply copies that byte.

2.3 Implementation

In this section, we present important implementation de-

Algorithm 2 Compression and Flash Storage Algorithm in Tracer component

```
if tracebuf[curr] is full then
  copy lbuf ← tracebuf
  for every consecutive procedure start and end exists in lbuf do
    Replace them with just procedure id that has compressed bit on.
  end for{Level 1 Compression}
  for every pattern of call sequence to procedures exist do
    Map the pattern to new id
    Replace every instance of the pattern with the new id
  end for{Level 2 Compression}
  for every repetition of procedure id and path do
    Map the procedure id and path to a new id
    Replace all the procedure id and path with the new id and number of repetitions
  end for{Level 3 Compression}
  copy the compressed bytes to flashpage[curr]
  if flashpage[curr] is full then
    swap current pointer to use the other flashpage while this page is being written to flash
    initiate flashWrite
  end if
end if
```

tails. We implemented our lightweight tracing tool in Tiny OS and tested on mica2 motes. The tracer is implemented as a component which has a trace buffer of length 192 bytes and 2 flash pages each of which is 16 bytes long. It implements two commands which are called from entry and exit blocks of each procedure. Each procedure name is mapped to a unique identifier. At the procedure entry, each procedure is also assigned a unique session identifier. These two identifiers distinguish the procedure from every other procedure invocation and thus concurrent invocation can be recorded unambiguously.

To create a session identifier per invocation of a procedure, we use a global session identifier buffer. When a procedure is invoked, it increments the bits corresponding to the procedure in the global buffer and uses the incremented value as session identifier. When a procedure exits, it returns the session identifier by decrementing the bits in the global buffer. The number of session identifier bits is configurable.

At the exit of a procedure, the compress task is posted if a threshold amount of bytes has accumulated in the trace buffer. This task copies the global buffer to local memory and compresses them as explained in algorithm 2. The compressed bytes are written to the current flash page. If current flash page is not full and another flash write is not in progress, flash write is started. We use modulo arithmetic to handle circular buffers.

3. EVALUATION

In this section we describe our preliminary evaluation and promising results. We evaluated our lightweight tracing scheme for its effectiveness and run-time overhead. We show the effectiveness of our approach through a case study of diagnosing a tricky fault in TinyOS 1.x. We show that our approach is lightweight by measuring the run-time overhead incurred for tracing.

3.1 Case Study

In this section, we describe a case study of a tricky fault [16] in TinyOS 1.x to show the effectiveness of our approach. This fault was in the TinyOS 1.x CC1000 radio stack and was fixed in version TinyOS version 1.1.8. We demonstrate how our technique could have helped in diagnosing this fault. The code snippet containing this fault is shown in the Figure 2.

When a transmission is done (`TXSTATE_DONE`), the CC1000 radio SPI interrupt handler is supposed to notify the application using **post** `PacketSent()` and if that post is successful, switches the radio to receiving mode using functions `SpiByteFifo.rxMode()` `CC1000Control.RxMode()` and changes radio state to `IDLE_STATE`. As shown by lines 13-16 in Figure 2, the functions to switch radio state are called before posting the `PacketSent` task. The problem is that when the task queue is full, the post task fails and the failure execution path takes a relatively long time (around 400 microseconds) because the failure execution path has functions to switch radio state which invoke `wait()`. The SPI interrupt rate is so high that it essentially starves the task queue, which means posting `packetSent()` would keep failing and the node hangs forever. To fix this, the function to switch radio state are called only if the post succeeds as shown in the lines 16-18 in Figure 2. In this case, when the task queue is full, the failure execution path is not long and the task queue is not starved, thus allowing post task to succeed eventually.

We used the same hack used in [16] to reproduce the fault and these lines of code are shown as comments in Figure 2. We applied our control flow tracing technique to the CC1000 radio component. The trace was recovered using a watchdog timer after the node deadlocked. The trace contained the repeated invocations to SPI interrupt handler with the path traversing along the case `TXSTATE_DONE` and encountering a failure in posting the `PacketSent` task. This is a clear indication of task queue overflowing and the repeated interrupt handler executions starving the task queue. Without the trace, it requires substantial amount of work to diagnose the fault. This case study demonstrates power of control-flow tracing in diagnosing tricky faults.

```
1 //task void shortDelay() { TOSH_uwait(10); }
2
3 //void postDelay() {
4 //  for(i=0;i<8 i++) post shortDelay();
5 //}
6
7 async event result_t
8  SpiByteFifo.dataReady(int8_t data_in) {
9   ...
10  case TXSTATE_DONE:
11  default:
12     // if (bTxPending == TRUE) { postDelay(); }
13    - call SpiByteFifo.rxMode();
14    - call CC1000Control.RxMode();
15    bTxPending = FALSE;
16    if (post PacketSent()) {
17    +    call SpiByteFifo.rxMode();
18    +    call CC1000Control.RxMode();
19          RadioState = IDLE_STATE;
20  ...
21 }
```

Figure 2: Partial listing of CC1000 radio stack code

3.2 Runtime Overhead

The important run-time overhead include the usage of external storage and the data memory in motes. We measured these overheads and show they are low. In addition, we show that the increase in code size or program memory after our instrumentation is also acceptably low.

Since there is no competing tracing solution available for NesC programs (see section 4), we chose three WSN applications, namely, Surge, Oscilloscope, and Blink that are included with the operating system TinyOS 1.x for our evaluation. Surge and Oscilloscope are widely used programs and are representatives of data collection tasks. The Blink application toggles the red LED every second. Since it is one of the simplest TinyOS programs, the instrumentation overhead will be highlighted. Since it performs a single task, repeatedly, it offers the best scenario for our compression algorithm. The Oscilloscope application records light sensor values every 125 milliseconds and accumulates these values in memory and when a threshold number of entries are recorded, it sends the accumulated sensor values in a message to the base station. In addition, when it receives the `resetCounter` message from the base station, it resets the accumulated counter to zero and starts accumulating afresh. Since this application timer fires 8 times in a second, it can generate large amounts of trace information and thus stretches our run-time tracing scheme. The Surge application is similar in functionality to Oscilloscope but uses a slower timer that ticks about every 2 seconds which makes it representative of common WSN applications.

Table 1 shows the memory and external storage overhead incurred by our tracing scheme for the three programs. By design we use a circular buffer in the flash memory that stores the trace for the past hour. Therefore, we measure the external flash storage per hour for our three programs. The external flash storage size is at least 512 KB for mica and telosb families. Table 1 shows the actual flash storage per hour and the space savings obtained due to compression. The space savings is defined as $(U - C)/U$ where U is the size of the uncompressed trace and C is the size of the compressed trace. We see that the amount of trace information generated in an hour for a normal WSN application is only a few KB (5KB for Surge) and for a high-throughput application like Oscilloscope is about 50KB. Moreover, we see that our simple compression techniques are effective as we get upto 91.74 percent space-savings ratio.

The data memory requirement is around 315 bytes and strongly independent of the program instrumented. The program memory represents the code size and the data memory represents the working memory/RAM size. We obtained these values by compiling with default switches. We note that in all our example programs, the program memory requirement is about 5 KB, which is low when considering the 128 KB mica (mica2,micaz) family or 48 KB telosb families of motes. The exact program memory requirement depends obviously on the number of procedures in the instrumented program.

Table 1: Storage overhead for control-flow tracing

Metric	Example	Overhead (in bytes)
Flash Storage **per hour** (space savings ratio)	Surge	5392 (75.04%)
	Oscilloscope	52704 (72.92%)
	Blink	912 (91.74%)
Data memory	Surge	315
	Oscilloscope	315
	Blink	315
Program memory	Surge	5206
	Oscilloscope	5156
	Blink	4928

4. RELATED WORK

The existing work in WSN debugging can be classified into (1) tools that aid in automating debugging, (2) tools that provide visibility into the network and (3) tools that analyze programs for testing or extracting higher-level abstraction.

4.1 Automating Debugging

In [8], Krunik et al. proposed NodeMD, a tool that can detect stack overflow, livelock, and deadlock. NodeMD is similar to our work in that they instrument the code and encode high-level events with a small number of bits to record traces. However, the trace recorded does not contain the exact control-flow path information and hence cannot be automatically replayed later to reproduce fault from a trace. Luo et al. [11] proposed recording all events in a given time period and replaying within the node at a later point in time. The trace recorded contains function calls and the input parameters. Such a functionality is targetted at in-field or post-deployment testing. Moreover, recording function calls and input parameters is not highly compressible like ours and therefore require larger storage. Österlind et al. [12] proposed checkpointing network state in testbeds to reproduce the execution later. Similar to Envirolog, the checkpoints are not highly compressible and may not be applicable in post-deployment settings.

Sympathy [13] collects network metrics such as connectivity and data flow, and node metrics periodically to detect failures and localize it to a node or sink or the communication path itself. Passive Diagnosis [10] uses lightweight monitoring of the network such as packet marking to infer the failure and its cause using bayesian network analysis. Sympathy and PAD focus on network faults and do not handle application implementation defects.

In [4], Herbert et al. present an invariant-based application-level runtime monitoring tool that can be used to detect errors in WSN applications. This tool doesn't have the diagnostic framework required to find the *root* cause of the error detected. Declarative TracePoints [2] propose an SQL-based language interface for debugging. Dustminer [5] propose machine-learning based approach to diagnosis. It uses the pattern mining algorithm to distinguish good execution from bad execution in the logs collected. Our trace can be used in both these techniques for diagnosis.

4.2 Visibility

Since the motes doesn't include a user-friendly interface but only three LEDs, it is important to provide visibility into the state of the network to gain confidence that the network is functioning properly. Some of the important tools that provide visibility into the network include Marionette[15], Clairvoyant [16] and Hermes [7]. These visibility tools in-

cur large run-time overhead as they require several message exchanges before the root causes can be found. However, by recording traces and starting to diagnose only when an error is detected, we can reduce this run-time overhead significantly.

4.3 Program Analysis Based Tools

The program analysis based tools are mainly used for test coverage [9] and for mining finite-state machines from code to get better understaing of the code[6]. But these tools are not meant for run-time detection of failures and diagnosis of faults like our tracing does.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have shown that program tracing can be performed efficiently and accurately in wireless sensor networks. To that end, we have presented a lightweight tracing tool which consists of a novel trace encoding scheme and a basic trace compression scheme and supports trace storage and retrieval. We showed the effectiveness of our approach through a case study and illustrated its overhead through measurements.

Backed by this evidence of the viability of our approach, we are exploring different improvements to our technique. For instance, the unrolling of loops can potentially reduce the number of entries into trace buffers; our code for runtime trace generation is currently weakly structured, which leads to same code being inserted in many places during instrumentation, instead of using shared procedures; the query interface can support requesting particular trace information.

6. REFERENCES

[1] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[2] 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 *SenSys '08: Proceedings of the 3rd international conference on Embedded networked sensor systems*, New York, NY, USA, 2008. ACM.

[3] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association.

[4] D. Herbert, V. Sundaram, Y.-H. Lu, S. Bagchi, and Z. Li. Adaptive correctness monitoring for wireless sensor networks using hierarchical distributed run-time invariant checking. *ACM Trans. Auton. Adapt. Syst.*, 2(3):8, 2007.

[5] M. Khan, H. Le, H. Ahmadi, T. Abdelzaher, and J. Han. Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In *SenSys '08: Proceedings of the 3rd international conference on Embedded networked sensor systems*, New York, NY, USA, 2008. ACM.

[6] N. Kothari, T. Millstein, and R. Govindan. Deriving state machines from tinyos programs using symbolic execution. *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference*, pages 271–282, April 2008.

[7] N. Kothari, K. Nagaraja, V. Raghunathan, F. Sultan, and S. Chakradhar. Hermes: A software architecture for visibility and control in wireless sensor network deployments. *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, pages 395–406, April 2008.

[8] V. Krunic, E. Trumpler, and R. Han. Nodemd: diagnosing node-level faults in remote wireless sensor systems. In *MobiSys '07: Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 43–56, New York, NY, USA, 2007. ACM.

[9] Z. Lai, S.-C. Cheung, and W.-K. Chan. Inter-context control-flow and data-flow test adequacy criteria for nesc applications. In *SIGSOFT '08/FSE-16: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, 2008. ACM.

[10] K. Liu, M. Li, Y. Liu, M. Li, Z. Guo, and F. Hong. Pad: Passive diagnosis for wireless sensor networks. In *SenSys '08: Proceedings of the 3rd international conference on Embedded networked sensor systems*, New York, NY, USA, 2008. ACM.

[11] 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. *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–14, April 2006.

[12] F. Österlind, A. Dunkels, T. Voigt, N. Tsiftes, J. Eriksson, and N. Finne. Sensornet checkpointing: Enabling repeatability in testbeds and realism in simulations. In *EWSN'09*, volume 5432 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2009.

[13] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 255–267, New York, NY, USA, 2005. ACM.

[14] K. Römer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, Dezember 2004.

[15] 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 *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 416–423, New York, NY, USA, 2006. ACM.

[16] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 189–203, New York, NY, USA, 2007. ACM.