Bug-Free Sensors: The Automatic Verification of Context-Aware TinyOS Applications

Doina Bucur and Marta Kwiatkowska

Computing Laboratory, Oxford University, UK
{doina.bucur,marta.kwiatkowska}@comlab.ox.ac.uk

Abstract. We provide the first tool for verifying the logic of context-aware applications written for the mainstream sensor network operating system TinyOS; we focus on detecting programming errors related to incorrect adaptation to context.

1 Introduction

Intelligent ambients often include *sensors* in their designs; sensor hardware reads the current value of a specific *context*, such as a patient's cardiogram, skin temperature, blood pressure and oxygen saturation. The decision-making is then left to the software, which infers decisions based on contextual inputs, actuates in output, and is required to gracefully adapt to highly dynamic contexts, in *context-aware* fashion.

For the mainstream sensor operating system TinyOS [1], a programmer writes concurrent, shared-memory software in either nesC or the recent C *TosThreads* API [2], with both languages inheriting C's low-level features. Such software runs in an execution environment in which there is no user-kernel boundary and no guards against *memory violations*. Much more elusive, *concurrency bugs* arise because of the nondeterministic thread interleavings, while *context-awareness bugs* are due to the application's inability to deal with unexpected context. All bug categories can render a deployed sensor node unusable.

To prevent the occurence of context-awareness and concurrency bugs in sensor software *before* deployment time, we provide the very first verification tool for multithreaded, adaptive TinyOS 2.x applications written in TinyOS's C TosThreads API. We *statically verify* a TinyOS application running on a sensor node against a *context-aware safety specification* requiring the program to be in a "safe" state w.r.t actuation and memory configuration, given a—possibly nondeterministic—pattern of incoming context data.

The method our tool employs is *scalable*: We verify a given application modularly, by extracting it from the rest of the TinyOS kernel, and replacing the latter with interface-preserving models. While this requires reviewing the TinyOS code base to learn the semantics of all system calls, the method is good value for developers: it only needs to be provided once, is reusable by all applications and (given that TinyOS is fairly stable) requires little maintenance over time.

Our tool builds on SATABS [3], a generic software verification tool for ANSI C; SATABS takes specifications written as user-specified assertions of boolean

M. Tscheligi et al. (Eds.): AmI 2009, LNCS 5859, pp. 101–105, 2009.

conditions inserted in the code. The verification is *sound* (and *complete* for finite-state applications): The program's state space is exhaustively explored for violations of the specification, including e.g. behaviours triggered by unexpected, but possible, events such as scrambled incoming network packets. An execution trace is returned as a bug witness, allowing the programmer to correct the fault before deploying the application.

We (i) add native support for the C TosThreads API to SATABS, (ii) implement a SATABS-readable C model of the TinyOS system calls to stand in for the OS kernel, and finally (iii) verify application and kernel model against context-aware safety specifications written as SATABS assertions. We report benchmarks on running our tool on standard applications distributed with TinyOS's sources, and on a more complex healthcare application; we find routine violations of safety requirements in staple TinyOS code.

2 The Automatic Verification of TinyOS Applications

This section presents our verification method. We first overview TinyOS and the structure of a TinyOS application, which then allows us to underline possible sources of TinyOS software bugs. Finally, we assess performance with a set of benchmarks and point to the cause and nature of the bugs found.

Modelling the TinyOS Kernel

A TinyOS application programmed in the TosThreads C API [2], as depicted in Fig. 1, is tightly connected to the rest of the operating system's kernel by calling kernel services; these either manage execution scheduling (e.g. thread creation

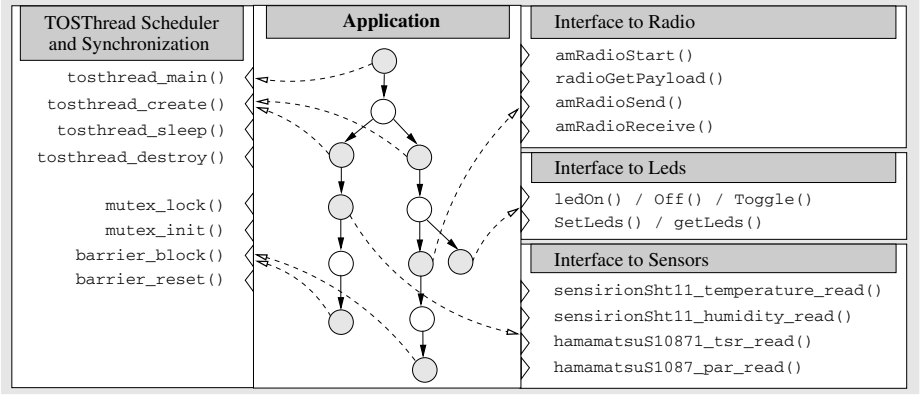

Fig. 1. A TinyOS application programmed in the TosThreads C API, calling TinyOS kernel services, as available on a Tmote Sky sensor node with integrated sensors for temperature, humidity and light intensity

and suspending) or access hardware (e.g. the radio port, sensor chips or resident leds) on their respective software interfaces.

When analysing all possible execution traces of a program to check for violations of a specification, the complexity of the analysis increases with the number of program instructions; when the program is concurrent, the complexity increase is exponential, due to having to explore all possible context switches between threads (a problem known as *state explosion*). Given this fact, we choose to verify a TinyOS application modularly: instead of analysing the application code together with the existing, sizable implementation of the kernel services from Fig. 1, we *model* these services, ensuring that their interface behaviour is preserved; e.g., if `amRadioSend([..])` can fail returning `EBUSY`, so can its model.

Categories of Bugs in Context-Aware Software

In addition to generic memory violation issues, a concurrent context-aware program will also exhibit additional programming errors, which are the focus of our verification. We categorise these from two points of view: generic concurrency (as in Table 2) and correct awareness of context (Table 1). One particular bug can be seen from both viewpoints: an application blocked in waiting for a network packet from a network neighbour which unexpectedly moved away exhibits a bug categorized both as a *network exception* and as *deadlock*.

Table 1. Categories of bugs in context-aware, TinyOS applications

Sensing exceptions Incomplete treatment of sensing errors.
Network exceptions Incomplete treatment of network errors.
Interface use Incorrect use of interface to kernel services.
False reasoning Incorrect decision-making given a context situation.

Table 2. Categories of bugs in generic concurrent software

Data race Multithreaded (write) access to shared resource. Not necessarily a bug.
Atomicity violation Failure to enforce the atomicity of a code region.
Order violation Failure to enforce execution order between two code regions.
Deadlock A thread's failure to release a lock-like resource, halting execution.

Case Studies

We first look at *SenseAndSend*, the staple monitoring application in the TinyOS source tree. Four working threads monitor the Tmote Sky on-board sensors, and each write a fresh value in the data field of a network message; a fifth thread sends the message on the radio. The program's specification states that such messages should be sent periodically, containing valid readings, and accompanied by led signalling. The *claims* are specifications written in assertion form.

We give the results of our verification runs in Table 3. LOC stands for Lines of Code in the application's control-flow graph, including the kernel model; whenever a bug exists, it is categorised as in Tables 1,2. Claim 79 uncovers a misuse of the radio interface; thread 0 fails to ensure that the radio is turned on by not checking `amRadioStart`'s returned error code, before a call to `amRadioSend`:

Table 3. Verification benchmarks. *Blink* is a simple led actuating application; we give its run for comparison. *SenseAndSend* is an application monitoring on-board sensors; *PatientNode* is an extension including distributed monitoring. Verification times are given for runs on a Mac OS X with a 2.4GHz Duo Intel Core and 2GB RAM.

Application (Threads/LOC)	Claim line	Verified?	Time	Bug: context awareness	Bug: concurrency
Blink 4/64	66	yes	2.9s	-	-
SenseAndSend 6/347	79	**no**	32.2s	interface use	order violation
	136	**no**	1m08s	sensing exception	-
	146	yes	4m25s	-	-
PatientNode 6/439	172	yes	29.9s	(interface use)	(order violation)
	254	yes	3m55s	(sensing exception)	-
	230	**no**	35m07s	network exception	deadlock
	268	yes	2m38s	(false reasoning)	-
	262	yes	61m12s	(false reasoning)	-

```
amRadioStart();                                    // thread 0, main
if(amRadioSend(AM_BROADCAST_ADDR, &send_msg, ..) // thread 5, sending
```

More importantly, claim 136 detects that a radio message could be sent with an outdated (temperature) reading. This is the case when the memory location in which the sensing call stores its reading (`sensor_data->temp`) is considered valid even if the sensing call itself failed, and no reading was written in:

```
sensor_data = radioGetPayload(&send_msg, [..]);  // thread 0, main
read_sensor(sensirionSht11_temperature_read,     // thread 2, sensing
            &(sensor_data->temp));
amRadioSend(AM_BROADCAST_ADDR, &send_msg, [..]); // thread 5, sending
```

PatientNode is a SenseAndSend extension tailored for monitoring patients in a pervasive healthcare wireless network [4]. A number of biosensors monitor each patient; a PatientNode application resident on one such sensor collects readings from all of the patient's sensors, sends them in a network message, and signals an abnormal condition by a lit-led configuration. Claims 172 and 254 are re-verifications of SenseAndSend's corrected bugs; claim 230 uncovers that a misplaced closing brace brings the program into a deadlock on a barrier, if a message expected to be received doesn't show up:

```
if(amRadioReceive(&recv_msg, [..]) == SUCCESS) { // thread 3, receiving
  barrier_block(&send_barrier);
}
barrier_block(&send_barrier);                    // thread 5, sending
barrier_reset(&send_barrier, [..]);
amRadioSend(AM_BROADCAST_ADDR, &send_msg, [..]);
```

Claims 268 and 262 verify application logic: the first, that an abnormal received reading is treated as a false alarm if it is not confirmed by a subsequent reception,

and the second, that the maximum span of time between outgoing packets is bounded, regardless of the contents of incoming packets.

3 Related Work and Conclusions

While we know of no verification tools for context-aware specifications, there exist tools which bring a degree of memory and interface-use safety to existing sensor code. TinyOS's own nesC compiler has a built-in simplistic data-race detector. *Safe TinyOS* [5] is an established TinyOS extension which enforces memory safety *at runtime*. It checks e.g. dereferencing null pointers and buffer overflows, which SATABS also provides to our tool; e.g. the verification of a null-pointer claim for PatientNode takes 24.5s. The *interface contracts* [6] act like our own checks of interface use, only at runtime. Both differ from our tool in providing safety at runtime—when the sensor node is possibly deployed out of reach.

Our contribution is a tool for the verification of the logic of context-aware programs written in TinyOS 2.x's C API. It supports complex program features such as dynamic thread creation and ANSI-C pointer use; the program is statically checked before deployment, ensuring reliability to a greater extent than simply verifying against memory violations and interface use.

We keep the verification runs reasonably short by analysing our case studies scalably, against an informed model of the TinyOS kernel services; this precludes the need to explore the execution traces of the entire kernel, monolithically, and gives our method scalability. Finally, in the longer term, we aim at (i) extending this method to nesC, and (ii) verifying *networks* of sensor nodes, by checking individual nodes against specifications in assume-guarantee style.

Acknowledgments. The authors are supported by the project *UbiVal: Fundamental Approaches to Validation of Ubiquitous Computing Applications and Infrastructures*, EPSRC grant EP/D076625/2.

References

1. TinyOS: An Open-Source Operating System for the Networked Sensor Regime: `http://www.tinyos.net/` (accessed August 2009)
2. Klues, K., Liang, C.J., Paek, J., Musăloiu, R., Govindan, R., Terzis, A., Levis, P.: TOSThreads: Safe and Non-Invasive Preemption in TinyOS. In: SenSys (2009)
3. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
4. Varshney, U.: Pervasive healthcare and wireless health monitoring. Mobile Networks and Applications (MONET) 12(2-3), 113–127 (2007)
5. Cooprider, N., Archer, W., Eide, E., Gay, D., Regehr, J.: Efficient Memory Safety for TinyOS. In: ACM SenSys, pp. 205–218 (2007)
6. Archer, W., Levis, P., Regehr, J.: Interface Contracts for TinyOS. In: Information Processing in Sensor Networks (IPSN), pp. 158–165. ACM Press, New York (2007)