T-Check: Bug Finding for Sensor Networks

Peng Li
School of Computing, University of Utah, USA
peterlee@cs.utah.edu

John Regehr
School of Computing, University of Utah, USA
regehr@cs.utah.edu

ABSTRACT

Sensor nodes are resource poor and failure-prone. Sensor networks are composed of many such nodes that are often hard to physically reach and that are connected by unreliable wireless links. Together, these factors make sensor network debugging into a challenging activity, and in fact it is not uncommon for a deployed sensornet to encounter sporadic faults that are effectively impossible to locate, reproduce, and fix.

We developed T-Check, a tool that uses random walks and explicit state model checking to find safety and liveness errors in sensor network applications running on TinyOS. By building upon TOSSIM—an event-driven simulator that abstracts away interrupt-driven concurrency and other low-level hardware interaction—T-Check loses the ability to detect certain low-level errors, but gains enough scalability to detect distributed errors such as a collection tree protocol's failure to properly repair when a node dies. We have used T-Check to find previously unknown bugs in TinyOS.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and Embedded Systems; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

General Terms

Performance, Verification

Keywords

sensor networks, T-Check, model checking, random walk, TinyOS, safety, liveness, event-driven

1. INTRODUCTION

Creating and deploying a highly reliable sensor network is difficult, and it is not uncommon for a real network to have a data yield well below 100%. For example, the median node in Werner-Allen et al.'s volcano monitoring network [29] successfully downloaded 68.5% of detected events to a base-station. Of course, there are many kinds of root cause for a network's failure to deliver data: hardware failure, battery problems, software bugs, network link quality and variability, etc.

We developed T-Check to support early detection of software bugs in sensor network applications. Early detection is important because it can be very painful to find and fix a bug that sneaks into a deployment. We believe that bugs survive laboratory sensornet testing for the simple reason that the deployment environment is inevitably different from a controlled testbed or simulator. The apparent inadequacy of testing motivated us to choose an approach based on *state space exploration*: exhaustive enumeration of the states that a sensor network can find itself in. This approach can—in principle—find all possible violations of some kinds of program properties, regardless of the characteristics of the environment the sensornet is eventually deployed in.

T-Check employs model checking and random exploration of a sensornet's state space to find violations of safety and liveness properties. Safety properties, of the form "the sensornet never does X," can be specified by developers, and we also inherit a large number of compiler-generated safety properties from Safe TinyOS [7]. For example, a typical Safe TinyOS assertion would be "the array index is in the range 0..9." Liveness properties, of the form "the sensornet eventually does Y," must be provided by developers. Typical examples are "the buffer eventually becomes unlocked" or "the node eventually finds a place in the routing tree."

Due to complications such as interrupt-driven concurrency and free-running timer registers, the state space of even a single sensor node may be extremely large. For example, the [mc]square model checker [28] performs explicit-state checking of binaries for AVR microcontrollers. Although this approach has been used to check small industrial applications, it is unclear how to scale it up to networks of sophisticated sensornet programs. Therefore, we decided it was necessary to simplify the problem by abstracting away some low-level details. We settled on TOSSIM [22] as a suitable basis for T-Check.

TOSSIM is an event-driven simulator for networks of nodes running TinyOS; it gains simplicity and speed by not supporting concurrent execution and by emulating hardware devices at the level of a TinyOS interface, rather than at the level of hardware registers. This design means that TOSSIM cannot find, for example, timing errors, call stack overflows, and race conditions caused by interrupt preemptions. On the other hand, a coarse-grained, event-based execution model lends itself well to efficient state space exploration. Even so, in some cases we found TOSSIM to be too high-level to support effective bug-finding, and so we extended the ADC, serial, and SPI subsystems to model more low-level behavior.

The research questions that motivate our work include: What safety and liveness properties should be checked in sensornet executions? How can we efficiently explore the very large state space of a sensornet? Is it a good tradeoff to abstract away low-level details in order to find higher-level bugs? What sources of non-determinism in sensornet execution should be used as the basis for state space exploration? What are the tradeoffs between random testing and model checking? Can we effectively find bugs in the heavily used and generally high-quality TinyOS 2 code base?

Using T-Check, we found 12 previously unknown bugs in TinyOS 2. Most of these have been confirmed by developers, fixed in the TinyOS source code repository, and will be part of the upcoming 2.1.1 release.

2. BACKGROUND

This section provides some background on the systems and techniques on which T-Check is built.

2.1 TinyOS

TinyOS [15] is a wireless sensornet operating system. Its mechanisms and abstractions are designed for ultra-low-power microcontrollers with limited RAM and no hardware support for memory isolation. TinyOS typically runs at 1–8 MHz on 16-bit microcontrollers that have 4–10 kB of SRAM and 40–128 kB of flash memory [25].

The operating system uses components as the unit of software composition [15]. Like objects, components couple code and data. Unlike objects, however, they can only be instantiated at compile time. TinyOS components, written in a dialect of C called nesC [8], have interfaces which define downcalls ("commands") and upcalls ("events"). Upcalls and downcalls are bound statically: the absence of function pointers simplifies call graph analysis. Each TinyOS component is either a *module*, containing code, or a *configuration*: a container wiring together other modules and configurations.

The TinyOS core has a highly restricted, purely event-driven execution model. Using a single stack, it supports only interrupt handlers and run-to-completion deferred procedure calls called *tasks.* Tasks are similar to bottom-half handlers in UNIX implementations: they run at lower priority than interrupts and do not preempt each other.

Since TinyOS uses a single stack, computations cannot block. Instead, a *split-phase* idiom is used to permit concurrency during potentially long-running operations. For example, to send a packet, an application would invoke a `Send` command, which initiates the send operation and then returns. Later, the network subsystem delivers a `sendDone` event to the application, notifying it that the operation has finished.

2.2 TOSSIM

TOSSIM [22] is a simulator for TinyOS wireless sensor networks; it achieves high performance and good scalability in three ways. First, it compiles TinyOS source code into native host platform code, as opposed to simulating a sensor node at the instruction level. Second, it employs high-level device models that are very efficient, for example dealing with an entire radio packet at a time rather than simulating byte-wise or bit-wise transmission over the medium. Third, TOSSIM has a non-preemptive execution model: it does not simulate interrupts in a direct fashion.

As far as T-Check is concerned, the most important property of TOSSIM is that its simulation events execute atomically. In other words—unlike real sensornet nodes—code running on TOSSIM is never preempted by interrupts. This change to the TinyOS execution model greatly reduces the size of the state space that T-Check needs to explore. The cost of this choice is that certain kinds of errors, including timing errors, concurrency errors, and bugs in low-level device drivers cannot be detected by TOSSIM, and therefore not by T-Check either.

2.3 Safe TinyOS

Since nesC is an unsafe language and TinyOS nodes lack memory protection hardware, pointer and array bugs lead to corrupted RAM and difficult debugging. Some microcontrollers place their registers in the bottom of the memory map, exacerbating the problem. On these architectures, null pointer dereferences corrupt the register file.

Safe TinyOS [7] uses the Deputy compiler [6] to enforce type and memory safety using static and dynamic checks. Deputy is based on a dependent type system that exploits array bounds information already stored in memory. Therefore, unlike other memory-safe versions of C, it has no RAM overhead on a Harvard-architecture microcontroller.

From the point of view of T-Check, the main benefit that Safe TinyOS provides is the large number of assertions that it inserts into application code. These serve as inline safety property checks. T-Check can look for violations of these properties without additional help from the user.

2.4 Model Checking

A model checker [5] explores the state space of a computer system. For any given state, it is possible to define a predicate over that state: a formula that evaluates to true or false, indicating that the system in that state either holds or fails to hold some interesting property. An *execution* is a path through the state space that corresponds to an execution of the actual system.

A *safety property* is true if something bad never happens. If any state in an execution violates the safety property, the entire execution violates that property. A *liveness property* holds if something good will eventually happen. An execution satisfies a liveness property if the execution will encounter a *live* state in finite time. Conversely, an execution violates a liveness property if it contains an infinite sequence of states that does not hold the property.

T-Check builds on Killian et al.'s work [18], which is based on the idea that liveness violations can be detected heuristically by looking for sufficiently long violations of the property. Based on this intuition, liveness violations are separated into two categories: *transient* liveness violations that may eventually reach a live state, and *dead* liveness viola-

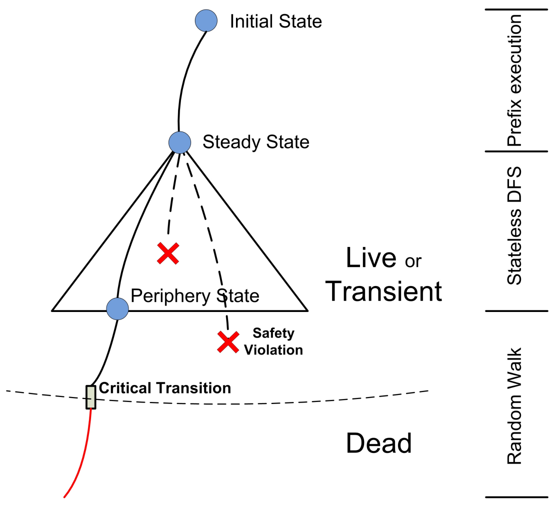

Figure 1: T-Check uses both random walks and depth-bounded model checking to look for violations of safety and liveness properties in sensor network applications

tions that will never reach a live state. The *critical transition* is the last state transition that is transiently dead. Finding the critical transition is useful because this point of no return usually has something to do with the root cause of the liveness violation.

3. T-Check

T-Check is a state space exploration tool that builds on TOSSIM. TOSSIM is a scriptable simulator, typically driven by a Python program. T-Check inherits this structure: a script is used to specify the number of nodes, their topology, etc.

T-Check supports two major modes of operation. First, it can act as a random tester. Second, it can act as a model checker whose overall strategy builds on work by Killian et al. [18]. In model-checking mode, T-Check runs in distinct phases:

1. Random execution is used to get the sensor network past its initialization phase and into a steady state.
2. Depth-bounded explicit state model checking is used to exhaustively explore the state space of the system up to some depth.
3. Random walks through the state space are used to find additional safety violations and to verify that potential liveness violations are real.
4. Additional random walks are used to find the critical transitions leading to liveness violations.

Figure 1 illustrates these steps. The following subsections describe T-Check in more detail.

3.1 Safety and Liveness Properties

T-Check exploits the type safety checks inserted by Safe TinyOS and also any user-written assertions already present in the source code. Liveness properties and additional safety properties may be provided by developers; T-Check ensures that each property holds at every state transition.

Table 1 shows the properties that we checked in TinyOS applications. We developed these properties by reading code, papers, and other documentation. It is likely that the authors or maintainers of these subsystems could do a better job than we did at characterizing the most important properties. To be integrated into T-Check, a property must be expressed as executable code; some examples are given in Section 4.

3.2 Exposing Non-determinism

Although TOSSIM incorporates some randomness, for example in its network model, it is generally a deterministic simulator. The firing times for events are determined by the expected real-world latencies of actions being simulated, and TOSSIM has a single mechanism for choosing the next event: it fires the event with the smallest time.

In contrast to TOSSIM, to successfully explore the state space of a sensornet—regardless of whether model checking or random walk is used—T-Check should exploit all available sources of non-determinism. If a non-deterministic choice is missed, we will fail to explore some part of the state space and may miss bugs. On the other hand, if non-determinism is added where it was not present in the original system, we will explore false paths and report errors that do not actually exist.

The first kind of non-determinism that T-Check uses is *communication non-determinism*. For example, a sent packet is non-deterministically delivered or dropped; a received packet is non-deterministically successful or corrupted. At the implementation level, each call to the radio model has to be replaced with a non-deterministic choice operator. In model checking mode, the non-deterministic choice operator explores both alternatives. In random walk mode, the non-deterministic choice operator returns a random alternative.

The second kind of non-determinism supported by T-Check is *coarse-grain node-level non-determinism*, which includes node arrival, death, and reboot. The third kind of non-determinism is *event ordering non-determinism*, which exploits the lack of ordering guarantees within a single TinyOS node. For example, if an application initiates two split-phase operations, such as sending a packet and reading a block from flash memory, the completion events can arrive in either order. However, events cannot be arbitrarily reordered. For example, within a node, if an application posts two tasks, the TinyOS scheduler guarantees that they run in the order in which they were posted.

To model the TinyOS execution semantics, T-Check maintains multiple event queues for each node that it simulates; Figure 2 illustrates this. Events from different queues may be arbitrarily reordered, but events in a single queue must execute in order. The event at the head of a queue is *enabled*, and the *enable set* for a node is just the set of all enabled events.

To determine the proper mapping of events to queues, we observed that the basic source of non-determinism in microcontroller execution is interrupts. Thus, T-Check maintains one queue for each interrupt-generating device, in addition to a queue for the TinyOS pending tasks. Recall that T-Check, like TOSSIM, does not support preemptive interrupts. Rather, interrupts are modeled as atomic events whose firing may be interleaved with the firing of tasks and

Application	Type	Property
Serial Stack	Liveness	$\forall n \in \textbf{nodes} : \neg n.bufZeroLocked \wedge \neg n.bufOneLocked$ Eventually, each of two buffers becomes unlocked
	Safety	$\forall n \in \textbf{nodes} : \neg n.isCurrentBufferLocked() \wedge$ PKT_COMING_IN $\rightarrow$ RECV_BEGIN The current buffer, if unlocked, should successfully receive an incoming packet
CTP	Liveness	$\forall n \in \textbf{nodes} : n(.parent)^* \neq n$ Eventually, there is no loop in the routing tree
	Liveness	$\forall n \in \textbf{nodes} : n$ has no path to sink $\vee$ $n \in trees$ Eventually, all nodes that have a path to a sink become part of some collection tree
Drip	Liveness	$\forall n \in \textbf{nodes} : (n.valueCache =)^*$ VALUE Eventually, all nodes have consistent values
Dip	Liveness	$\forall n \in \textbf{nodes} : (n.valueCache =)^*$ VALUE Eventually, all nodes have consistent values
Dhv	Liveness	$\forall n \in \textbf{nodes} : (n.valueCache =)^*$ VALUE Eventually, all nodes have consistent values
FTSP	Liveness	$\forall n \in \textbf{nodes} : (n.synchronized =)^*$ TRUE Eventually, all nodes achieve time synchronization

Table 1: TinyOS 2 properties that we checked

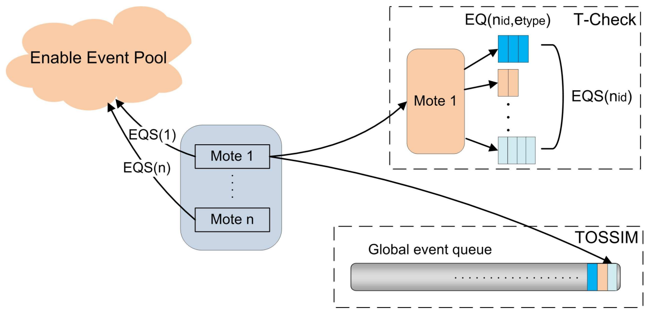

Figure 2: T-Check maintains multiple event queues per node to model event-ordering non-determinism

of other interrupt events.

TOSSIM models only the timer interrupt. While creating T-Check, we extended TOSSIM to model more interrupt sources: ADC (analog to digital converter), UART (serial port) transmit and receive, and SPI (serial peripheral interface). These extensions provide two benefits in terms of bug-finding. First, they permit additional low-level device driver components to be tested. Second, they permit non-deterministic executions driven by these interrupt sources to be explored by T-Check.

Since T-Check does not support preemptive execution of interrupts, and since (on a real mote) interrupts must always be enabled in between executing adjacent tasks, we do not need to model the processor's global interrupt enable flag. On the other hand, the individual enable bits associated with interrupt sources must be modeled in order to avoid exploring infeasible parts of the state space. For example, on AVR platforms the ADC interrupt can only fire when the ADIE hardware bit is set. T-Check models this behavior and considers events in the ADC event queue to be enabled only when the bit is set.

3.3 Depth-bounded Explicit State Model Checking

T-Check is an *execution driven* model checker: it actually runs the code, as opposed to symbolically evaluating it. T-Check uses *stateless depth-bounded depth-first search with partial order reduction.* Illustrated by Figure 1, starting from an initial state or steady state, all reachable states are explored using DFS. *Stateless* execution means that backtracking is implemented by returning to the initial or steady state and re-executing down some new path, as opposed to returning to a saved checkpoint other than the initial state. This approach is relatively simple to implement and conserves RAM, at the expense of wasting CPU time in redundant re-execution.

A naïve exploration of the state space of a distributed system is wasteful. For example, consider a network of two nodes that both have pending ADC interrupt handlers. The model checker has two choices: it can execute node 0's interrupt handler and then node 1's, or vice versa. However, since there is no dependency between the ADC interrupt handlers on two different nodes, the final system state is the same regardless of which handler runs first. *Partial order reduction* (POR) [10] is a family of strategies for avoiding exploration of redundant states. T-Check implements a form of *static POR*, which requires advance knowledge of which state transitions are potentially dependent. The following rules are used:

- A pair of transitions on the same node is always dependent.
- A pair of transitions on different nodes is independent unless the events are a matched send/receive pair.

The goal of T-Check is to explore all non-redundant states of the distributed system that can be reached within a predetermined number of state transitions. It works as follows. Initially, the *sleep set* and *transition stack* are empty. The sleep set supports POR and the transition stack records the sequence of state transitions currently being explored. Also, the initial state for the run is saved so that the model checker can return to it later.

A model checking step starts by checking if any safety property is violated. If so, T-Check prints an error message and dumps the current transition stack, which serves as a counterexample for the property. Next, T-Check builds a *ready set*: the set difference of the union of all nodes' enable sets and the sleep set. If the ready set is empty, the system backtracks. Otherwise, T-Check removes an element t from this set, pushes t onto the transition stack, and executes the corresponding code. Next, all events dependent on t are removed from the sleep set. Finally, as long as the depth of the transition stack does not exceed the pre-determined depth bound, the model checking step code is recursively invoked.

To backtrack, either because the depth bound is reached or the ready set is empty, T-Check pops the last transition from the transition stack, inserts it into the sleep set, restores the system to its saved (initial or steady) state, and then executes the state transitions determined by the contents of the transition stack. Once all state transitions have been performed, the model checking step operation is invoked.

Resetting the sensornet to a saved state requires restoring all nodes' state variables, register values, and event queues. We modified the nesC compiler to generate code to help save and restore nodes' states.

3.4 Randomly Exploring the State Space

The advantage of model checking is that it guarantees that any bugs within the depth bound can be found. However, the exponential size of the space imposes strong limits on the utility of this technique. In practice, random execution is a useful counterpart to model checking [12]. As shown in Figure 1, when the system reaches the depth bound, T-Check can also continue with a random walk phase to catch more safety errors and identify the potential liveness violations. Our random walk algorithm is to repeat these steps until a safety bug is found or until the user gets tired of waiting.

First, check if any safety properties are violated. If so, dump the current event trace and terminate. Otherwise, for each liveness property and each node, clear the *violation count* for any satisfied property and increment the violation count for any violated property. If the violation count for any liveness property exceeds a heuristic threshold (100,000 events has worked well for us), signal a liveness violation. Finally, choose a random event from a random node's enable set and execute it.

A tricky aspect of random testing is assigning appropriate probabilities to various event choice operators. If probabilities are chosen poorly, testing will waste time in uninteresting parts of the state space and miss interesting parts. By default, T-Check assigns uniform probabilities to all enabled events, since during each state, the enable event set for each node is similar, and thus the whole simulation execution topology is not irregular, resulting in relatively uniform trace sampling probability. Although this has worked well so far, in the future we plan to look for other probability assignments that find property violations more rapidly. T-Check also permits users to specify their own probability distributions if they so choose.

T-Check implements Killian et al.'s *critical transition* algorithm [18]. This is a binary search where each transition in a liveness-violating trace is used as the starting point for a random walk that tests for eventual liveness. The critical transition occurs between the last state from which a live state can be reached, and the subsequent state, which is the first state that is definitely dead. Once an execution trace is considered as liveness-violating, T-Check will dump a trace starting from the first transition of random walk to critical transition, making users better track down the root cause.

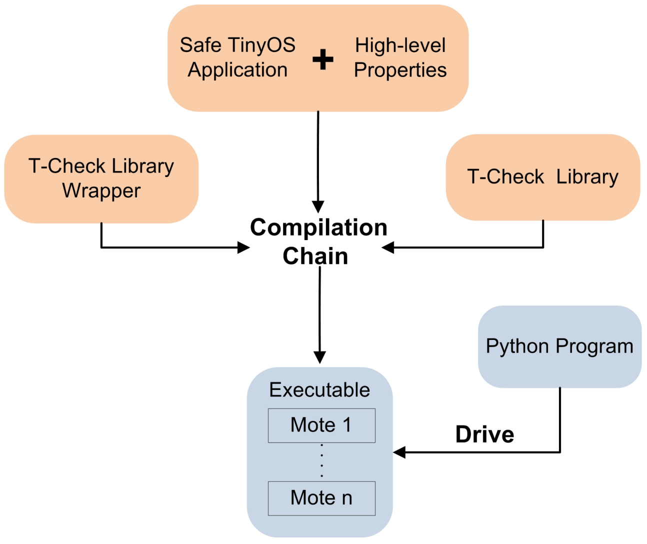

Figure 3: The T-Check toolchain

3.5 Finding Short Error Traces

Error traces found while model checking tend to be short, and in fact if the model checker runs to completion, the shortest possible error trace for each violated property is guaranteed to be among the errors found. On the other hand, random execution tends to lead to long error traces, making it difficult for users to understand the sequence of events that lead to the problem.

To shorten an error trace found during random execution, T-Check uses an algorithm loosely inspired by delta debugging [30]. At all times, the shortest-known trace to the error is saved. First, one of the state transitions in the error trace is chosen as a *change point*. All transitions before the change point come from the saved error trace, whereas transitions after it are chosen randomly. The change point is chosen heuristically: we use both an exponential search and pure random choice. If the new trace reaches the length of the saved trace without finding the error, the new trace is discarded and the current iteration ends. If the new trace finds the error more quickly, it becomes the new shortest trace and the current iteration ends. The algorithm terminates after a fixed number of iterations. Another heuristic we have found to be effective is to bias execution following the choice point to give increased probability to events on the node that is expected to show the property violation. Although it is simple, when iterated tens of thousands of times this technique is quite effective at reducing the length of error traces.

4. USING T-CHECK

Figure 3 shows the T-Check toolchain; it is used as follows. First, if a developer wishes to provide high-level properties, these take the form of nesC components providing the `SimProperty` interface, which has two commands: `safetyPropertyCheck` and `livenessPropertyCheck`. T-Check provides a script that automatically wires property components into the application. The first property listed in Table 1, which requires that the serial stack eventually unlocks its buffers, is implemented as:

Component	Safety	Liveness
MultihopOscilloscopeC	1	0
LinkEstimatorP	0	1
SerialDispatcherP	1	1
DipSummaryP	2	0
DipVersionP	1	0
MHPacketM	1	0
DhvSummaryP	1	0
DhvVBitP	2	0
DhvHSumP	1	0

Table 2: Summary of bugs found by T-Check

```
command bool SimProperty.livenessPropertyCheck() {
  return !receiveState.bufZeroLocked &&
         !receiveState.bufOneLocked;
}
```

Checks that inspect the state of multiple nodes are slightly more complex and the `sim_set_node` utility function is needed to change the current context. For example, the third property listed in Table 1 can be implemented as:

```
command bool SimProperty.livenessPropertyCheck() {
  int tmpMote = sim_node(), mote;
  for (mote = 0; mote < sim_simulated_mote_num(); mote++) {
    sim_set_node (mote);
    if (sim_mote_forms_loop (mote)) {
      sim_set_node (tmpMote);
      return FALSE;
    }
  }
  sim_set_node (tmpMote);
  return TRUE;
}
```

Second, a developer configures T-Check using an extended version of TOSSIM's configuration mechanism, via a Python script. New configuration parameters include probabilities used in random execution, depth of model checking, whether to use partial order reduction, etc.

Finally, the application is compiled as usual, for example by `make micaz safe sim`, and run. If a T-Check-enabled application encounters a safety violation, it will dump an execution trace leading to the violation. If a liveness violation is found, T-Check searches for the critical transition and dumps an execution trace leading to this transition.

5. RESULTS

This section demonstrates the effectiveness of T-Check as a bug-finding tool and evaluates its performance.

5.1 Bugs Found

We evaluated T-Check by using it to look for bugs in TinyOS applications from the publicly accessible `tinyos-2.x` and `tinyos-2.x-contrib` CVS repositories. Table 2 summarizes our results: we found 12 previously unknown bugs, some of them in core services and in applications that have been used for several years.

5.1.1 Serial Stack Bug #1

Typically, a sensor network contains one or more base station nodes that communicate occasionally or continuously with a PC using a serial link. To support this, TinyOS has a serial stack [11]. The `SerialDispatcherP` component aggregates incoming bytes into packets and dispatches them to the appropriate higher-level component. It uses double buffering so that a node can receive data into one buffer while application code is processing the other buffer. Figure 4 shows the data structure representing this component's internal state, and also a utility function for checking whether the current buffer is locked. A buffer is locked if it is being used by the application or the serial stack, and unlocked when it is idle and ready to be used.

```
typedef struct {
  uint8_t which:1; // Indicates current buffer
  uint8_t bufZeroLocked:1; // buffer 0 locked?
  uint8_t bufOneLocked:1;  // buffer 1 locked?
  uint8_t state:2;
} recv_state_t;
recv_state_t receiveState;

message_t* messagePtrs[2] =
  { &messages[0], &messages[1] };

bool isCurrentBufferLocked() {
  // BUG -- the switch cases are reversed
  return (receiveState.which) ?
    receiveState.bufZeroLocked : receiveState.bufOneLocked;
}
```

Figure 4: Code for serial bug #1

We used T-Check to test the following safety property: if a packet is arriving over the wire, and the current buffer is unlocked, it should be possible to successfully receive the packet and deliver it to the application level. T-Check found a violation of this property when `buffer0` is the current buffer, and is receiving a serial packet from the PC. When packet reception ends, the current buffer is switched to `buffer1` and the serial stack posts a task to notify the application level that `buffer0` is ready to be used.

There is a period of time, then, when `buffer0` is locked while the application is processing the serial data. If, during this time period, the serial stack starts receiving more data, it attempts to lock `buffer1` so that it can be filled with data. However, `isCurrentBufferLocked` (shown in Figure 4) contains a flaw that causes it to erroneously report the lock status of `buffer0` instead of `buffer1`, causing the serial stack to fail to receive the new incoming packet, violating the safety property.

The fix is obvious: the if and else branches of `isCurrentBufferLocked` need to be switched around. The TinyOS 2 maintainers committed this fix.

5.1.2 Serial Stack Bug #2

This bug is also in `SerialDispatcherP`. Figure 5 shows the actions taken by the serial stack when the last byte of a packet is received. If the receive was successful, and if the serial stack is not currently waiting for the application to process the other buffer, a buffer swap is performed and a task is posted which will signal application-level code that a packet was received.

A high-level liveness property for serial stack is that eventually, both `buffer0` and `buffer1` become unlocked. The serial stack contains a bug that permits it to deadlock, violating this property. The bug is triggered by a failed packet

```
async event
   void ReceiveBytePacket.endPacket(error_t result) {
    uint8_t postsignalreceive = FALSE;
    atomic {
      if (!receiveTaskPending && result == SUCCESS) {
        postsignalreceive = TRUE;
        ... code omitted ...
      }
      // BUG -- the buffer never gets unlocked when
      // the if condition is not true
    }
    if (postsignalreceive){
      post receiveTask();
    }
}
```

Figure 5: Code for serial stack bug #2

```
command void* DipSend.getPayloadPtr() {
  // returns NULL if message is busy
  if(busy) {
    return NULL;
  }
  return call NetAMSend.getPayload(&am_msg, 0);
}

command error_t DipDecision.send() {
   dip_msg_t* dmsg;
   dmsg = (dip_msg_t*) call SummarySend.getPayloadPtr();

   // BUG -- dsmg can be NULL here
   dmsg->type = ID_DIP_SUMMARY;

   ... code omitted ...
}
```

Figure 6: Code for DIP bug #1

receive, for example due to a failing CRC. In this case, the buffer never gets unlocked, nor does the buffer swapping logic execute. Thus, all subsequent serial packets find the current buffer locked, and they are dropped. The fix is to release the lock on the current buffer when an erroneous packet is received. The TinyOS 2 maintainers have committed this fix.

5.1.3 DIP Bug #1

DIP [23] is a data discovery and dissemination protocol. Figure 6 shows the code to send the DipSummary message, which is to summarize and hash the version information of the data items within a range. The `getPayloadPtr` command may return NULL, causing the subsequent line to dereference a NULL pointer with unpredictable results (or with a safety violation, if Safe TinyOS is being used). The fix—committed by the TinyOS 2 maintainers—is to add a check which fails the `send` command when `getPayloadPtr` returns NULL.

5.1.4 More DIP Bugs

Figure 7 shows the code of `findRangeShadow`, which calculates left and right indices in the array `shadowEstimates` for subsequent hash computation. This code contains two bugs that permits out-of-bounds array accesses to occur. The fix for these problems is slightly involved and we describe the

```
void findRangeShadow(dip_index_t* left,
                     dip_index_t *right) {
   // Precondition: shadowEstimates is an array
   // with UQCOUNT_DIP elements

   ...  code omitted ...
   // Here, even if (LBound + len) <= UQCOUNT_DIP,
   // then we can't guarantee that RBound does not
   // go over UQCOUNT_DIP
   if(LBound + len > UQCOUNT_DIP) { RBound = UQCOUNT_DIP; }
   else { RBound = highIndex + len; }

   ... code omitted ...
   for(i = LBound ; i + len <= RBound; i++) {
      est1 = shadowEstimates[i];
      // When the RBound is violated,
      // this access is out-of-bound
      est2 = shadowEstimates[i + len];

      ... code omitted ...
   }
   *left = highIndex;
   *right = highIndex + len;
}
```

Figure 7: Code for DIP bugs #2 and #3

process of finding and fixing these bugs in three steps.

First, a developer changed

```
est2 = shadowEstimates[i + len]
```

to

```
est2 = shadowEstimates[i + len - 1]
```

If the maximum of `RBound` is exactly `UQCOUNT_DIP`, then there would be no out-of-bounds access. T-Check then found a trace where `RBound` can go over `UQCOUNT_DIP`, causing another safety violation.

Second, a developer changed the

```
LBound + len > UQCOUNT_DIP
```

test to

```
highIndex + len > UQCOUNT_DIP
```

After this change, is it guaranteed that `RBound` is not beyond `UQCOUNT_DIP`. Although this fix eliminated one array bounds violation, T-Check subsequently found a case where the residual value in `right` was incorrect, causing an array access error in a different function.

Finally, to avoid the incorrect value of `right`, `i + len <= RBound` was changed to `i + len < RBound`, and `shadowEstimates[i + len - 1]` was restored to `shadowEstimates[i + len]`. T-Check has found no further problems.

5.1.5 A Link Estimator Bug

The Collection Tree Protocol (CTP) [9] computes anycast routes to a single or a small number of designated sinks in a wireless sensor network. Figure 8(a) shows an example of a collection tree computed by CTP during a T-Check run. The first liveness property that we wrote for CTP is that eventually, all cycles are removed from the collection tree. We found no violations of that property.

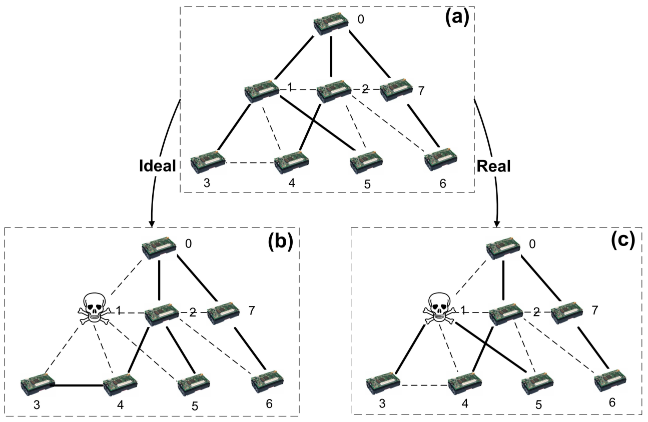

Figure 8: Example collection tree. Neighbors are connected by dotted lines, parents and children are connected by solid lines. (a) denotes a collection tree computed by CTP; (b) shows the desired repaired tree after node failure; (c) shows the actual collection tree after node failure, for a buggy version of TinyOS

```
void updateNeighborEntryIdx(uint8_t idx, uint8_t seq) {
    // packetGap means the packet number gap between
    // currently received packet and last received packet
    uint8_t packetGap;

    // BUG -- this test may never be true
    if (packetGap >= BLQ_PKT_WINDOW) {
      updateNeighborTableEst(NeighborTable[idx].ll_addr);
    }
}
```

Figure 9: Code LinkEstimator bug

The second property that we wrote checks that for any node that is transitively connected to some sink, it should eventually become part of some collection tree. Our initial CTP tests found no violations of this property either. However, at one point a CTP developer committed a change to the TinyOS link estimation component that prevented a dead node's children from joining the CTP tree, as shown in Figure 8(c), as opposed to the desired behavior shown in Figure 8(b). T-Check found that the critical transition was the death of node 1.

The code responsible for this problem is shown in Figure 9. The problem is that to rejoin the network, the `updateNeighborTableEst` function must be invoked, but for this to happen, a number of packets must be lost. In an insufficiently lossy network, `packetGap` never reaches `BLQ_PKT_WINDOW` and the node remains disconnected forever. We discovered this bug in parallel with other developers.

5.1.6 *An Array Bounds Bug*

Figure 10 shows code that relies on a timing race to reset an array index in the MultihopOscilloscope application. The timer `fired` event initiates a sensor read under the assumption that the `readDone` event will happen before the next timer arrives. If the timer expires first, it is possible for the variable `reading` to be used as an array index

```
typedef struct oscilloscope {
   ... code omitted ...
   uint16_t readings[NREADINGS];
} oscilloscope_t;
oscilloscope_t local;

uint8_t reading;

event void timer.fired() {
   if (reading == NREADINGS) {
      ... code omitted ...
      reading = 0;
   }
   if (call Read.read() != SUCCESS)
     fatal_problem();
}

event void Read.readDone(error_t result, uint16_t data) {
   ... code omitted ...
   // BUG when reading >= NREADINGS
   local.readings[reading++] = data;
}
```

Figure 10: Code for a bug in MultihopOscilloscope

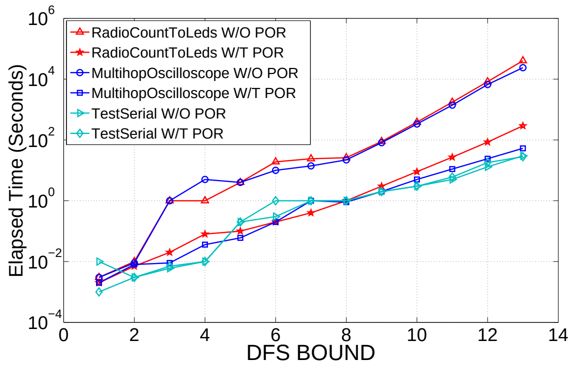

Figure 11: Time to model check three applications with and without partial order reduction. For RadioCountToLeds and MultihopOscilloscope, a two-node topology was used. TestSerial is single-mote application and therefore fails to benefit from POR.

when it is larger than NREADINGS−1. If the `Read` interface is wired directly to a sensor, there is no bug: the second call to `Read.read` will fail because the resource is still busy, and the extra `readDone` event cannot happen. On the other hand, if `Read` is wired through a resource arbiter, as it is on the MicaZ platform, then the extra `read` command succeeds after being placed into the arbiter's queue, creating the possibility that two `readDone` events will be signaled before the timer has a chance to reset the index, causing the out-of-bounds access. This bug is also present in the Oscilloscope and TestMultihopLqi applications.

5.2 Effectiveness of Partial Order Reduction

Figure 11 clearly shows the exponential relation between the depth bound and the time required to run the model checker. For a 13-step model checking run, POR speeds up T-Check by a factor of 449 for MultihopOscilloscope and a factor of 138 for RadioCountToLeds.

5.3 Comparing TOSSIM, Random Walk, and Model Checking

Table 3 compares bug-finding power of TOSSIM against T-Check in both model-checking and random walk modes, for two-node and eight-node topologies. For TOSSIM with Safe TinyOS, we ran 500,000 steps. For T-Check in model checking mode, we randomly executed 30,000 steps to get a steady state, then performed model checking with a 50-step bound, timing out after 10 hours. For T-Check in random walk mode the maximum number of steps was irrelevant since the bug was always found. Since only the model checker is deterministic, the results for TOSSIM and for T-Check in random walk mode are averaged over 25 runs.

TOSSIM only found four of the 10 safety bugs, and its traces were always longer than our shortened traces. There are two factors at work. First, T-Check models each sensornet node at a slightly more detailed level than TOSSIM (while still falling well short of modeling all the detail of a real mote). Second, T-Check exploits the abundant non-determinism in sensornet executions, whereas TOSSIM is largely deterministic.

We were surprised to find that random testing out-performs model checking in terms of bug-finding power. It is possible that additional optimizations to our model checker, such as exploiting the independence of some pairs of events inside a node, or implementing dynamic POR, would reverse this trend. As things stand, the main advantage of model checking is that if it finds an error, the event trace leading to that error is guaranteed to be the shortest one.

5.4 Unchecked Code

Table 4 summarizes the kinds of code that T-Check does and does not check. The main components that we miss are the drivers for timers and for the CC2420 radio chip. All of these components are heavily-used and seem quite solid in practice. Even so, it would be worthwhile devising ways to locate any residual bugs in them, for example using the [mc]square model checker for AVR object code.

6. RELATED WORK

Our work is most closely related to MaceMC [18]. Like TinyOS, Mace provides an event-driven infrastructure for constructing distributed systems. Unlike TinyOS, but like TOSSIM, Mace's events run to completion, providing a good match for an explicit state model checker. T-Check directly implements MaceMC's algorithm for finding the critical transition, and has adapted Mace's overall blend of random testing and model checking.

Harbor [21], t-kernel [13], and Safe TinyOS [7] all aim to catch memory safety errors in sensor network applications by adding runtime safety checks. T-Check is complementary to these efforts: regardless of how safety checks are added, a state space explorer provides a good way to look for violations of them.

KleeNet [27] uses symbolic analysis to generate test cases for sensor network code, and has been used to find new bugs. KleeNet and T-Check each have advantages and disadvantages; it would be interesting to find out which bugs we found can also be found using KleeNet, and vice versa.

EnviroLog [24] automatically and accurately records all events generated by lower-layer and can replay them for system tuning and performance evaluation of sensornets. TracePoint [4] and NodeMD [20] both use user-provided annotations and logging to effectively detect, trace, and debug the software faults in sensor network. Dustminer [17] and Khan et al.'s work [16] apply data mining techniques to logs to detect and catch complex interactive bugs.

Nguyet and Soffa [26] looked at ways to represent the internal structure of TinyOS applications. This kind of work should be directly useful to efforts like T-Check, for example to support inference of independent tasks and interrupts on the same node to perform better partial order reduction. Similarly, Kothari et al. [19] inferred the state machines hidden inside TinyOS applications. These machines probably abstract away too many details to be directly useful in model checking, but they may provide convenient states for T-Check's high-level property specifications to use.

TinyOS interface contracts [1] add many safety conditions to TinyOS applications. We would have liked to use T-Check to look for contract violations, but significant work would have been required to port contracts to TinyOS 2.

Symbolic model checkers like BLAST [3] and SLAM [2] generate abstract models from source code, and reason about them. In contrast, execution-driven model checkers like Verisoft [10] and Java Pathfinder [14] execute the code directly. T-Check is an example of the latter family of model checkers.

7. CONCLUSION

We have presented T-Check, a tool that exploits both explicit state model checking and random walks to find bugs in sensor network applications running on TinyOS. T-Check offers users a good value proposition: by exploiting safety checks inserted by Safe TinyOS, users can find bugs without any extra annotation effort. However, if users provide additional, higher level safety and liveness properties, these can also be checked. User-specified properties may be in terms of a single node (e.g., "packets are eventually received") or in terms of the entire network (e.g., "eventually, all nodes are part of the routing tree"). We have used T-Check to find 12 previously unknown bugs in TinyOS 2.1, and we plan to make T-Check available to the sensor network community as open-source software.

8. ACKNOWLEDGMENTS

The authors would like to thank Adam Dunkels (our shepherd), David Gay, Phil Levis, Eric Mercer, Raimondas Sasnauskas, Bastian Schlich, and Matthias Woehrle for much useful feedback on our work. This material is based upon work supported by the National Science Foundation under Grant No. 0615367.

9. REFERENCES

[1] Will Archer, Philip Levis, and John Regehr. Interface contracts for TinyOS. In *Proc. of the Intl. Conf. on Information Processing in Sensor Networks (IPSN'07), SPOTS Track*, Cambridge, MA, April 2007.

[2] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. of the 29th ACM Symp. on Principles of Programming Languages (POPL)*, Portland, OR, USA, January 2002.

Component (Bug Type)	Safe TOSSIM		T-Check Model Check		T-Check Random Walk		Shortest Trace
	2-node	8-node	2-node	8-node	2-node	8-node	8-node
SerialDispatcherP (Reversed Switch)	N/A	N/A	50 (100)	50 (100)	2,280	2,280	110
MultihopOscilloscopeC (OOB)	N/A	N/A	104 (1,081,717)	N/A	89,273	39,964	111
DipSummaryP (NULL pointer)	N/A	N/A	1,139 (1,139)	156 (16,108,781)	33,883	14,150	1,464
DipSummaryP (OOB #1)	7,206	4,236	252 (252)	46 (46)	11,821	3,729	1,332
DipSummaryP (OOB #2)	7,164	27,866	158 (75,683)	146 (488,537)	11,787	34,339	2,831
DipSummaryP (OOB #3)	7,229	28,782	N/A	N/A	11,935	40,611	3,246
DhvHSumP (NULL pointer)	N/A	N/A	4,102 (9,770,648)	N/A	20,743	41,960	3,090
DhvSummaryP (NULL pointer)	N/A	N/A	63 (63)	151 (88,810,681)	11,187	11,293	685
DhvVBitP (NULL pointer)	N/A	N/A	1,231 (10,067,253)	728 (1,232,914,546)	65,557	47,362	3,534
DhvVBitP (OOB)	19,565	23,354	2,880 (593,827)	140 (26,536,472)	10,607	7,384	2,891

Table 3: Comparing the bug-finding ability of TOSSIM and T-Check in model checking and random execution modes, for two-node and eight-node topologies, for each safety bug that we found. The primary metric is the length of the trace to the bug; for model checking results we also give the total number of events explored, in parentheses. "N/A" means the bug could not be found.

Applications	Hardware Only	Simulation Only	Both
TestSerial	chips/atm128(2/307) chips/atm128/timer(2/252) chips/atm128/pins(2/42) system(2/168)	chips/atm128/sim(2/397) chips/atm128/timer/sim(2/564) chips/atm128/pins/sim(1/24) lib/tossim(2/178)	lib/serial(5/1083), lib/timer(4/267) chips/atm128/timer(1/160) apps/tests/TestSerial(1/75) system(1/101), platforms(2/43)
MultihopOscilloscope	chips/atm128/adc(2/100) chips/atm128/timer(7/513) chips/atm128(2/307) chips/atm128/pins(5/141) chips/cc2420(12/2494) system(6/331) platforms(2/89) lib/timer(2/168)	chips/atm128/adc/sim(2/132) chips/atm128/timer/sim(4/748) chips/atm128/sim(2/397) chips/atm128/pins/sim(1/24) lib/tossim(4/980) lib/tossim(3/216)	lib/net/ctp(7/1375) lib/serial(5/1083) chips/atm128/adc(1/85), system(10/677) chips/atm128/timer(1/160) apps/MultihopOscilloscope(1/185) lib/net/4bitle(1/496) platforms(2/43) lib/timer(3/200)

TestDissemination	chips/atm128/timer(6/380)
	chips/atm128/pins(5/141)
	chips/cc2420(12/2494)
	system(6/331)
	platforms(2/89)
	lib/timer(2/168)
		chips/atm128/timer/sim(1/415)
		chips/atm128/pins/sim(1/24)
		lib/tossim(4/980)
		lib/tossim(3/216)
			lib/net/drip(4/249), lib/timer(3/200)
			chips/atm128/timer(1/160)
			system(6/384)
			TrickleTimerMilliC.nc(1/179)
			platforms(2/43)
			apps/tests/TestDissemination(1/70)

TestDip	chips/atm128/timer(6/380)
	chips/atm128/pins(5/141)
	chips/cc2420(12/2494)
	system(6/331)
	platforms(2/89)
	lib/timer(2/168)
		chips/atm128/timer/sim(1/415)
		chips/atm128/pins/sim(1/24)
		lib/tossim(4/980)
		lib/tossim(3/216)
			lib/net/dip(8/898), lib/timer(3/200)
			chips/atm128/timer(1/160)
			system(6/384)
			TrickleTimerMilliC.nc(1/179)
			platforms(2/43)
			apps/tests/TestDip(1/400)

TestDhv	chips/atm128/timer(6/380)
	chips/atm128/pins(5/141)
	chips/cc2420(12/2494)
	system(6/331)
	platforms(2/89)
	lib/timer(2/168)
		chips/atm128/timer/sim(1/415)
		chips/atm128/pins/sim(1/24)
		lib/tossim(4/980)
		lib/tossim(3/216)
			lib/net/dhv(10/1205), lib/timer(3/200)
			chips/atm128/timer(1/160)
			system(6/384)
			TrickleTimerMilliC.nc(1/179)
			platforms(2/43)
			apps/tests/TestDhv(1/406)

TestFtsp	chips/atm128/timer(6/380) chips/atm128/pins(5/141) chips/cc2420(13/2671) system(6/331) platforms(2/89) lib/timer(1/101)	chips/atm128/timer/sim(1/415) chips/atm128/pins/sim(1/24) lib/tossim(5/1229) lib/tossim(3/216)	lib/ftsp(1/399), lib/timer(4/267) chips/atm128/timer(1/160) system(6/384) platforms(2/43) apps/tests/TestFtsp(1/53)

Table 4: Summary of the number of modules, and total lines of code, from various directories in the TinyOS tree. "Hardware-only" modules do not run in TOSSIM and are not checked by T-Check. "Simulation-only" modules do not run on mote platforms. The remaining modules run on motes and in TOSSIM/T-Check.

[3] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *Intl. Journal on Software Tools for Technology Transfer*, 9(5–6), October 2007.

[4] Qing Cao, Tarek Abdelzaher, John Stankovic, Kamin Whitehouse, and Liqian Luo. Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks. In *Proc. of the 6th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Raleigh, NC, USA, November 2008.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8:244–263, April 1986.

[6] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proc. of the 16th European Symp. on Programming (ESOP)*, Braga, Portugal, March–April 2007.

[7] Nathan Cooprider, William Archer, Eric Eide, David Gay, and John Regehr. Efficient memory safety for TinyOS. In *Proc. of the 5th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 205–218, Sydney, Australia, November 2007.

[8] David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, CA, June 2003.

[9] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *Proc. of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Berkeley, CA, USA, November 2009.

[10] Patrice Godefroid. Model checking for programming languages using Verisoft. In *Proc. of the Symp. on Principles of Programming Languages*, pages 174–186, Nice, France, January 1997.

[11] Ben Greenstein and Philip Levis. TinyOS Extension Proposal (TEP) 113: Serial Communication, 2006. `http://www.tinyos.net/tinyos-2.x/doc/html/tep113.html`.

[12] Alex Groce and Rajeev Joshi. Random testing and model checking: Building a common framework for nondeterministic exploration. In *Proc. of the 6th Intl. Workshop on Dynamic Analysis (WODA)*, Seattle, WA, USA, July 2008.

[13] Lin Gu and John A. Stankovic. t-kernel: Providing reliable OS support to wireless sensor networks. In *Proc. of the 4th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Boulder, CO, November 2006.

[14] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *Intl. Journal on Software Tools for Technology Transfer*, 2(4), March 2000.

[15] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, Cambridge, MA, November 2000.

[16] Mohammad Maifi Khan, Tarek Abdelzaher, and Kamal Kant Gupta. Towards diagnostic simulation in sensor networks. In *Proc. of the Intl. Conf. on Distributed Computing in Sensor Systems (DCOSS)*, Santorini Island, Greece, June 2008.

[17] Mohammad Maifi Khan, Hieu Khac Le, Hossein Ahmadi, Tarek F. Abdelzaher, and Jiawei Han. Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In *Proc. of the 6th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Raleigh, NC, USA, November 2008.

[18] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Detecting liveness bugs in systems code. In *Proc. of the 4th Symp. on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, April 2007.

[19] Nupur Kothari, Todd Millstein, and Ramesh Govindan. Deriving state machines from TinyOS programs using symbolic execution. In *Proc. of the 7th Intl. Conf. on Information Processing in Sensor Networks (IPSN 2008)*, St. Louis, MO, 2008.

[20] Veljko Krunic, Eric Trumpler, and Richard Han. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *Proc. of the 5th International Conference on Mobile Systems, Applications, and Services (Mobisys)*, San Juan, Puerto Rico, June 2007.

[21] Ram Kumar, Eddie Kohler, and Mani Srivastava. Harbor: software-based memory protection for sensor nodes. In *Proc. of the 6th Intl. Conf. on Information Processing in Sensor Networks (IPSN07)*, Cambridge, MA, USA, 2007.

[22] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. of the 1st ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 126–137, Los Angeles, CA, November 2003.

[23] Kaisen Lin and Philip Levis. Data discovery and dissemination with DIP. In *Proc. of the 7th Intl. Conf. on Information Processing in Sensor Networks (IPSN08)*, pages 433–444, St. Louis, MO, USA, April 2008.

[24] Liqian Luo, Tian He, Gang Zhou, Lin Gu, Tarek F. Abdelzaher, and John A. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with EnviroLog. In *Proc. of the 25th Conf. on Computer Communications (INFOCOM)*, Barcelona, Spain, April 2006.

[25] Moteiv. Telos rev. B datasheet, 2005. `http://www.moteiv.com`.

[26] Nguyet T. M. Nguyen and Mary Lou Soffa. Program representations for testing wireless sensor network applications. In *Proc. of the Workshop on Domain Specific Approaches to Software Test Automation (DoSTA'07)*, Dubrovnik, Croatia, 2007.

[27] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proc. of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Stockholm, Sweden, April 2010.

[28] Bastian Schlich. *Model Checking of Software for Microcontrollers.* Dissertation, RWTH Aachen University, Aachen, Germany, June 2008.

[29] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, Berkeley, CA, USA, November 2006.

[30] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.