Program Representations for Testing Wireless Sensor Network Applications

Nguyet T.M. Nguyen
Department of Computer Science
University of Virginia
ntn3a@cs.virginia.edu

Mary Lou Soffa
Department of Computer Science
University of Virginia
soffa@cs.virginia.edu

ABSTRACT

Because of the growing complexity of wireless sensor network applications (WSNs), traditional software development tools are being developed that are specifically designed for their special characteristics. However, testing tools have yet to be proposed. One problem in developing testing tools is the need for a program representation that expresses the execution behavior. Due to characteristics of WSN applications that use a concurrent, event-based execution model, a representation is challenging to develop. In this paper, we present novel representations for WSNs applications that express the execution behavior of event and tasks, the major components of a WSN application. Our representations include a task posting graph, an event graph and finally an application graph that expresses the relationships among events and tasks as well as both timing and environmental interrupts. These representations are the first step in developing testing tools for WSN applications. Based on the graphs, traditional and event-based coverage criteria can be evaluated. When combined with individual Control Flow Graphs(CFGs) of events and tasks, the graphs' paths can be used as a criterion for evaluating the completeness of the test cases.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Verification

Keywords

program representation, wireless sensor networks, test criteria

DoSTA'07, September 4, 2007, Dubrovnik, Croatia.

1. INTRODUCTION

Wireless sensor networks (WSNs) continue to expand in both the number of such networks and the size of the networks as the recognition of their importance increases and technology improves. They are used in many different applications, from monitoring and controlling aspects of the environment to managing safety critical events such as the braking system on a car. Indeed, scientists have forecasted that WSN applications will dominate the software market in the near future [3].

As the demand and complexity of the applications increases and market forces become more competitive, developers of WSN applications are turning to traditional software development techniques. Thus, WSN applications are being implemented in high level languages such as Java and source level debuggers are being developed [13]. However, tools to systematically test WSN applications have yet to be developed.

Identifying software faults is one of the significant challenges in developing WSNs systems today. Due to resource limitations and the desire to develop more complex functionality, applications on WSNs often have a limited capacity to handle errors: the software crashes before being able to inform the user about the error. Moreover, as WSNs are mainly deployed in hard-to-reach locations, patching and fixing software after deployment is very costly. Therefore, testing tools for WSNs are critical components to help provide the reliability that is needed. Unfortunately, neither the WSN nor the software engineering research communities have applied systematic, structural software testing or regression testing tools and techniques to this domain. Current testing of WSN has depended on the use of ad-hoc methods using simulators and either simulated, profiled or randomly generated inputs from the environment [6, 10, 11]. However with this approach, the application is not being tested on the actual hardware or in the actual environment in which it will run.

WSN applications are concurrent, event-driven systems which frequently interact with their environments through interrupts and events that happen at arbitrary times. Therefore, to analyze and test WSN applications, we need to be able to determine the impact of these interrupts on the code. Currently, there is no comprehensive set of testing tools, or a testing framework, that can be used to systematically test WSNs that takes into account the execution environment and its execution model.

An important component of a testing framework is a program representation that can model all possible execution behaviors of the software. Because of the novel characteris-

tics of WSN applications, conventional representations such as CFGs or event graphs [9] cannot adequately model the execution behavior. Although there have been some efforts to construct representations for WSNs [4, 12], they have been developed for a purpose other than testing and their models are not fine-grained enough to express the execution behavior at statement level; thus they cannot be employed for structural testing purposes.

Because WSN applications employ an event-based, concurrent execution model, challenges arise in the development of a detailed representation that can accurately express every program execution path. For instance, in tinyOS [1], the most widely used operating system for WSNs, program execution is rooted in hardware interrupts caused by timers, sensors, and communication devices. Long running processes can be triggered in response to these interrupts by tasks that are placed in a single task queue. Execution order is tightly bound to the timing of interrupts, so that the sequencing of events cannot be determined using only source code. As a result, conventional representations, such as call graphs and CFGs, are helpful for extracting some information, but cannot be used to determine the order of execution of events and tasks and their relationships. The event graph that Memon proposed for representing a GUI [9] cannot represent the concurrency of WSNs. Hence, a new representation for WSNs, which does consider the functionality of the code as well as the possible behavior of events and interrupts that can occur on motes is necessary to facilitate the development of testing tools, and in particular the definition and evaluation of testing coverage criteria.

In this paper, we present a representation of an application executing on a mote of a WSN that expresses the execution order of events and tasks, and thus program paths. Our representation expresses (a) the time at which events are anticipated and (b) the order in which a certain event directly or indirectly posts its tasks to the system task queue. In particular, we present three representations: an event graph and a task posting graph, which together are used to form an application graph. The graph for an event expresses the order in which tasks are posted; the task posting graph represents the order in which tasks post other tasks and the application graph represents all of the events in an application. After integrating the CFG of event handlers, tasks and both timing and environmental interrupts, our application graph can be used to obtain all program execution paths, which can be used as a testing coverage criterion. Conventional code coverage criteria such as all-events coverage, all event sequence coverage and def-use can be naturally derived and checked.

It should be noted that our representation is for a single mote running in a network. Our representation expresses communication from other motes to the mote and the interrupts that can occur. We do not represent the entire network though, which is a part of our future work.

In the next section, we present a brief background of WSN applications. Then, challenges in developing representations for WSN applications are discussed in Section 3. Section 4 describes each of the three representations in detail. We then briefly discuss our implementation in Section 5. Section 6 addresses related work.

2. WSN BACKGROUND

Among existing executing systems for wireless sensor networks, tinyOS is the most commonly used operating system [1]. Although tinyOS assumes a particular execution model, other systems, such as SensorSim [10] and EmStar [6], have very similar characteristics to those of tinyOS. Therefore, to provide focus, our research is directed to establishing a representation for an WSN application running on tinyOS. Although we focus on tinyOS and the language NesC, our technique is general for other wireless sensor networks that use a concurrent, event driven execution model.

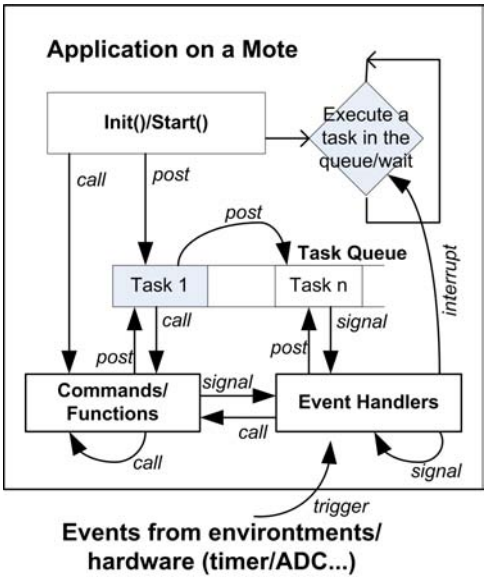

Figure 1: Elements of an application on a mote and their interactions

To support event-based programming of WSNs, tinyOS introduces a new concurrency model with two types of threads: tasks and hardware event handlers. A task is a non-preemptive function, which is posted into the task queue and its execution is deferred until it is at the front of the queue. After the mote is initialized and started, the application enters a loop to perform any tasks in the task queue in the order of posting. If the task queue is free, the mote goes to sleep. If a hardware interrupt (event) occurs, the corresponding hardware event handler is activated, preempting tasks, functions and other event handlers. The event itself can post other tasks and this process is repeated. Therefore, the order of events and tasks during execution cannot be determined statically from the program code because of the implicit control flow caused by events and deferred tasks. As a result, the representation needs to model not only the code of application but also the order in which tasks are posted as well as all possible orders in which events can occur.

NesC [5], a dialect of C, is specifically designed for developing WSN applications running with tinyOS. NesC fully supports tinyOS's concurrency model and WSN's characteristics with the introduction of a component-based structure and a set of new keywords, including *post, signal* and *call*. Each nesC application is a set of components linked together by interfaces. Typically, a component corresponds to a hardware device and a special component called Main initializes, starts hardware components and initializes global variables. Each interface defines commands and events used to communicate with the hardware. The component must implement

```
l1: command result_t StdControl.start(){
l2:    call Timer1.start(TIMER_REPEAT, TIMER_INTERVAL1);
l3:    call Timer2.start(TIMER_REPEAT, TIMER_INTERVAL2);
l4:    return SUCCESS;
l5: }
-------------------------------------------------
l7: event result_t Timer1.fired(){
l8:    if (numberOfRead==0) post task1();
l9:    else post task4();
l10:  return SUCCESS;
l11:}
-------------------------------------------------
l13:task void task1(){
l14:   ++numberOfRead;
l15:   uint8_t readrate=1/numberOfRead;
l16:   if (readrate>=MAX_RATE) post task2();
l17:   else post task3();
l18:}
-------------------------------------------------
l19:event retult_t Timer2.fired(){
l20:   uint8_t readrate2=1/numberOfRead;
l21:   post task3();
l22:   post task4();
l23:}
```

Figure 2: Code of the Example Application

all commands of each provided interface, all events handlers of each used interface, and any necessary tasks. **Command** is the traditional type of function called by the *call* keyword, an **event handler** is triggered by either an interrupt or the *signal* keyword and a **task** is placed on the task queue as indicated by the *post* keyword.

In this paper, we refer to tasks, commands and event handlers and other regular functions as **task posting units**. Figure 1 illustrates detailed interactions between different types of task posting units. As implied by the name, any task posting unit can post tasks to the task queue. Moreover, they can also call commands or functions and signal software events or secondary events. A hardware event handler is the only type of task posting unit that is able to interrupt tasks and other task posting units. Hardware event handlers usually belong to one of the tinyOS's system components; hence, secondary events are considered as hardware event handlers with a limited number of interrupting points. More information of interrupting points and events is presented later in Section 4.3

Figure 2 displays a portion of a WSN application that performs a calculating task at two different frequencies and toggles a set of leds when the calculation finishes. Hence, the program employs two different instances of the *TimerC* component to communicate with the mote's timers and a *LedC* component to control the leds via corresponding interfaces. The application's *Main* component starts the two timers. The application code consists of the implementation of each command of the initialization and the two firing events of the two timers. When each timer fires, the application chooses to perform some tasks (*task1, task2, task3, task4*), depending on the mote's status. Figure 2 illustrates the main part of application code: a command (*StdControl.start*), a task (*task1*) and 2 event handlers (*Timer1.fired, Timer2.fired*). The example will be used in later sections as we describe the representations.

3. CHALLENGES

Unique characteristics of tinyOS, nesC and WSNs lead to challenges in developing a representation useful for testing that can successfully express the ordering relationships in which tasks and events are executed in a mote application.

First, the concurrency model of tinyOS extends the traditional control flow. Tasks add a new type of control flow which are deferred function calls. Traditionally, when a program performs a function call, the execution immediately jumps to the entry of the function. After the function completes, control is returned back to the calling routine. Hence, calling statements explicitly define the execution order of the callee and caller. However, the time at which a task is posted is not an indicator of when the task will actually be executed; it only executes when it is at the front of the queue. Therefore, the comparative ordering of a posted task and a posting routine is different than the traditional ordering between a caller and a callee and needs to be represented differently.

Furthermore, tasks can change the behavior of events. If an event is trigged by an interrupt, the event handler obtains control of the system and starts executing. After the event handler completes, it may leave some unfinished tasks in the task queue. We cannot entirely analyze the impact of an event on the application without thoroughly investigating these tasks. Moreover, because events can occur at any time during an application's lifetime, an event can happen when tasks posted by other events have not been executed. Tasks from the new event interact with tasks of existing events that have yet to complete.

WSN applications typically need to comply with some timing restrictions, usually soft real-time requirements. Thus, an appropriate representation needs to cover program execution caused by timing interrupts. As some events can repeat, the representation should be able to model repeating events and their frequency.

4. REPRESENTATION

As discussed in Section 2, a WSN application is characterized by a set of events and tasks. An application's behavior continues to change in response to the appearance of events. An event's effects of the application depend on both the code of the event and the time at which the event occurs. Hence, to understand the behavior of a particular application, we need a representation that describes (a) the time of events' anticipation and (b) the order in which a certain event directly or indirectly posts its tasks to the task queue. Our application graph (G_{APP}) integrates the information of (a) and (b). Analyzing a path of the G_{APP}, we can obtain a sequence of event handlers and tasks in a specific ordering of their appearance during the application's lifetime. Combining this sequence with the CFG of each task posting unit, an execution path of the application can be determined that is useful for testing purposes.

To develop a G_{APP}, we need to collect the following information: (i) the order in which a task posting unit directly posts tasks; (ii) the set of tasks posted directly or indirectly by a certain event and the order of posting, and (iii) the point that each event is anticipated during the application's execution. Hence, we develop G_{APP} by first developing graphs that express individual tasks and events: (1) Posting graphs (G_{POST}) describe all tasks directly posted by a task posting unit and (2) Event graphs (G_{EVENT}) express the posting of tasks that originate in an event without considering any interactions from other events. The G_{APP} then incorporates the set of events and tasks with events' anticipating

information.

Before describing the details of each graph, we introduce some related definitions. First, we define the relationships among events, tasks and other task posting units in Definition 1 based on the activation method used to access a task posting unit. Then, Definition 2 defines an interrupting point.

Definition 1. Three types of relationships that can occur between a pair of task posting unit in a NesC application are:

- **Called in**: the **"called in"** relationship refers to a relation between a caller and callee in a function or command call.
- **Posted after**: A task posting unit u_j is **"posted after"** task posting unit u_i if u_j is a task and u_i posts the task to the task queue at some point. If the task queue is free, the entry point of u_j will immediately follow the exit point of u_i in an execution path; otherwise the task executes when it is at the front of the queue. In our graphs, both "called in" and "posted after" relations correspond to an edge.
- **Anticipated in**: the **"anticipated in"** relationship implies a relation between a task posting unit and an event handler. The event handler e_j is **anticipated in** an unit u_i if (a) the hardware event e_j can occur while the system is executing u_i or (b) a *signal* statement in the code of u_i explicitly activates e_j. As a result, the entry point of e_j's CFG might immediately follow any valid interrupting points of u_i and e_j stops executing before returning back to u_i. Each **"anticipated in"** relation is represented by an anticipating edge in our G_{App} and is characterized with a set of valid interrupting points.

Definition 2. In general, an *interrupting point* of a task posting unit is a location within the task posting unit's CFG at which the task posting unit can be interrupted by an arbitrary hardware interrupt. Usually, it is a statement outside any *atomic* region.

Each task posting unit has a set of interrupting points; however, not every interrupting point of the unit is valid for a specific event. The set of valid interrupting points accompanying a pair of $< unit, event >$ depends on the constraints of the events and types of events. More information about events' prerequisites is investigated in Section 4.3.

4.1 Task Posting Graph

The basic step in our representation's construction is to determine the order in which tasks are directly posted. A task posting graph (G_{Post}), defined in definition 3, describes the direct relationship between a particular task posting unit to other task posting units. A G_{Post} also reveals the relative orders of tasks posted by the same task posting unit.

Definition 3. A **task posting graph** for a task posting unit u is a two-tuple $G_{\text{Post}}(u)=< N_u, E_u >$, where N_u is a set of nodes and E_u is a set of directed edges. Each node $n_i \in N_u$ is a location of a *call* or *post* statement in u's CFG and is named by the target of this *call* or *post* statement. There is an edge $e_{<n_i,n_j>}$ in E_u if n_i is accessed before n_j in a path of u's CFG. Two special nodes n_{entry} and n_{exit} are the starting and the exiting points of G_{Post}.

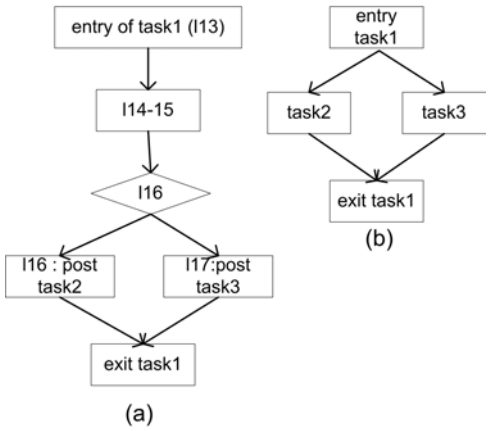

Figure 3: (a) CFG and (b) G_{Post} for *task1*

A G_{Post} is obtained from the task posting unit's CFG by eliminating irrelevant blocks, which are blocks without any *call* or *post* statements or *call* or *post* blocks to targets located in standard libraries or system components. A G_{Post} is **empty** if relevant blocks do not exist. Figure 3 illustrates the CFG of *task1* in Figure 2 and its corresponding G_{Post}. Only two posting blocks of *task2* and *task3* are in the G_{Post}. The G_{Post} specifies that *task1* posts either *task2* or *task3* in an execution path and there is no relationship between *task2* or *task3*.

4.2 Event Posting Graph

WSN applications interact with the outside environment through events. When an event occurs, its event handler is executed and the event is **activated**. However, some tasks posted directly or indirectly from the event handler would execute after the event handler exits. Therefore, an event is not **deactivated** when the event handler terminates but only when all posted tasks have finished. Hence, an event handler's G_{Post} is insufficient to analyzing the event's behavior. We need to employ another graph, Event Posting Graph (G_{Event}) as given in definition 4.

Definition 4. An **event posting graph** for an event e is a two-tuple $G_{\text{Event}}(e) = < N_u, E_u >$ where the set of nodes N_e is defined as:

$$N_e = n_{Activate} \bigcup n_{Deactivate} \bigcup_{i=1}^{i=n} N_{u_i}$$

Each u_i is a task directly or indirectly posted from e's handler and N_{u_i} is the set of nodes of $G_{\text{Post}}(u_i)$. $n_{Activate}$ and $n_{Deactivate}$ are the activation point and deactivation point of e. $n_{Activate}$ corresponds to e's handler. E_u is a set of directed edges. Each edge $e_{<n_j,n_k>}$ in E_e connects the two nodes if n_j is posted before n_k in a path of a $G_{\text{Post}}(u_i)$ or n_j is posted inside n_k.

Each path in a G_{Event} models a possible ordering of tasks occurring from event e's activation time to the deactivation time of e. If the task queue is free whenever e posts a task, that path is expanded into a code execution's path. In case of multi-events, tasks from other events can interleave with

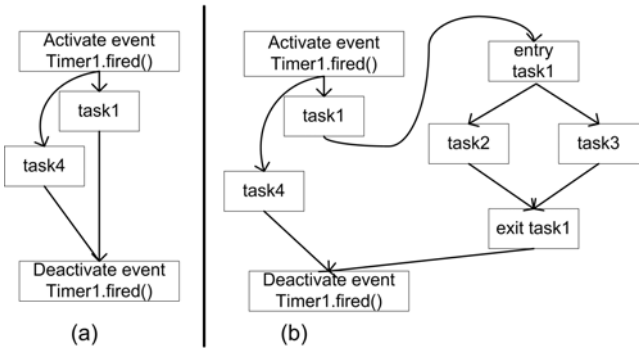

Figure 4: (a) G_{Post} of Timer1.fired()'s handler and (b) the complete G_{Event} after expanding *task1*

tasks of event e. These interactions are not covered by a G_{Event} but by a G_{App}.

To establish a G_{Event}, we first expand node $n_{Activate}$ as follows: (1) eliminate nodes associating with any *call* statements by inlining these G_{Post}s of the target to the G_{Post} of $n_{Activate}$ and (2) connect the G_{Post} of each posted task to the corresponding G_{Post} of e's handler. Then, the expansion process is repeated with each newly added nodes until all the newly added tasks have empty G_{Post}s.

Figure 4 depicts two steps in developing the complete G_{Event} for event *Timer1.fired()*. Starting with the G_{Post} of *Activate Timer1.fired()* depicted in Figure 4(a), we connect it to the G_{Post} of *task1* by (i)identifying the last task posted by the event before *task1* is executed and (2) inserting the G_{Post} corresponding to *task1* in between of the last task and *Deactivate Timer1.fired()*. As *task1* is posted in only one branch of the event handler, the other branch still connects to the *Deactivate* and is not effected by the G_{Post} of *task1*. Because *task2,task3,task4* do not post any tasks, their G_{Post}s are empty and are not analyzed.

If the task posting process contains a loop, the above connecting procedure becomes an infinite loop. Therefore, the number of orderings of paths may grow exponentially. To solve this problem, we apply k-limiting approach[7] to find a reasonable approximation of an G_{Event} with only k instances of each node appearing in a specific G_{Event}. With this approximation, each path in an G_{Event} presents either a complete or a partial execution order of tasks originated by the event.

4.3 Application Posting Graph

A WSN application can have multiple events that interact through task postings and interrupts. To understand relationships between events, we determine all potential anticipating points of events. The information is encoded in an Application Posting Graph or a G_{App}. Definition 5 below explains how the anticipating information is modeled and Definition 6 describes a G_{App}.

Definition 5. An **anticipating edge** connecting a task posting unit u_i and an event e_j is a three-tuple $AEdge_{<u_i,e_j>}$ $= < u_i, e_j, \text{IPOINTS}_{<u_i,e_j>}$, where e_j is **"anticipated in"** u_i, $\text{IPOINTS}_{<u_i,e_j>}$ is the set of valid interrupting points at which e_j can take controls from u_i.

Definition 6. An **application posting graph** for an application A is a three-tuple $G_{\text{App}} = <$ EVENTS_A, G-EVENTS_A, $\text{AEDGES}_A >$, where EVENTS_A is the set of events anticipated in A, G-EVENTS_A is the set of G_{Post}s of all events in EVENTS_A and AEDGES_A is a set of anticipating edges. Each application has three special events called e_{init}, e_{start} and e_{stop} for three standard commands *Main.StdControl.init()*, *Main.StdControl.start()* and *Main.StdControl.stop()* which are always executed at the beginning and at the end of an application.

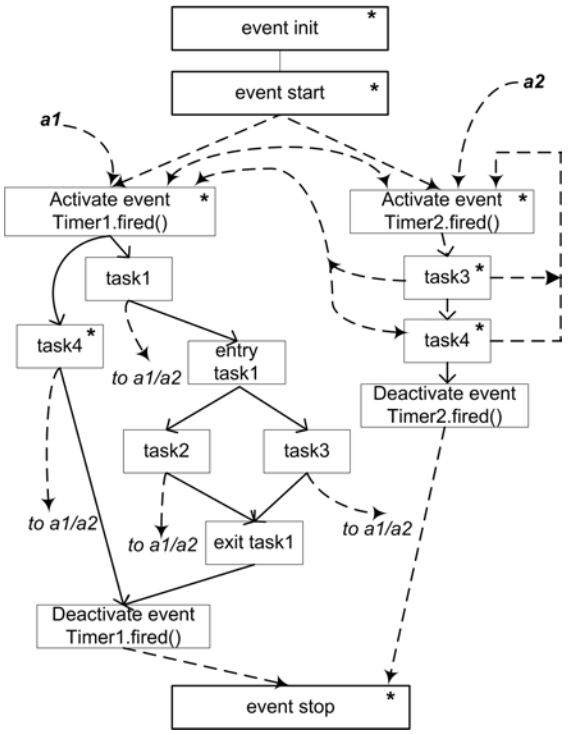

Figure 5: An Application Posting Graph for the example application

To generate anticipating edges, we detect **"anticipated in"** relations between task posting units. In other words, our algorithm classifies which event is expected after a certain statement. As mentioned in Section 2, an event is trigged by either a hardware interrupt or a *signal* statement so the event can occur if (a) the hardware is ready to send interrupts to the mote or (b) there is an explicit *signal* statement. Sometimes, the *signal* statement is located in a system component, and hence, it is not visible to the application code. The conditions of hardware readiness are opaque to application's code as well. As a result, we can only detect implicit anticipating information with the help of preset semantic rules found in the system interfaces' declaration.

We observed that each command in an interface may accompany an event from the same interface. For example, event *clock.fired()* can only appear after the corresponding *clock* is started and event *ADC.dataReady()* should follow the command *ADC.getData()*. Therefore, we scan through each interface in the system's code to establish semantic rules for each event. These semantic rules of a particular event are considered as **activating conditions** of that event. Each event may also have another set of semantic rules called **deactivating conditions**, after each of which the event is no longer expected.

After detecting an initial set of anticipating events for each node in G_{EVENT} of e_{init}, e_{start}, each anticipating event's G_{EVENT} is added to its source with an anticipating edge. The set of anticipating events for each node is transferred to its successors in G_{EVENT}. The above process is repeated until no more anticipating edges are detected.

Figure 5 illustrates the G_{APP} of the example application. As both timers are started within event *start* (line l1 and l2 in Figure 2), both event *Timer1.fired()* and *Timer2.fired()*) are anticipated in event *start*. Because timers' events are repeating, the anticipating information remains at each node in G_{EVENT}s of event *Timer1.fired()* and event *Timer2.fired()* until their deactivation conditions are fulfilled in event *stop*. Therefore, event *Timer2.fired()* is anticipated during the execution order of event *Timer1.fired()* and during itself.

The next step of refining anticipating edges is to establish a set of valid interrupting points (IPOINTS) for each each anticipating edge. An IPOINTS depends on the type of the incoming event. If the event is a hardware interrupt, the IPOINTS is calculated by analyzing atomic regions in the outgoing node's CFG. Otherwise, the event is signaled by a task in a system's component so it cannot actually interrupt code's execution. That event's IPOINTS contains only the entry and exit points of the outgoing node. For instance, in Figure 5, $\text{IPOINTS}_{<task1,Timer2.fired>}$ only consists of the entry and exit point of *task1* because *Timer.fired()* is implemented by a task.

A G_{APP} expresses all possible events in an application, including anticipated events that could occur during the execution path of other events and tasks. When an anticipated event is triggered, its task posting units will be interleaved with the execution chain of existing events. Hence, our representation framework provides a mechanism to expand that edge into a complete execution order for each interrupting point. To reduce the overhead of the analyzing process, the expansion is performed only on demand when the user needs to study a specific edge. Back to Figure 5, the expanded path associating with $AEdge_{<task1,Timer2.fired>}$ at the exit point of *task1* is *task1, Timer2.fired(), task2* or *task3* (posted by *task1*), *task3, task4* (from *Timer2.fired()*).

5. DISCUSSION

We have completed the implementation of the above representations. The front end of the representation builder employs the parser and compiler of TinyDT [8], an Eclipse plug-in for WSN development. From abstract syntax trees obtained from TinyDT, our tool constructs the CFGs and three graphs described above. Preliminary experimental results on the overhead have been produced and are promising. We are also developing an analysis to detect patterns in the configuration code of the application using our graphs to detect seeded errors.

As our representations provide detailed information about the application and its environment, they enable testing a WSN system within its environment in a number of ways.

For instance, a G_{APP} describes all possible events in the application, including anticipated events that could occur during the execution path of other events and tasks. Analyzing anticipating edges in the G_{APP} and their sets of interrupting points can determine a scenario demonstrating interactions between the application and the environment. Similarly, a G_{EVENT} contains ordering information of task posting units within a stand-alone event and analyzing the G_{EVENT} can guarantee code coverage of each event. Therefore, using these representations can help frame code coverage criteria at different levels and develop the analysis for determining the coverage needed. Possible criteria for WSNs are based on the GUI event coverage criteria [9] and propose all-event coverage, all static-path coverage as well as all possible sequence of events coverage. The all-event criterion ensures that each event is executed at least once during the testing. The all-static path criterion expresses that each path in the corresponding G_{EVENT} will be executed at least once. The sequence of events criterion is fulfilled when a particular sequence of events in the G_{APP} is executed. Similar to the approach in [9], we argue that instead of checking unlimited length sequences, shorter sequences are sufficient to detect errors in WSNs as is the case with GUIs.

Furthermore, a complete order of event handlers and tasks in an application can be integrated through their CFGs to form an inter-event CFG. The obtained inter-event CFG contains concrete program execution paths that can facilitate structural testing and static analysis technique. A test case can be developed by specifying a set of inputs leading to a given execution path in the inter-procedural CFG. For example, the path denoted by **(*)** in Figure 5, which consist of *init*, *start*, *Timer1.fired()*, *Timer2.fired()*, *task4*, *task3, task4*, *stop* in that order, leads to the inter-event CFG with the following execution path $\{< l1 - l5 >, l8, < l10 - l11 >, < l20 - l22 >\}$ (code is given in Figure 2). The path triggers the divided-by-zero exception in line l20 of the code and the test case associating with that path would reveal an existing bug of the program: variable *numberOfRead* has not been increased before variable *readrate2* is calculated. Therefore, when used together with CFGs, G_{EVENT} and G_{APP} can generate execution paths which lead to test cases and test oracles.

Beside testing, our graphs can also be used for other static analysis techniques. Being able to trace the impact of each *post* statement in the application within and beyond the border of functions, our G_{EVENT} and G_{POST} can detect call patterns and configuration errors as well [14].

6. RELATED WORK

There have been a few research efforts to provide an intermediate representation for analyzing WSNs' applications for various purposes. One approach uses different types of automata to model a WSN application such as hybrid automata [4] and parallel interpreted automata [12]. Although this approach succeeds for simulation or model checking techniques such as life time analysis, automata are not fine-grained enough to determine the behavior of each individual statement. Therefore, automata cannot be used for static analysis needed in testing. Another approach [2] discusses that recording every state of every single mote in the network because the number of nodes is tremendous. To solve this problem, Agha [2] proposes using statistic values as values for properties of a system. Again, this representation loses details of each individual mote; hence this representation cannot help static analysis which is the focus of our representation framework.

7. REFERENCES

[1] Tinyos homepage. http://www.tinyOS.net.

[2] G. Agha. Computational models, programming languages and algorithms for sensor networks: Towards a paradigm shift in computer science. *sutc*, 1:2, 2006.

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, March 2002.

[4] S. Coleri, M. Ergen, and T. J. Koo. Lifetime analysis of a sensor network with hybrid automata modelling. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 98–104, New York, NY, USA, 2002. ACM Press.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[6] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 201–213, New York, NY, USA, 2004. ACM Press.

[7] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[8] W. P. McCartney and N. Sridhar. Tosdev: a rapid development environment for tinyos. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 387–388, New York, NY, USA, 2006. ACM Press.

[9] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 256–267, New York, NY, USA, 2001. ACM Press.

[10] S. Park, A. Savvides, and M. B. Srivastava. Sensorsim: a simulation framework for sensor networks. In *MSWIM '00: Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 104–111, New York, NY, USA, 2000.

[11] J. Regehr. Random testing of interrupt-driven software. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded Software*, pages 290–298, New York, NY, USA, 2005. ACM Press.

[12] L. Samper, F. Maraninchi, L. Mounier, and L. Mandel. Glonemo: global and accurate formal models for the analysis of ad-hoc sensor networks. In *InterSense '06: Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, page 3, New York, NY, USA, 2006. ACM Press.

[13] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the second international conference on Virtual Execution Environments*, pages 78–88, New York, NY, USA, 2006. ACM Press.

[14] N. Zhang. Fault localization in NesC program by semantic analysis. Technical report, University of Virginia, 2007.