T-Morph: Revealing Buggy Behaviors of TinyOS Applications via Rule Mining and Visualization

Yangfan Zhou[*][†] Xinyu Chen[*] Michael R. Lyu[†][‡] Jiangchuan Liu[§]
[*]Shenzhen Research Institute, The Chinese Univ. of Hong Kong, Shenzhen, China
[†]Dept. of Computer Sci. & Eng., The Chinese Univ. of Hong Kong, Hong Kong, China
[‡]School of Computers, National Univ. of Defense Technology, Changsha, China
[§]School of Computing Science, Simon Fraser Univ., Burnaby, BC, Canada

ABSTRACT

TinyOS applications for Wireless Sensor Networks (WSNs) typically run in a complicated concurrency model. It is difficult for developers to precisely predict the dynamic execution process of a TinyOS application by its static source codes. Such a conceptual gap frequently incurs software bugs, due to unexpected system behaviors caused by unknown execution patterns. This paper presents `T-Morph` (`TinyOS application tomography`), a novel tool to mine, visualize, and verify the execution patterns of TinyOS applications. `T-Morph` abstracts the dynamic execution process of a TinyOS application into simple, structured application behavior models, which well reflect how the static source codes are executed. Furthermore, `T-Morph` visualizes them in a user-friendly manner. Therefore, WSN developers can readily see if their source codes run as intended by simply verifying the correctness of the models. Finally, the verified models allow `T-Morph` to automatically check the application behaviors during a long-term testing execution. The suggested model violations can unveil potential bugs and direct developers to suspicious locations in the source codes. We have implemented `T-Morph` and applied it to verify a series of representative real-life TinyOS applications and find several bugs, including a new bug in the latest release of TinyOS. It shows `T-Morph` can provide substantial help to verify TinyOS applications.

Categories and Subject Descriptors

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

Keywords

TinyOS, Dynamic Analysis, Sensor Networks

1. INTRODUCTION

For over a decade, Wireless Sensor Networks (WSNs) [11] have yet to be realized as a practical, prevalent bridge between the physical world and the cyberspace. One of the major obstacles to extensively deploy such networks is their low reliability [17]. Various software bugs have frequently brought troubles and even failures to existing WSN applications [2, 17, 31]. Towards the wide applications of WSNs, how to effectively eliminate software bugs is still a critical concern to WSN developers [17] as well as potential industrial customers [23]. Unfortunately, we are still lacking convenient tools to fight against WSN software bugs.

TinyOS [29] is a prevailing operating system for WSNs, which provides a typical *event-driven* programming paradigm for network-embedded sensors. In TinyOS applications, an *event procedure* (*i.e.*, event-handling logic) is triggered by its corresponding event (*i.e.*, a hardware interrupt). Events may occur at any time [19, 29]. This consequently introduces complicated interleaving executions of source codes. Such interleaving executions can make the *dynamic* execution process of a TinyOS application difficult to be precisely predicted based on the *static* source codes. The nondeterministic execution patterns of the TinyOS codes may cause unexpected system behaviors, leading to system failures. Software bugs are therefore inevitable.

Testing is an important means to cope with software bugs. It is well-known that a long-term testing run is necessary to trigger potential TinyOS bugs [16, 24]. Unfortunately, we lack an automatic mechanism to verify the pass/fail of such a testing run. Manual inspection of the tremendous execution trace generated in the testing run is, however, extremely labor-intensive, not to mention correlating a bug with a source code defect [35].

Mining a behavior model (*i.e.*, a specification) from program execution traces sheds light on manual verifications. However, existing techniques (*e.g.*, [1, 21, 22]) generally focus on mining sequential programs, where functions call one another in a sequential manner. Mining the interleaving executions of general concurrent programs remains difficult [15], for the number of possible interleaving scenarios is often huge. These pose a great challenge to employ behavior mining approaches in verifying TinyOS applications.

We find that unlike general concurrent programs [3, 10], TinyOS applications possess more tractable interleaving scenarios and execution processes, due to its specific programming paradigm. The concurrent execution of TinyOS applications can be narrowed down into sequential event procedures together with their interleaving models. Specifically, we observe that by tracking the interrupts and two system calls, the complicated execution process can be anatomized into a set of event procedures. More importantly, TinyOS applications are implemented in a job-oriented manner: A developer writes a segment of source codes for a specific job. TinyOS implements the codes as many event procedures of different types, and executes them in sequence to jointly accomplish the intended job. Driven by recurrent events (e.g., timer timeouts or packet arrivals), the intended jobs are performed iteratively, which results in a repetition of a series of event procedures in the execution process. This allows us to specifically tailor a set of rule mining algorithms to abstract the execution process into simple, structured application behavior models. Such semantic models closely capture how the static source codes are executed, and thus can be employed to verify the applications.

Hence, this paper proposes `T-Morph` (`T`iny`O`S application `to`mo`graph`y), a novel tool to mine, visualize, and verify the execution patterns of TinyOS applications. `T-Morph` takes a two-step approach. First, it executes a baseline testing run, which produces the initial behavior models. They are visualized in a user-friendly manner. Developers can readily see if their source codes run as intended by simply checking the correctness of the models. Then the *verified* models reinforce the knowledge of `T-Morph` to the target application. This allows `T-Morph` to automatically troubleshoot the application in a long-term testing run by checking its behaviors against the verified models. The violation of the models can unveil potential bugs to the developers and direct them to suspicious locations in the source codes. This can greatly reduce human efforts in troubleshooting TinyOS applications.

We have implemented `T-Morph` and successfully applied it to identify bugs in several representative real-world TinyOS applications covering typical usage of all WSN interrupts. These bugs, caused by complicated execution scenarios, can hardly be discovered by merely examining the source codes. This shows `T-Morph` can bridge the conceptual gap between the source codes and their execution process. Consequently, bugs caused by such a gap can be effectively eliminated.

The rest of the paper is organized as follows. Section 2 presents the related work. We discuss the features of WSN applications and the design considerations of `T-Morph` in Section 3. Section 4 elaborates how `T-Morph` mines the behavior models in the execution process. We show how these models can help troubleshoot WSN applications in Section 5. Three case studies are discussed in Section 6. Sections 7 and 8 provide further discussions and conclude this paper.

2. RELATED WORK

Recent publications frequently reported that various software bugs have brought failures to field-deployed WSNs, posing a major barrier to their extensive applications [17][31]. To make WSNs more reliable, many troubleshooting tools, debugging supports, bug-preventing approaches, and testing methods have been proposed.

Dustminer [12] and Sentomist [35] are two recent troubleshooting approaches to identify bugs in WSN applications. Relying on a function-level logging engine, Dustminer checks the discriminative patterns from a function call sequence that causes some bad behaviors. It focuses on simple, sequential executions of the target applications. In contrast, `T-Morph` tackles their complicated interleaving executions, which are generally involved due to the concurrency model of WSN applications. Aiming to unveil transient bugs, Sentomist [35] anatomizes a long term execution process into event-procedure instances. It finds outliers among them and considers them as bug symptoms. Sentomist can identify when a transient bug manifests. But locating source code defects still needs tedious manual efforts by understanding the execution process represented in the instructions, *i.e.*, the machine codes. `T-Morph`, on the other hand, aims at helping WSN developers link the bug symptoms shown in the execution process to the source codes. Moreover, unlike Sentomist, `T-Morph` does not focus on transient bugs only.

Existing debugging tools (*e.g.*, Marionette [32], Clairvoyant [34], MDB [27]) can provide interactive remote debugging interfaces for sensor nodes. These tools can to some extent let WSN developers know how their source codes are executed. However, inserting debugging activities (*e.g.*, enabling breakpoints) generally affects the execution process of the target application. Moreover, adding declarative tracepoints [4] has also been suggested for extracting program runtime information after observing abnormal behaviors. Nevertheless, mapping the runtime information to source code defects still depends heavily on manual efforts.

Safe TinyOS [6] and Neutron [5] automatically enforce run-time memory safety for TinyOS applications. NodeMD extends compilers by inserting checking codes [14]. T-Check [20] and KleeNet [26] find bugs by exploring program states extensively via simulations. All these preventive tools focus only on certain types of bugs (*e.g.*, memory access violation). They are still not adequate to eliminate faulty system behaviors caused by complicated interleaving executions due to improper design.

Testing is an important means to verify the correctness of an application. Unfortunately, conventional software testing tools for testing sequential programs (*e.g.*, [8]) and concurrent programs (*e.g.*, [18, 33]) are inadequate in fighting against bugs in WSN applications due to their specific concurrency model. Regehr [24] is the first to point out that testing a WSN application requires to schedule a lot of random artificial interrupts, so as to allow the application to explore more execution scenarios. Lai *et al.* [16] study the test adequacy criteria for TinyOS applications. In general, testing a TinyOS application will produce a long-term execution [16]. Human inspection of the testing results is labor-intensive. There is no handy tools for TinyOS developers to understand such a long term execution process, to locate errors, and to correlate them with the source code defects.

Specification mining approaches (*e.g.*, [1, 21, 22]) can infer behavior models from program execution traces. In addition, PIP [25] abstracts the behaviors of distributed systems and detects the unexpected. These approaches shed light on the design of `T-Morph`. `T-Morph` moves a step further, and focuses on mining the concurrent executions of TinyOS applications, instead of sequential programs.

Kothari *et al.* [13] propose to abstract WSN application source codes into finite state machines via static analysis. But no work aims at visualizing the dynamic execution process of a WSN application, which is however crucial for WSN developers to verify the correctness of the application. `T-Morph` closes this gap by properly abstracting the execution process and visualizing the models.

3. TINYOS APPLICATION SPECIFICS

This section provides some preliminaries and our key observations of how TinyOS applications are executed on a typically resource-constrained hardware platform. We will first define event procedures (the building blocks of the execution process of a TinyOS application) and their concurrency models. Finally we will show how a job intended by developers is typically completed in runtime. These are the basics for `T-Morph` to model TinyOS application behaviors.

3.1 Event procedures of TinyOS applications

TinyOS applications typically take an event-driven programming paradigm. An event is essentially indicated by a hardware interrupt, *e.g.*, one denoting a timer timeout. When an event occurs, the microcontroller unit (MCU) will automatically invoke its corresponding handler, *i.e.*, certain application logic to process the event [29]. We call such application logic an *event procedure*.

Usually in TinyOS applications, event procedures with different functionalities may share the same entry due to hardware design. In other words, different types of event procedures may be triggered by the same hardware interrupt, while different application logic will subsequently be involved according to complicated system state information. For example, all operations on the wireless interface chip (*e.g.*, CC1000 for Crossbow Mica2) are triggered by Serial Peripheral Interface (SPI) interrupt [29]. There are, surprisingly, tens of different functionalities (*i.e.*, tens of event procedure types) that may be triggered by the SPI interrupt for a typical TinyOS application, including those involved in sending a data packet (*e.g.*, transmitting the packet data, receiving the corresponding acknowledgement) and those involved in receiving a data packet.

Such "all-in-one" usage of a single hardware interrupt makes it quite difficult to automatically distinguish among different types of event procedures even when a particular hardware interrupt is captured, posing a challenge for `T-Morph` to model their functionalities.

3.2 TinyOS application concurrency model

An event procedure may not complete shortly. Hence, to avoid the monopolization of the MCU resource, TinyOS [29] typically implements an event procedure as two separated parts: an asynchronous *interrupt handler*, which is immediately invoked when an event occurs, and some deferred synchronous procedure calls, namely, *tasks*. Tasks are posted in a global queue by interrupt handlers or other tasks and will be executed in a first-in-first-out (FIFO) manner. Moreover, an interrupt handler may preempt a task or even another interrupt handler (*e.g.*, for nodes with Atmel ATmega128L MCUs such as MicaZ).

Such a concurrency model, though efficient for resource-constrained WSNs, causes complicated executions of applications. As events may arrive at any time, the asynchronous part of an event procedure (*i.e.*, the interrupt handler) may not start with a deterministic program context. It may preempt another event-procedure instance. Moreover, event-procedure instances may also post tasks in an interleaving manner, leading to their own interleaving executions.

Figure 1 shows an example of how two interleaving event-procedure instances may run. Event-procedure instance 1 starts with its corresponding interrupt handler I_1, which defers some application logic by posting a task T_{11} during execution. Later, task T_{11} will be executed, during which a new event arrives. Its corresponding interrupt handler I_2 preempts T_{11}, which starts event-procedure instance 2. I_2 also posts a task T_{21} during execution. After I_2 ends, T_{11} resumes its execution. It posts another task T_{12} and exits. Now the task queue contains T_{21} and T_{12}. Since

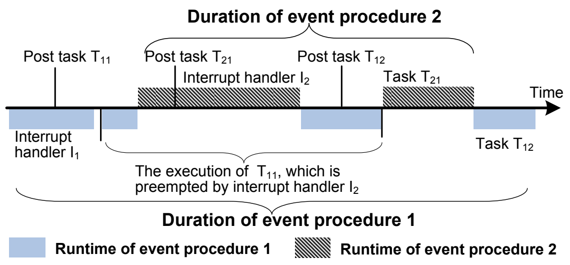

Duration of event procedure 2
Post task T11
Post task T21
Post task T12
Interrupt handler I2
Task T21
Time
Interrupt handler I1
Task T12
The execution of T11, which is preempted by interrupt handler I2
Duration of event procedure 1
Runtime of event procedure 1
Runtime of event procedure 2

Figure 1: Interleaving executions of two event-procedure instances.

tasks are scheduled in a FIFO manner, T_{21} will be executed, followed by T_{12}. We can see that the preempting interrupt handler and the interleaving tasks cause the executions of the two event-procedure instances overlap in a complicated manner. Furthermore, such interleaving execution patterns are generally not deterministic during the coding phase. For example, task T_{11} in another instance of event procedure 1 may not be interrupted by I_2.

The concurrency model for supporting multitasking makes it prohibitively hard for WSN developers to understand the complicated interleaving executions of their source codes. Tackling this difficult concurrency model is one of the objectives of `T-Morph`.

3.3 Job flows in execution process

An event procedure reflects what a TinyOS application will do when a particular event occurs. Dividing the system runtime into event-procedure instances and properly modeling their behaviors can help unveil the executions of source codes. However, this is still far from enough.

We observe that WSN developers generally write source codes in a job-oriented manner: A segment of source codes is designed for a specific job at first. Then after the codes are compiled, the operating system may implement them as many event procedures of different types, and execute them in sequence to jointly accomplish the intended job. Examining each one individually (*e.g.*, as proposed in [35]) may cause the developers to lose critical sequential information to verify the correct execution of their codes. One important notion of `T-Morph` is that the job-specific patterns of event procedures are crucial for the developers to understand the dynamic execution process of their static source codes.

We show this via an example in Figure 2(a). It provides the nesC [9] source codes of function `Read.readDone` adapted from the `Oscilloscope` application [29], which allows a node to obtain, cache, and send its sensor readings. The function is invoked by an ADC (Analog-to-Digital Converter) interrupt handler triggered by an event indicating a sensor reading is ready on the ADC chip. By writing the codes, the developers intend to read and cache every three sensor readings, and then send them in one packet. The codes look simple. However, a bug may be manifested here: If a new ADC interrupt arrives with a new sensor reading before the data of the previous packet (`packet->content`) has been sent, it will invoke `Read.readDone` again. As a result, the packet content is altered by this new sensor reading (line 5).

There are three types of event procedures (denoted by A, B, and C) involved when the codes are executed, shown in Figure 2(b). A obtains and saves a sensor reading; B obtains a reading, puts it together with two previous readings, and

```
...
// The Read.readDone() function is invoked by the
// interrupt handler for the ADC data-ready interrupt.
event void Read.readDone(error_t err, uint16_t reading)
{
    packet->content[index] = reading;
    index ++;
    // After 3 sensor readings have been collected,
    // post a task which starts sending the readings.
    if(index == 3)
    {
        index == 0;
        post sendDataPacket();
    }
    return;
}
...
```

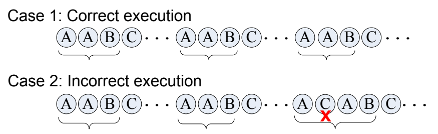
(a) nesC source codes of a buggy function in handling sensor readings

Ⓐ Obtain the reading and cache it

Ⓑ Obtain the reading, cache it, and initialize the sending of the cached data

Ⓒ Perform the steps for sending a data packet

(b) Three types of involved event procedures


Case 1: Correct execution
A A B C · · · A A B C · · · A A B C · · ·
Case 2: Incorrect execution
A A B C · · · A A B C · · · A C A B C · · ·

(c) Event-procedure sequences of two executions

Figure 2: An example: a buggy function in an ADC event procedure.

triggers a packet sending; C represents the process of sending the data packet.[1] A and B are triggered by the ADC interrupt. They are directly resulted from the source codes. If examined individually, the logic of each event procedure appears to comply with the intention of the developers.

Let us further check the execution sequence of the event procedures, as shown in Figure 2(c). The pattern AAB will immediately shows this is the intended job of the source codes: Read and cache the first two sensor readings (AA), and after reading and caching the third one, initialize a packet sending process to send the three cached readings (B). If the pattern AAB always appears iteratively as in case 1 in Figure 2(c), the developers can verify that the codes run correctly in the execution process. But when the bug manifests, the resulting sequence may be case 2 in Figure 2(c). The arrival of a new sensor reading triggers A before the sending process of the previous packet is done. Hence, the node will go on sending the previous packet after A is done, resulting in a C lying in between two A's. Obviously this is not the intention of the source codes. Thus, the developers can quickly locate the defect of the source codes.

We can see from the above example that it will be critical to map the execution process into the jobs that the developers design their codes to accomplish. This can greatly help the developers understand the execution process of their source codes, so that they can consequently verify the correctness of their source codes and find potential defects. We name a sequence of the event procedures that jointly accomplish a job intended by the developers a *job flow*. The pattern AAB in the above example represents a job flow.

Job flows are obviously application-specific. Also, job flow instances can interleave with each other in a complicated manner as well. These are the major challenges faced by

[1]As we discussed in Section 3.1, sending a data packet will involve many event procedures of different types. For the convenience of our discussions, let C denote the event procedures triggered by the SPI interrupt.

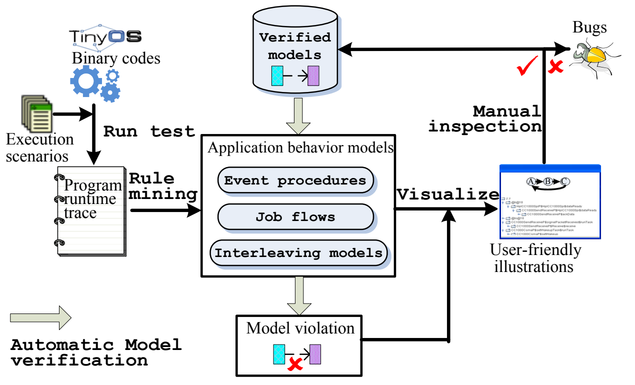


Figure 3: Overview of the T-Morph framework.

T-Morph to identify and model job flows automatically, so as to clearly describe to developers what a job is intended.

4. MINING TINYOS EXECUTION PROCESS

The key notion for T-Morph to bridge the conceptual gap between the source codes and their complicated execution process is that the system runtime should be mapped into job flows of event procedures, reflecting the actual design purpose of the source codes. To this end, T-Morph should identify, model, and present a job flow in a generic way convenient for the developers to verify its functionality. Figure 3 provides an overview of the framework of T-Morph. Centric to this framework is the rule mining approach that abstracts the application behavior models, *i.e.*, the models of the job flows *per se* and how they are interleaved, together with the models of their building blocks, *i.e.*, event procedures.

T-Morph first identifies event-procedure instances from the execution process of a TinyOS application. By properly featuring these instances, it classifies them into different types, where those in the same type bear similar functionalities. The program runtime can hence be abstracted into a sequence of event-procedure models of different types (*e.g.*, A, B, and C in the example in Figure 2). T-Morph then extracts the patterns of the models out of the sequence (*e.g.*, AAB in Figure 2), and forms the job flows of interest.

Finally, the resulting behavior models will be visualized for the developers in a graphic manner. This user-friendly illustration can allow the developers to verify whether the execution process of their source codes follows their intentions. If *yes*, the models can be saved for further verification of the TinyOS application. Otherwise, the detected inconsistency, whose cause is close-associated with the source codes, can substantially help the developers locate potential defects in the source codes.

4.1 Analyzing the execution process

Since a job flow is essentially a sequence of event procedures, T-Morph first analyzes the program runtime and finds the corresponding building blocks of job flows. As discussed in Section 3.2, the concurrency model of TinyOS applications causes complicated interleaving executions of event procedures. Fortunately, we have found that by tracking the interrupts and the task posting/executing system calls (*i.e.*, the postTask and runTask functions), we can obtain the runtime of each event-procedure instance, *i.e.*, the runtime of its corresponding interrupt handler and the runtime of all the tasks that belong to event-procedure instances (see Figure 1 for some examples).

To analyze the program runtime, we first summarize the execution rules of TinyOS applications based on the concurrence models of TinyOS [35]:

- Rule 1: Interrupt handlers and tasks all run to completion unless preempted by other interrupt handlers;
- Rule 2: Tasks are scheduled in a queue by interrupt handlers or other tasks with postTask system call, and are executed in a FIFO manner with runTask system call. Hence, the task scheduled via the ith postTask is executed via the ith runTask in the execution process.

Based on these rules, we can obtain the runtime of an interrupt handler and that of a task as follows.

Runtime of an interrupt handler: For an interrupt n, its interrupt handler starts from its entry int(n) and ends at its exit reti. Since int(n) and reti are nested, we can then locate the corresponding int(n) and reti for each interrupt handler. Based on Rule 1, we know the runtime of an interrupt handler is the time between its entry int(n) and its exit reti, excluding the time during the preemptive executions of other interrupt handlers, if any.

Runtime of a task: Tasks are executed by invoking the runTask system call. Hence, based again on Rule 1, the runtime of a task is the time between the invocation of its corresponding runTask and its return, excluding the preemptive executions of interrupt handlers, if any.

An interrupt handler starts a new event-procedure instance. Based on Rule 1, the tasks scheduled by the postTasks during the runtime of the interrupt handler are the tasks of the event-procedure instance. Based on Rule 2, T-Morph can then locate their runTasks. Thus, the runtime of these tasks can be obtained. Then again based on Rule 1, during the runtime of these tasks, if there are any new tasks scheduled by the postTasks, the new tasks also belong to the event-procedure instance. Similarly T-Morph can locate their corresponding runTasks and obtain their runtime. This progress continues iteratively till no new tasks are scheduled.

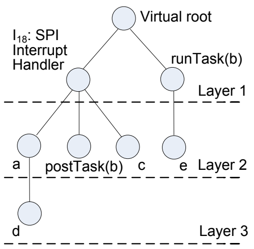
T-Morph thus gets the runtime of each event-procedure instance. It is then convenient to record the TinyOS application behaviors during the execution of each event procedure instance, which will be described next.

4.2 Featuring behaviors of event procedures

After analyzing the execution process with a set of event-procedure instances, the question now is what kind of runtime data can best feature the behaviors of each instance. As discussed in Section 3.1, event procedures of different types may usually share the same interrupt entry in typical TinyOS applications. Hence, we cannot simply represent the behaviors of an event-procedure instance based *only* on its triggering hardware interrupt. Another option is to feature the behaviors by recording the instructions (*i.e.*, binary machine codes) executed during the lifetime of the instance [35]. In this approach, the involved instructions can directly reflect the application behaviors. However, they are not user-friendly for the WSN developers: It is hard, if not impossible, for the developers to understand the execution of the machine codes.

We observe that TinyOS applications are generally not designed to perform complex data computation with many looping and branching control flows due to hardware limitation [29]. Hence, showing the involved function invocations

```
   ...
// code segment of the SPI
// interrupt handler
    a();
    post b(); //post a task
    c();
   ...
void a()
{
    d();
}
void b()
{   //called by runTask
    e();
}
// Functions c(), d(), and e()
// do not call other functions
   ...
```



The SPI interrupt triggers the execution of the following functions in sequence: a, d, postTask(b), c, runTask(b), and e. Therefore, the layered sequence is I$_{18}$:1, a:2, d:3, postTask(b):2, c:2, runTask (b):1, and e:2.

Figure 4: An example layered function sequence.

would be enough to indicate the control flows of the instance, and thus well captures its behaviors. This is also noticed by some existing approaches (*e.g.*, [12]). However, they simply consider to record a sequence of all function calls, which inevitably loses important information of function invocation relations. Moreover, such approaches result in a long function sequence including many functions that may not be of the developers' interest. They introduce not only huge noise for human inspection, but also much unnecessary complexity for automatic algorithms (*e.g.*, [12]) to analyze the sequence efficiently. In contrast, T-Morph solves these deficiencies by employing *layered function sequence*, instead of traditional function sequence, to represent the application behaviors.

The item set in a layered function sequence contains the functions called within the event-procedure instance, as well as the interrupt entries and the task posting/executing functions. We build an *invocation tree* describing the invocation relations of the items for easy illustration of layered function sequence. The tree has a virtual root (layer 0). Interrupt entries and tasks are the children of the root, since they are called by the operating system. They are hence in layer 1. If a function is called by a node in layer n, it is a child of the node and hence in layer $n+1$. By decorating each node of the tree with its name (the name of a function, interrupt, or task) and its layer number, a layered function sequence is then the preorder traversal sequence of the tree.

Figure 4 shows an example of such a tree and its corresponding layered function sequence. Note that function entries and returns are nested in an execution process. By tracking function entries and function returns, it is easy to employ a pushdown automaton to construct the invocation tree and thus get the layered function sequence.

Encapsulating the function invocation relations, layered function sequence describes more accurately the behaviors of an event-procedure instance. It also equips T-Morph with a flexible way to filter function calls. Unlike traditional function sequence that implies all functions are of equal importance, layered function sequence encodes the hierarchy of the involved function calls. Due to the nature of invocation tree, a function in a lower layer provides more control flow information than that in a higher layer, and hence is more important to describe the coarse-grained functionality. This is the basic for T-Morph to model the functionalities of event-procedure instances and classify them into different types. The details will be elaborated in Section 4.3.

Moreover, layered function sequence is more suitable for

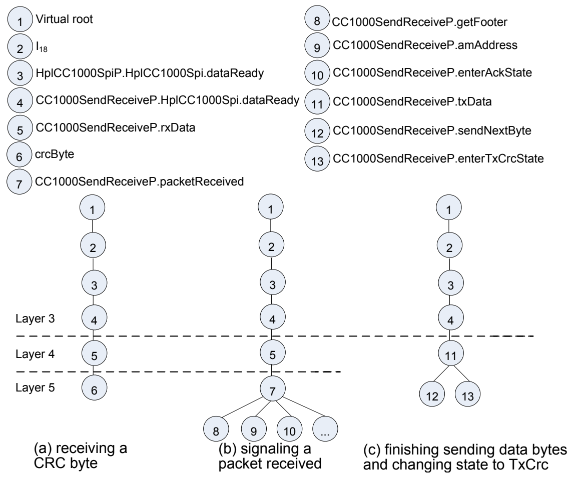

1 Virtual root
2 I18
3 HplCC1000SpiP.HplCC1000Spi.dataReady
4 CC1000SendReceiveP.HplCC1000Spi.dataReady
5 CC1000SendReceiveP.rxData
6 crcByte
7 CC1000SendReceiveP.packetReceived
8 CC1000SendReceiveP.getFooter
9 CC1000SendReceiveP.amAddress
10 CC1000SendReceiveP.enterAckState
11 CC1000SendReceiveP.txData
12 CC1000SendReceiveP.sendNextByte
13 CC1000SendReceiveP.enterTxCrcState
Layer 3
Layer 4
Layer 5
...
(a) receiving a CRC byte
(b) signaling a packet received
(c) finishing sending data bytes and changing state to TxCrc

Figure 5: Invocation trees of three example layered function sequences.

T-Morph to visualize the functionality of an event procedure. First, via source code analysis, a function name can be linked to its source codes, which provides a straightforward means to understanding the purpose of an event procedure. Second, the layered function sequence can be mapped to an invocation tree conveniently. This property is pretty desirable, since a tree-like demonstration of function invocation relations can greatly facilitate human reasoning. Finally, the layered function sequence offers a flexible way to visualize an instance. The developers can get the coarse-grained information by simply examining the involved functions in low layers. When more details are needed, the tree can be expanded to show functions in deeper layers.

4.3 Modeling event procedure functionalities

A typical TinyOS application may involve various event procedures of different functionalities during its runtime. For example, tens of event procedures types (in terms of the functions invoked) may run during a packet transmission process. This poses a challenge to analyzing an execution process into a small number of simple behavior models for human inspection. T-Morph solves it by filtering the functions in layered function sequences according to their layer numbers. We discuss it via a motivating example in Figure 5, where the invocation trees of three event-procedure instances triggered by the SPI interrupt are shown.

The first instance is for receiving a byte, the second for signaling a packet received, while the third for sending a data byte. Their functions in layer 3 are the same. By its source codes we can know the instances are all for packet transmissions. Based on their layer-4 functions, they can be classified into two types, *i.e.*, the left two instances for receiving packets (denoted by R) and the last for sending packets (denoted by S). By examining the layer-5 functions, we know the first two instances can further be grouped into two types ($R1$ and $R2$) denoting different states of packet sending. It is easy to see that if we need a discrimination between packet sending and receiving, grouping the instances into R and S is enough. This example shows that the functions in a deeper layer determine more fine-grained details, although it is less important in abstracting the functionalities of an event procedure instance. Based on this notion, T-Morph can eliminate the trivial information (*i.e.*, the function calls in high layers) of the execution process, so as to classify event procedure instances into a much smaller number of types.

For two layered function sequences α and β, we define they are *the same in depth* n (n is an integer ≥ 0), if their functions and their orders in layers 0 to n are the same. For example, the left two layered function sequences in Figure 5 are the same in depth 4, but they are not the same in depth 5. Moreover, if α and β are the same in depth n, but not the same in depth $n+1$, it is possible that the functions in layer $n+1$ of one sequence is a substring of those of another. We consider two sequences with such a relationship are more similar than two without. To describe such similarity, we say α and β are *the same in depth* $n+0.5$ if they are the same in depth n and the functions in layer $n+1$ of one sequence is a substring of those of another.

Given a parameter m ($m = 0.5$, 1, ...), T-Morph can then classify a set of event procedure instances into different types so that in each type the layered function sequences of the instances are the same in depth m. We say the instances in the same type bear the same *event-procedure model*.

Thus, the instances with the sam event-procedure model have a common part in their layered function sequences. When m is a natural number, the common part can be obtained by removing the functions in the layers deeper than m. Otherwise (*i.e.*, $m = 0.5$, 1.5, ...), the common part can be obtained by removing functions in the layers deeper than m, except those in layer $m + 0.5$ that exist in all the layered function sequences.

Such a common part describes the similarity of the behaviors among the instances in each type. T-Morph uses this information as a model to summarize the functionalities of the instances for each type, and depicts it in a tree-like manner to the developers. Since function names can directly link to source codes, such a model provides a favorable way for human verification of the functionalities of each type.

Then, by substituting the event-procedure instances with their corresponding models, the execution process of an application can be abstracted as a sequence of event-procedure models. We name it an *event-procedure model sequence*, or in short, a *model sequence*. Since a model sequence describes the execution process of a TinyOS application, the job intended by the developers, *i.e.*, a job flow, lies in the sequence. Next we will discuss how to infer job flows in the sequence.

4.4 Mining job flows from model sequence

We observe that, running over simple hardware and driven by recurrent events (*e.g.*, timer timeouts or packet arrivals), TinyOS applications are generally designed to perform certain functionalities (*i.e.*, jobs) iteratively. This results in a repetition of a series of event-procedure models in the model sequence. Hence, job flows can be identified via mining the frequent patterns from the model sequence.

However, a TinyOS application running on a sensor node is usually designed to conduct several jobs simultaneously. As a result, its model sequence may contain several job flows, for example, those for sending a packet and those for obtaining a sensor reading. The execution of different job flows may overlap, disturbing the patterns of each other. It is a challenge to mine the frequent patterns from the model sequence directly. We tackle this problem with a separation-of-concern approach in the following two steps.

Step 1: Different intended jobs involve different interrupts. T-Morph divide the model sequence into several subsequences. The items in each one is related to the same interrupt i. We call it the *model sequence of interrupt* i.

Step 2: By controlling the parameter m when abstracting the functionalities of event procedure instances, we can derive event-procedure models of different levels of granularity. Starting with a small m, T-Morph allows the developers to check the derived models. If a model is not of interest, it can be filtered out from the model sequence of interrupt i. T-Morph then derives a new model sequence of interrupt i with a larger m. This process repeats to let the developers focus on more close-related event procedures until a job flow capturing the developer's intention is obtained. For example, in Figure 5, when $m = 4$, a model sequence of interrupt SPI containing R and S is obtained. If S (packet sending) is not of interest, T-Morph removes its related event-procedure instances and increases m to obtain a sequence containing only $R1$ and $R2$.

A job flow can now be mined from the resulting sequence with a frequent pattern mining approach, as described in Figure 6. For a substring s of size k_s in the sequence with length l, it finds how many times s appears (denoted by t_s). The percentage $\mathcal{P}_s$ of the occurrences of s in the whole sequence is:

$$\mathcal{P}_s = \frac{k_s \cdot t_s}{l} \ . \tag{1}$$

T-Morph considers that the substring j which results in the maximum $\mathcal{P}_s(\forall s)$ is the job flow of interest. Formally,

$$j = \operatorname*{argmax}_{\forall s} \mathcal{P}_s = \operatorname*{argmax}_{\forall s} \frac{k_s \cdot t_s}{l} \ . \tag{2}$$

If there are many such substrings, we select the one with the smallest size. For example, for a model sequence ABABAB ABABAB with size being 12, $\mathcal{P}_{AB}$ is $\frac{2\times6}{12} = 1$, which is the maximum among all substrings. Consequently, AB is the resulting job flow. Note that although $\mathcal{P}_{ABAB}$ is also equal to 1, the size of $ABAB$ is larger than AB. So it is not the resulting job flow. The complexity of the approach is $O(l^2)$.

4.5 Modeling job flow interleaving executions

Due to the concurrency model of TinyOS, the execution of a job flow instance can be interleaved. Such interleaving executions may cause problems if the design of the job flow and its interleaving parties bear some implicit dependencies. Hence, T-Morph should model the interleaving executions and illustrate them to the developers.

The execution of a job flow instance can be interleaved by event-procedure instances (which may belong to other job flow instances) in two ways. First, the execution of the event-procedure instances of the job flow can be interleaved by other event-procedure instances. Second, a job flow instance as a whole can be interleaved by the event-procedure instances occurring in between two of its adjacent member event-procedure instances, even if the executions of these event-procedure instances do not overlap.

For the first case, as pointed out in Section 3.2, such interleaving executions can be very complicated and diverse. A straightforward approach is to consider the detailed interleaving patterns, *e.g.*, where the interrupt handlers of other instances preempt their executions, and which tasks of other instances have been executed during their lifetime. Unfortunately, this will result in complicated interleaving models,

```
// INPUT:  A model sequence S of interrupt i
// OUTPUT: A job flow j
length ← the size of S
index ← 0
j ← 0
maxP_s ← 0
k_s ← 2
loop until k_s is larger than a threshold
   for each substring s of length k_s in S
      t_s ← the times s appears in S
      // P_s is the percentage of the occurrences
      // of s in S
      P_s ← (k_s * t_s)/length
      if maxP_s < P_s
         maxP_s ← P_s
         // so far the current s is the most likely
         // to be a job flow
         j ← s
      end if
   end for
   k_s ← k_s + 1
end loop
```

Figure 6: Job flow mining algorithm.

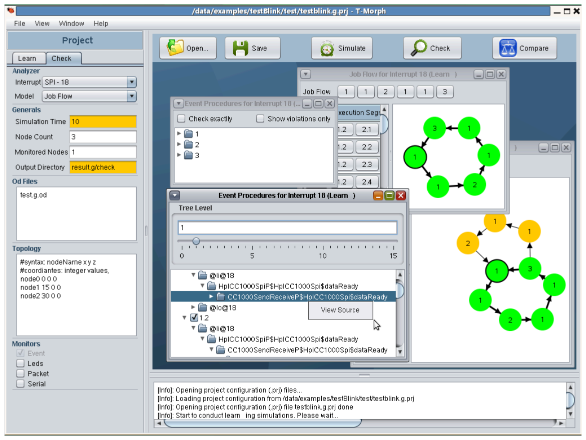
which provides no help for human verification. Hence, T-Morph takes a more reasonable way: It considers only in the abstracted event-procedure-model level as follows. For all event-procedure instances belonging to a job flow model (say J), T-Morph finds all the other interleaving event-procedure instances and records their models (constructed as discussed in Section 4.3) in a set $\mathcal{M}(J)$. $\mathcal{M}(J)$ is then used to describe how the event procedure instances in J is interleaved. Similarly for the second case, T-Morph finds all the other event-procedure instances occurring in between two adjacent event-procedure instances of J, and also includes their models in $\mathcal{M}(J)$.

Hence, for each job flow J, the developers can be directed to check if there is any unexpected dependency between J and its interleaving event-procedure models $\mathcal{M}(J)$. Since event-procedure models can be easily associated with their corresponding source codes, this interleaving modeling approach brings better focus on the source codes for the developers to verify correct executions of their applications. After all, details on how an event-procedure instance interleaves the execution of a job flow instance can still be provided if the developers need example cases.

5. TROUBLESHOOTING VIA VERIFIED BEHAVIOR MODELS

Testing TinyOS applications generally requires a long-term execution for exploring extensive program states so that potential bugs can manifest themselves [16][24][35]. Bug symptoms, however, are deeply hidden in the resulting long execution process. Even if the symptoms are identified, it is difficult to relate them to the source code defects due to the conceptual gap between the source codes and the execution process. This section discusses how T-Morph addresses this challenge in troubleshooting TinyOS applications.

First, let us discuss a key observation that motivates the automatic troubleshooting approach in T-Morph: Though it is labor-intensive to analyze a long testing execution process manually, it is generally an easy task for the developers to check whether a TinyOS application executes as expected with simple system settings. The developers can resort to T-Morph to run such a simple testing scenario (as described in Figure 3). T-Morph can mine the application behavior


/data/examples/testBlink/test/testblink.g.prj - T-Morph
File
View
Window
Help
Project
Learn
Check
Open...
Save
Simulate
Check
Compare
Job Flow for Interrupt 18 (Learn)
Event Procedures for Interrupt 18 (Learn)
Tree Level
View Source

```
Analyzer
Interrupt  SPI - 18
Model  Job Flow
Generals
Simulation Time  10
Node Count  3
Monitored Nodes  1
Output Directory  result.g/check
Od Files
test.g.od
Topology
#syntax: nodeName x y z
#coordiantes: integer values,
node0 0 0 0
node1 15 0 0
node2 30 0 0
Monitors
[✓] Event
[ ] Leds
[ ] Packet
[ ] Serial
```

```
▼ @li@18
  ▼ HplCC1000SpiP$HplCC1000Spi$dataReady
    ▶ CC1000SendReceiveP$HplCC1000Spi$dataReady
▶ @lo@18
▼ [✓] 1.2
  ▼ @li@18
    ▼ HplCC1000SpiP$HplCC1000Spi$dataReady
      ▼ CC1000SendReceiveP$HplCC1000Spi$dataReady
```

```
[Info]: Opening project configuration (.prj) files...
[Info]: Loading project configuration from /data/examples/testBlink/test/testblink.g.prj
[Info]: Opening project configuration (.prj) file testblink.g.prj done
[Info]: Start to conduct learn   ing simulations. Please wait...
```

Figure 7: An example workspace of `T-Morph`.

models from the execution process and visualize them. This can substantially help manual verification. Sometimes, we can even just check whether the outcome of the application is correct. For example, to check a packet forwarding mechanism of a sensor node (say u), we can use two other nodes (say v and w), v for sending a packet to u, and w for receiving the forwarded packet. If w can successfully receive the packet, we then confirm the testing result is correct.

If the simple testing scenario is considered correct by the developers, we call the model generated by `T-Morph` during this preliminary testing run the *verified models*. `T-Morph` can reinforce its knowledge of the target application with such verified models. This allows `T-Morph` to troubleshoot potential bugs in a long term testing execution process automatically, as also shown in Figure 3. With the same modeling approaches, `T-Morph` can abstract the long execution process into its behavior models (*i.e.*, event-procedure models, job flows, and how they are interleaved).

`T-Morph` then checks these behavior models against the verified models. If there are no violations found, `T-Morph` considers the target application in the long term testing run is correct. Otherwise, the model violations are considered as bug symptoms. There are two cases. The first one is that new event-procedure models or job flows have been found. This means somehow the behaviors of the application are changed. `T-Morph` will direct the developers to check whether these newly-introduced models are consistent with the design intention by visualizing them in a graphic manner. Second, if `T-Morph` finds new event-procedure models that interleave a job flow, this indicates the long term testing run has explored some new unexpected program states. `T-Morph` will also visualize these event-procedure models together with how they interleave the job flows for the developers to check whether they bear any unintended dependencies. In this way, `T-Morph` can direct the developers to the suspicious locations in the source codes that cause the inconsistency, which can significantly reduce human efforts in troubleshooting TinyOS applications.

6. CASE STUDIES

To show the power of `T-Morph` in modeling and verifying TinyOS applications, this section provides three representative case studies. We will see how bugs in source codes can be conveniently identified through verifying behavior models abstracted from the dynamic executions. All case studies are based on real applications distributed with TinyOS [29].

We implement `T-Morph` in Java, which contains over 10,000 lines of codes.[2] Like work in [16][35], `T-Morph` relies on Avrora, a widely-adopted WSN testing environment [30]. It provides a cycle-accurate emulation of hardware functionalities and their interactions, and executes an application in the instruction level. It can thus achieve the required fidelity in retrieving the application behaviors. Figure 7 demonstrates an example user interface of `T-Morph` workspace.

6.1 Case study 1: data forwarding

We first investigate how `T-Morph` models and visualizes wireless communication behaviors since wireless communication is a general functionality for every field-deployed sensor node. We pick a lightweight multi-hop packet forwarding protocol based on `BlinkToRadio` distributed with TinyOS [29] as our target application, and focus on the SPI interrupt since the wireless chip uses it to talk to the MCU.

Three nodes are deployed: node 0 as the sink, node 1 as the relay, and node 2 as the source. To analyze the packet forwarding mechanism, we examine how the application executes on the relay (*i.e.*, node 1) only.

First we consider a simple 3-second testing scenario, where the packet sending rate of node 2 is 1 packet/second. `T-Morph` quickly generates the event sequence models, A_{18}, B_{18}, and C_{18}.[3] With `T-Morph`'s visualization support, the functionality of these models can be instantly identified: A_{18} is for processing data during sending/receiving a packet, B_{18} is for signaling that a packet has been received and then forwarded, while C_{18} is for informing that a packet has been sent successfully, as shown in Figure 8. Based on the verified event procedure models, `T-Morph` mines a job flow with a simple pattern $A_{18}A_{18}B_{18}A_{18}A_{18}C_{18}$.[4] It is easy to verify this job flow is for receiving a packet, and then sending it. It is exactly the intended job of the application in a nutshell.

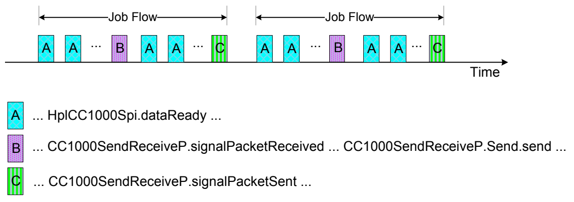
After we confirm the correctness of the application in the simple testing scenario, `T-Morph` takes the models generated in this testing run as the verified models. We then run a more complicated testing scenario by randomizing the packet sending rate of node 2, which results in random packet arrivals at node 1. As illustrated in Figure 3, `T-Morph` can automatically verify this new testing run by checking its behavior models against the verified models.

Although no violation to the event-sequence models are found, `T-Morph` finds one violation to the job flow model: a sequence of event procedure instances $A_{18}A_{18}B_{18}A_{18}A_{18}B_{18}$ $A_{18}A_{18}C_{18}$, as shown in Figure 9. Since $A_{18}A_{18}B_{18}$ denotes the process of receiving a packet, the violated sequence shows that node 1 has received two packets before sending one. Via an in-depth inspection of model B_{18}, we find that when the node receives a packet, it calls `Receive.receive`. Then, the first B_{18} instance forwards the packet directly, while the second actively drops the packet in function `AMSend.send` due to a busy flag. By examining the source

[2]`http://www.hkcloud.net/TMorph`

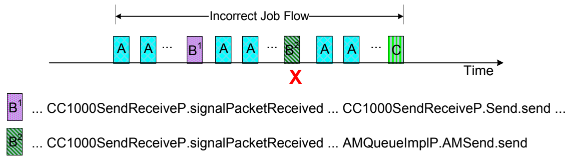
[3]The subscript 18 denotes that the event procedures are triggered by interrupt 18, *i.e.*, the SPI interrupt. We use similar notations for all event procedure models in the rest of our discussions.

[4]For notation simplicity, two consecutive identical models, *e.g.*, $A_{18}A_{18}$, indicate there are two or more consecutive identical event procedure models.



Job Flow
Job Flow
A A … B A A … C A A … B A A … C
Time
A … HplCC1000Spi.dataReady ...
B … CC1000SendReceiveP.signalPacketReceived ... CC1000SendReceiveP.Send.send ...
C … CC1000SendReceiveP.signalPacketSent ...

Figure 8: Job flow of data forwarding application.



Incorrect Job Flow
A A … B1 A A … B2 A A … C
Time
B1 … CC1000SendReceiveP.signalPacketReceived ... CC1000SendReceiveP.Send.send ...
B2 … CC1000SendReceiveP.signalPacketReceived ... AMQueueImplP.AMSend.send

Figure 9: An anomalous job flow instance.

codes, it is easy to see that the flag is set because node 1 is still in the process of sending the data packet. This means before the packet by the first B_{18} instance has been sent, another packet arrives unexpectedly, causing the application to drop the new arrival packet.

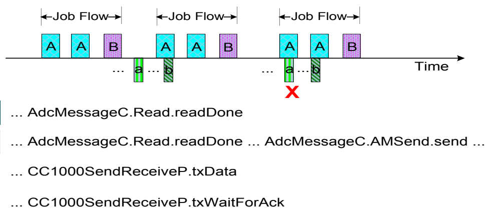
This case study shows that via identifying the violation to the job flow model, `T-Morph` can substantially help locate the defect in the source codes that causes accidental packet loss. Note that it is difficult for other approaches (*e.g.*, [35]) to detect such a defect since they consider only event procedures, but not their sequences.

6.2 Case study 2: data collection

Even a simple TinyOS application may involve multiple interrupts to accomplish its functionality. Consider an application adapted from `Oscilloscope` [29], which is also the motivating example discussed in Section 3. The major logics of this application are implemented via the ADC interrupt (*i.e.,* interrupt number 22). When a sensor reading is ready upon a periodic request, an ADC interrupt will be issued. As shown in Figure 2, after three sensor readings are collected, a task will be posted to initialize the sending of a data packet. This application involves several hardware interrupts: specifically, the timer timeout interrupt for triggering a periodical reading of a sensor, the ADC interrupt for obtaining the sensor readings, and the SPI interrupt for wireless communications. These interrupts cover most event types a typical application needs to handle.

Similarly to case study 1, `T-Morph` first runs a simple, easy-to-verify testing scenario, where the timer timeout period is 100ms. We focus on the ADC interrupt since it is centric to this application (see Step 1 in 4.4). `T-Morph` identifies two event-procedure models for the ADC events, one for reading sensor data and the other for performing Received Signal Strength Indicator (RSSI) functionalities. The RSSI-related model is for wireless communications. We can let `T-Morph` filter out its instances since we focus on sensor readings (see Step 2 in 4.4).

Thus, `T-Morph` performs a separation-of-concern analysis to the event-procedure models, and identifies two models, A_{22} and B_{22}. Both invoke the function shown in Figure 2. The difference between them is that B_{22} also posts a task to initialize the sending of a packet. This is consistent with the source codes in Figure 2. `T-Morph` then finds a job flow `T-Morph` with a simple pattern $A_{22}A_{22}B_{22}$.



Job Flow
Job Flow
Job Flow
A A B … a … b A A B … a … b A A B
Time
A … AdcMessageC.Read.readDone
B … AdcMessageC.Read.readDone ... AdcMessageC.AMSend.send ...
a … CC1000SendReceiveP.txData
b … CC1000SendReceiveP.txWaitForAck

Figure 10: A job flow model with interleaving event-procedure models.

The correctness in the simple testing scenario is then confirmed and `T-Morph` generates the verified models. Then we decrease the timeout period with a step size 20ms. `T-Morph` runs the application for each case to conduct a more complete testing, and checks the application behavior models against the verified ones. The job flow in these testing runs is still $A_{22}A_{22}B_{22}$, which is consistent with that in the verified models. However, `T-Morph` identifies a violation of the interleaving event-procedure models of the job flow. Specifically, as shown in Figure 10, a new model a_{18}, triggered by the SPI interrupt, interleaves the job flow.

`T-Morph` can associate a_{18} with the source codes and allows us to quickly realize its functionality is to transmit the content of a packet. This unexpected interleaving execution unveils that the previous packet is still being sent when a new sensor reading arrives. So we inspect the source codes that resulting in A_{22} and B_{22} to see whether the interleaving parties bear any dependencies. We then observe that the new sensor reading may pollute the packet content as it shares the same memory with the sending packet (Line 5 in Figure 2). The defect (a race condition between the reading saving codes and packet sending codes) can thus be located.

This case study shows that with a separation-of-concern mechanism, `T-Morph` works well to model the application behaviors, and it can effectively help identify source code defects caused by subtle interleaving executions of event procedures and job flows that bear implicit dependencies.

6.3 Case study 3: TYMO routing protocol

A routing protocol is crucial for sensor nodes in a WSN to transfer packets collaboratively. In this case study, we describe our experience of applying `T-Morph` to find a new bug in TYMO. TYMO is a TinyOS-based implementation of the DYMO (DYnamic MANET On-demand) routing protocol [28]. A testing scenario where four nodes (nodes 0 to 3) are reporting packets to a sink (node 4) via a one-hop communication is constructed. The reporting rate is 2 packets/second. We start our verification at node 3 and let `T-Morph` build its execution models.

We focus timer timeout interrupt (*i.e.*, interrupt 16) since it triggers packet reporting. `T-Morph` then finds one job flow, which contains one event-procedure model A_{16}. This is correct since the timer timeout in TYMO triggers a packet sending process generally.

To verify the functionality of the event-procedure model, we increase the value of parameter m, where `T-Morph` splits the event-procedure instances into several more detailed models. Figure 11(a) shows a part of the layered function sequence of a typical model, where all involved functions belong to the `ForwardingEngineM` and `MHEngineM` modules in TYMO. By showing its source codes, `T-Morph` allows us to know that the `selectRoute` function (vertex 2) is designed

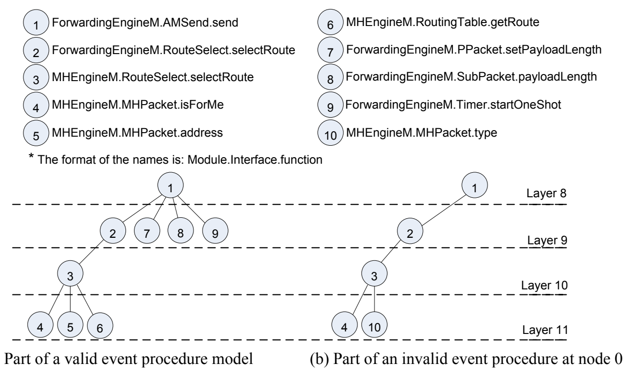

1 ForwardingEngineM.AMSend.send
2 ForwardingEngineM.RouteSelect.selectRoute
3 MHEngineM.RouteSelect.selectRoute
4 MHEngineM.MHPacket.isForMe
5 MHEngineM.MHPacket.address
6 MHEngineM.RoutingTable.getRoute
7 ForwardingEngineM.PPacket.setPayloadLength
8 ForwardingEngineM.SubPacket.payloadLength
9 ForwardingEngineM.Timer.startOneShot
10 MHEngineM.MHPacket.type
* The format of the names is: Module.Interface.function
Layer 8
Layer 9
Layer 10
Layer 11
(a) Part of a valid event procedure model
(b) Part of an invalid event procedure at node 0

Figure 11: Layered function sequences for Case 3

for TYMO to choose a route for sending a packet. It performs the following steps: After successfully determining a packet is not for itself via two function calls `isForMe` (vertex 4) and `address` (vertex 5), it tries to select a route by calling `getRoute` (vertex 6). If encountering a busy status, it passes the busy status back so that the packet sending process can be deferred by `startOneShot` (vertex 9). According to TYMO specification, such an execution process is correct. `T-Morph` then obtains the verified models, and check them against the behaviors of other source sensor nodes.

No violations are reported for sensor nodes 1 and 2. However, `T-Morph` finds a violation in node 0, *i.e.*, a new event procedure model. Figure 11(b) shows a part of its layered function sequence, which can be compared with the verified models. We can instantly see that node 0 always determines that it is the intended destination of a packet by calling `isForMe` (vertex 4). This leads to calling `type` (vertex 10), and it stops the packet sending process prematurely.

We can easily locate why `isForMe` always returns *true* for source node 0: The destination field of a packet is not set before calling `selectRoute`. It is left as its default value 0. This accidentally is the ID of node 0. Hence, the `isForMe` function of node 0 always returns *true*, indicating the packet is for the node itself. Thus, the packet will not be forwarded. With the support of `T-Morph`, we are the first to identify this bug in the latest release of TinyOS and locate its root cause.

7. FURTHER DISCUSSIONS

Let us first discuss some threats to the validity of our experiments, and the measures we take to address the threats, as well as some limitations of `T-Morph`. First, emulation is a popular way to verify the functionality and performance of TinyOS applications (e.g. [16]). `T-Morph` intentionally relies on *emulations*, but not real deployments, to capture the system runtime behaviors. The reasons are as follows. First, to test TinyOS applications, it generally requires us to explore a variety of application states (*i.e.*, testing scenarios) to hit the trigger conditions of bugs [16][24]. Hence, it is not cost-effective, if not infeasible, to conduct testing in real deployment. More importantly, to collect the behaviors on a real deployment will require the instrumentation of the target applications. The instrumented codes will inevitably disturb the behaviors of the target application by influencing the timing of the application logics, which will destroy the fidelity of `T-Morph`. Hence, emulations are a better choice than real deployment for `T-Morph` in out experiments.

Another possible threat is the fidelity of the emulation. `T-Morph` relies on Avrora [30]. It can provide high fidelity to the real world: It emulates hardware behaviors and their interactions with precise timing, and hence supports accurate interrupt preemptions and network communications. `T-Morph` can thus explore real-world interleaving executions of event procedures. Note that another widely-adopted simulator TOSSIM released with TinyOS [29] does not emulate hardware accurately, and hence cannot be an alternative.

The experiments are conducted in small-scale networks. But `T-Morph` does not limit itself to small-scale networks. `T-Morph` focuses on one sensor node to obtain the application behavior models. This mechanism is not influenced by the network size. Then, the verified models can automatically verify the correctness of the application running on any number of nodes.

Finally, our experiments employs short-term executions to obtain verified models and check them against long-term executions. Nevertheless, there is actually no sharp conceptual line between short-term executions and long-term ones. Any execution can be used to identify verified models, although it is surely easier to do so in short-term ones.

It is also important to identify the limitations of `T-Morph`. First, `T-Morph` is specifically tailored for TinyOS applications. There are, however, several other operating systems for WSNs, for example, Contiki [7], which `T-Morph` does not currently support. However, the general idea of mining the application behavior models from execution process, and linking them back to the source codes for verification and fault localization, can be applied to these systems. Second, `T-Morph` resorts to human inspection to verify the correctness of an application. We believe bug localization will inevitably involve human efforts. The aim of `T-Morph` is then to reduce such efforts.

8. CONCLUSION

TinyOS applications are fault-prone due to the conceptual gap of the WSN developers towards understanding the complicated execution processes of their static codes. We observe that system failures are typically triggered by unknown execution patterns of the source codes. This paper presents an effective tool, `T-Morph`, for mining, visualizing, and verifying the execution patterns of WSN applications. `T-Morph` features the dynamic execution process of a TinyOS application with sequences of event procedures and layered function calls. Through a user-friendly graphical representation, these models can be linked to the source codes directly. Thus, `T-Morph` can provide substantial help to developers to understand and verify how their static source codes are dynamically executed. We apply `T-Morph` to verify several representative TinyOS applications. Our experiments demonstrate that `T-Morph` can greatly save manual efforts in troubleshooting their TinyOS applications and eliminating software design defects before real-world deployment.

Acknowledgements

This work was substantially supported by the National Natural Science Foundation of China (Project No. 61100077), the National Basic Research Program of China (973 Project No. 2011CB302603), the Research Grants Council of the Hong Kong Special Administrative Region, China (Project Nos. CUHK 415311 and N_CUHK405/11). J. Liu's work was supported by a Canadian NSERC Discovery Grant.

9. REFERENCES

[1] G. Ammons, R. Bodik, and J. Larus. Mining specification. In *Proc. of the ACM PoPL*, pages 4–16, 2002.

[2] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli. The hitchhiker's guide to successful wireless sensor network deployments. In *Proc. of the ACM SenSys*, pages 43–56, 2008.

[3] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *Proc. of the ACM FSE*, pages 3–12, 2009.

[4] 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*, pages 85–98, Nov. 2008.

[5] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. Surviving sensor network software faults. In *Proc. of the ACM SOSP*, pages 235–246, 2009.

[6] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proc. of the ACM SenSys*, pages 205–218, 2007.

[7] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. of the IEEE LCN.*

[8] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, Oct. 1988.

[9] 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 PLDI*, pages 1–11, 2003.

[10] V. Kahlon, N. Sinha, Y. Zhang, and E. Kruus. Static data race detection for concurrent programs with asynchronous calls. In *Proc. of the ACM FSE*, pages 13–22, 2009.

[11] J. Kahn, R. Katz, and K. Pister. Next century challenges: Mobile networking for "smart dust". In *Proc. of the ACM MOBICOM*, pages 271–278, 1999.

[12] 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*, pages 99–112, 2008.

[13] N. Kothari, T. Millstein, and R. Govindan. Deriving state machines from tinyos programs using symbolic execution. In *Proc. of the ACM/IEEE IPSN*, pages 271–282, 2008.

[14] V. Krunic, E. Trumpler, and R. Han. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *Proc. of MobiSys*, pages 43–56, June 2007.

[15] S. Kumar. Specification mining in concurrent and distributed systems. In *Proc. of the ACM/IEEE ICSE*, pages 1086–1089, 2011.

[16] 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*, pages 94–104, Nov. 2008.

[17] K. Langendoen and A. B. O. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *Proc. of the IEEE IPDPS*, Apr. 2006.

[18] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32(6):382–403, June 2006.

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

[20] P. Li and J. Regehr. T-Check: Bug finding for sensor networks. In *Proc. of the ACM/IEEE IPSN*, pages 174–185, 2010.

[21] D. Lo and S. Maoz. Mining scenario-based triggers and effects. In *Proc. of the IEEE/ACM ASE*, pages 109–118, 2008.

[22] D. Lo, L. Mariani, and M. Pezze. Automatic steering of behavioral model inference. In *Proc. of the ACM FSE*, pages 345–354, 2009.

[23] ON World Inc. Industrial wireless sensor networks: A market dynamics report. Mar. 2010.

[24] J. Regehr. Random testing of interrupt-driven software. In *Proc. of the ACM EMSOFT*, pages 290–298, Sept. 2005.

[25] P. Reynolds, C. Killian, J. Wiener, J. Mogul, M. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proc. of the USENIX NSDI*, 2006.

[26] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weisez, S. Kowalewskiz, and K. Wehrle. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proc. of the ACM/IEEE IPSN*, pages 186–196, 2010.

[27] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: Global views of distributed program execution. In *Proc. of the ACM Sensys*, pages 141–154, 2009.

[28] R. Thouvenin. Implementing and evaluating the dynamic manet on-demand protocol in wireless sensor networks. Master Thesis, University of Aarhus, 2007.

[29] TinyOS Home Page. `http://www.tinyos.net`.

[30] B. Titzer, D. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the IEEE IPSN*, pages 477–482, May 2005.

[31] 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*, pages 381–396, Seattle, USA, Nov. 2006.

[32] 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/IEEE IPSN*, pages 416–423, Apr. 2006.

[33] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology*, 16(1), Feb. 2007.

[34] J. Yang, M. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *Proc. of the ACM SenSys*, pages 189–203, 2007.

[35] Y. Zhou, X. Chen, M. Lyu, and J. Liu. Sentomist: Unveiling transient sensor network bugs via symptom mining. In *Proc. of the IEEE ICDCS*, pages 784–794, 2010.