Towards Software Verification for TinyOS Applications

Doina Bucur and Marta Kwiatkowska
Overview

- Sensor networks
- Context-aware sensor software
- Context-aware safety specification
- Verification
Sensor networks

- Pervasive Healthcare. Body Sensor Networks
  [vip.doc.ic.ac.uk/bsn](http://vip.doc.ic.ac.uk/bsn)

- Sensor architecture / OS. TinyOS:
  - Modern OS and language in an embedded system
Software verification

- Verification vs. simulation / formal verification
- Advances in software verification
- Counterexample-guided abstraction refinement (CEGAR)
- CProver tools [cprover.org]
A TinyOS application

Application (C, nesC)

Kernel (nesC)

Drivers

Platform-specific drivers
Architectures

CPU

MSP430
(Texas Instruments)

AVR
(Atmel)

ARM

Telos
TelosB / Tmote Sky
TinyNode
eyesIFX

Mica2
MicaZ
Mica2Dot
Imote2

SunSpot
...and the software tool chain

Components (threaded, C, nesC)
Platform-specific inlined program (sequential, C + asm)

Machine code

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

with Marta Kwiatkowska

Proceedings of the European Conference on Ambient Intelligence (Aml 2009)
Springer Lecture Notes in Computer Science (LNCS), Nov 2009
Our tool builds on SATABS [3], a generic software verification tool for ANSI C; SATABS takes specifications written as user-specified assertions of boolean 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.
Our tool builds on SATABS [3], a generic software verification tool for ANSI C; SATABS takes specifications written as user-specified assertions of boolean 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.

### 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

![Diagram of TinyOS application](image)

Model kernel services, ensuring that their interface behaviour is preserved.
Programming errors and benchmarks

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensing exceptions</td>
<td>Incomplete treatment of sensing errors.</td>
</tr>
<tr>
<td>Network exceptions</td>
<td>Incomplete treatment of network errors.</td>
</tr>
<tr>
<td>Interface use</td>
<td>Incorrect use of interface to kernel services.</td>
</tr>
<tr>
<td>False reasoning</td>
<td>Incorrect decision-making given a context situation.</td>
</tr>
</tbody>
</table>

Table 2. Categories of bugs in generic concurrent software

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data race</td>
<td>Multithreaded (write) access to shared resource. Not necessarily a bug.</td>
</tr>
<tr>
<td>Atomicity violation</td>
<td>Failure to enforce the atomicity of a code region.</td>
</tr>
<tr>
<td>Order violation</td>
<td>Failure to enforce execution order between two code regions.</td>
</tr>
<tr>
<td>Deadlock</td>
<td>A thread’s failure to release a lock-like resource, halting execution.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Blink 4/64</td>
<td>66</td>
<td>yes</td>
<td>2.9s</td>
<td>-</td>
</tr>
<tr>
<td>SenseAndSend 6/347</td>
<td>79</td>
<td>no</td>
<td>32.2s interface use</td>
<td>order violation</td>
</tr>
<tr>
<td></td>
<td>136</td>
<td>no</td>
<td>1m08s sensing exception</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>146</td>
<td>yes</td>
<td>4m25s</td>
<td>-</td>
</tr>
<tr>
<td>PatientNode 6/439</td>
<td>172</td>
<td>yes</td>
<td>29.9s (interface use)</td>
<td>(order violation)</td>
</tr>
<tr>
<td></td>
<td>254</td>
<td>yes</td>
<td>3m55s (sensing exception)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>230</td>
<td>no</td>
<td>35m07s network exception</td>
<td>deadlock</td>
</tr>
<tr>
<td></td>
<td>268</td>
<td>yes</td>
<td>2m38s (false reasoning)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>262</td>
<td>yes</td>
<td>61m12s (false reasoning)</td>
<td>-</td>
</tr>
</tbody>
</table>
Rely-Guarantee Reasoning for Context-Aware Software

with Marta Kwiatkowska

In preparation.
New safety specifications

- CProver specification style: assertions

```c
sense_power(&ctx) // sensing
...
if(power >= MIN_PWR) Led0On(); // actuation
...
259: assert(__red_led_on);
```

- We want context-aware assertions:

```c
sense_power : Trig(Led0On)
```

... and thread-scalable verification.
A typing system

Side-effect and escape analysis

\[
\Gamma_m ::= \{ \text{ctx} : \rightarrow \text{sense\_power}_1 \} \\
\Gamma_1 ::= \{ \text{arg}_1 \rightarrow \text{sense\_power}_1 \}
\]

if (power >= MIN_PWR) Led0On();
if (power >= MIN_PWR) Led0On() // true?
A Rely-Guarantee logic --

- over “sensor” language:

\[(S_{MT}) \quad S ::= \text{sense}(\&x) \mid \text{actuate}(x) \mid \ldots\]

- with thread-local rely/guarantee actions:

\[(A_{CTION}) \quad R, G ::= [p] \mid p \times q \mid \exists X.R \mid R \lor R \mid R \land R \mid R \Rightarrow R \mid R * R\]

- ... and assertions:

\[(A_{SSERTION}) \quad p, q, r ::= \text{emp}_s \mid \text{Own}(x) \mid B \mid \exists X.p \mid \text{emp}_\alpha \mid \text{Own}(a) \mid \mathcal{P}(\text{Contexts}) \mid \text{Trig}(a) \mid a \mapsto n \mid p \lor q \mid p \land q \mid p \Rightarrow q \mid p * q\]
A thread’s verification

\[ \Gamma_m \lor \Gamma_1 \lor \Gamma_3 \land \Gamma_2; R_2, G_2 \vdash \]

\[ p_c \land p_4 \]

\[ \langle \text{arg}_2 := \text{ctx}2; \rangle \]

\[ p_{21} := p_c \land (\text{arg}_2 = Y) \]

\[ \langle \text{IF }!(\text{power} > \text{MIN}) \text{ EXIT} \rangle \]

\[ p_{22} := p_{21} \land (M > \text{MIN}) \]

\[ \langle \text{out} := \text{DB sel arg}_2; \rangle \]

\[ p_{23} := p_{22} \land (\text{out} = \text{DB sel Y}) \]

\[ \langle \text{IF }\text{power} > \text{MIN} \text{ display(out);} \rangle \]

\[ p_{23} \land q \]

16