Poster Abstract: Software Verification for TinyOS

Doina Bucur
University of Oxford, UK
doina.bucur@comlab.ox.ac.uk

Marta Z. Kwiatkowska
University of Oxford, UK
marta.kwiatkowska@comlab.ox.ac.uk

ABSTRACT

We describe the first software tool for the *verification* of TinyOS 2, MSP430 applications at *compile-time*. Given *assertions* upon the state of the sensor node, the tool boundedly explores all program executions and returns to the programmer an error trace leading to any assertion violation. Besides memory-related errors (out-of-bounds arrays, null-pointer dereferences), we verify application-specific assertions, including low-level assertions upon the state of the registers and peripherals.

Categories and Subject Descriptors

I.2.9 [**Artificial Intelligence**]: Robotics—*Sensors*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; D.4.5 [**Operating Systems**]: Reliability—*Verification*

General Terms

Reliability, Verification, Languages

Keywords

Sensor networks, TinyOS, Telos, MSP430, software verification, model checking, reliability, safety

1. VERIFICATION FOR TINYOS

Reliable sensor software is difficult to program. Interrupt-driven code has unrestricted access to the microcontroller's mapped memory, and a sensor node functions in a dynamic, error-prone network: the programmer has to account for context switches to interrupt handlers, and for (potentially corrupted) network data.

We specialize the state of the art in *software verification* for ANSI C to be used for TinyOS on MSP430 [1] platforms such as TelosB [2]; we instrument TinyOS code with *assertions* which should hold whenever they are reached, and input the result into a fully-automated verification tool chain which returns a program trace to an assertion violation.

Thus far, the task of locating programming errors in a sensor application traditionally was done by:

- *debugging by deployment*, with blinking LEDs allowing little visibility into the fault's cause;
- *runtime recovery tools* such as SafeTinyOS [3], with the disadvantage that, for each error, the deployed application has to recover by, e.g., reboot;
- simulation.

The difference between simulation and verification is that, while simulation unwinds one program trace, and thus may not detect an undesirable program feature, verification aims to inspect all executions. We use the bounded form of verification, which unwinds program loops up to a finite bound, translates the resulting state machine and any assertions into mathematical models (e.g., Boolean formulas), and checks whether the conjunction of the program's formula and an assertion's negated formula is satisfiable—in which case, there exists a trace on which the assertion is violated.

As formal verification is not *complete* for infinite-state programs (i.e., cannot unwind and verify the entire state-space), and doesn't *scale* as well, we intend our automated verification tool chain not to replace, but to complement runtime techniques and simulation. However, it statically analyses real source code, and guarantees to detect all errors up to a finite depth.

We depict our tool chain in Fig. 1. The `nescc` compiler first generates an inlined, MSP430-specific C program (for deployment, this is intended to be passed to the platform's `mspgcc` compiler [4] to generate a binary). Instead of generating a binary, this is passed to our own tool, `tos2cprover`, which:

- Does a source-to-source transformation to give the program high-level C semantics; the memory map is modelled by new global variables; e.g., variable `_P5OUT` is now the output register for peripheral port 5.
- Determines from function attributes (e.g. attribute `__attribute((interrupt(14)))` for the 12-bit Analog-to-Digital Converter `ADC`) which functions are IRQ handlers, then instruments the program so that IRQ calls are made whenever interrupts are enabled. A *partial order reduction* technique is used to reduce the number of these calls.

The resulting C code is a precise, but high-level model of the original embedded code, and serves as input to CBMC [5], a software verification tool for ANSI C from the CProver suite; this further instruments the code with assertions against standard errors (e.g., memory violations), and *boundedly* unwinds the program's loops to report violations occurring within the unwinding limit.

IPSN'10, April 12–16, 2010, Stockholm, Sweden.
ACM 978-1-60558-955-8/10/04.

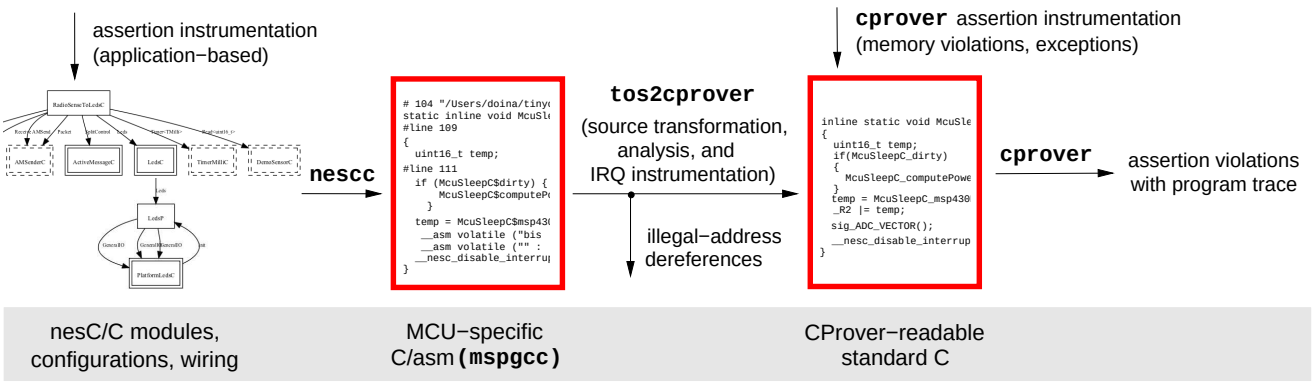

Figure 1: The verification tool chain

2. BENEFITS AND COSTS

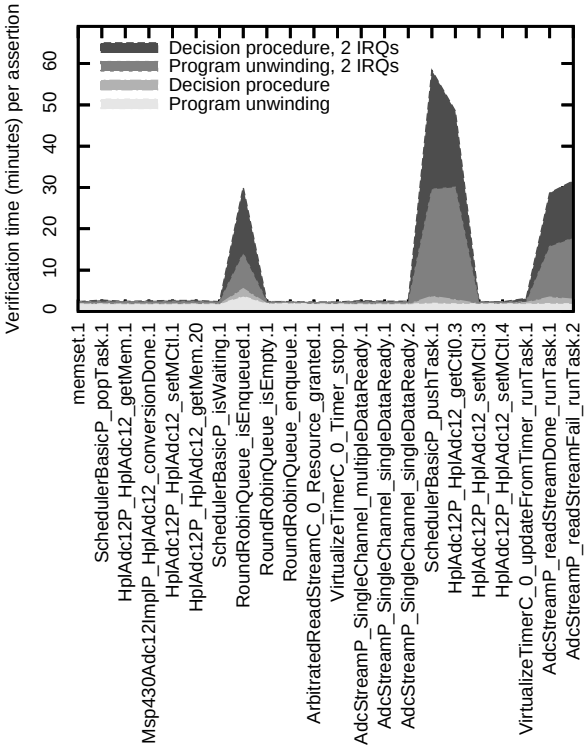

Figure 2: Verification times for *Sense*

Our *assertions* can be either hardware-aware, e.g.

```
assert(_P5OUT & 0x0008);
```

or high-level (e.g., asserting upon the value of a variable local to a component). Implicit assertions guarding buffer bounds, pointer use and arithmetic exceptions are introduced by CBMC. E.g., as function `pushTask(uint8_t id)` writes upon the task queue:

```
SchedulerBasicP_m_next[SchedulerBasicP_m_tail]=id;
```

CBMC will generate the upper-bound assertion:

```
Claim SchedulerBasicP_pushTask.1:
  (unsigned int)SchedulerBasicP_m_tail < 8
```

For the *Sense* application in the TinyOS `apps` repository, 132 memory-violation assertions are thus generated, with 747 for *TestDissemination*. Their verification runs came up negative, when unwinding two task scheduler loops and allowing one interrupt per loop.

The verification time sums the *program unwinding time* and the *decision procedure runtime* of the SAT solver on the program's formula. Fig. 2 exemplifies the runtimes for a subset of memory-violation assertions from *Sense*. The x axis is labeled with assertion identifiers: e.g., `SchedulerBasicP_pushTask.1` is the first assertion generated inside function `pushTask` from component `SchedulerBasicP`. Both runs are configured with one IRQ call per task loop; one run unwinds one task loop, and another, two. With notable exceptions for which the runtime explodes, most assertions are verified in constant time. While we did not discover any new errors in TinyOS, we were able to inject and detect known bugs.

Acknowledgments

We are indebted to Răzvan Musăloiu-E and Daniel Kroening, and are supported by the project *UbiVal: Fundamental Approaches to Validation of Ubiquitous Computing Applications and Infrastructures*, EPSRC grant EP/D076625/2.

3. REFERENCES

[1] Texas Instruments. MSP430x1xx family – user's guide (Rev. F). `www.ti.com`, 2006 (accessed 02/2010).

[2] Moteiv Corporation. Telos. Ultra low power IEEE 802.15.4 compliant wireless sensor module. Revision B: Humidity, Light, and Temperature sensors with USB. `http://www.moteiv.com`, 2004 (accessed 02/2010).

[3] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proc. of the Conference on Embedded Networked Sensor Systems (SenSys)*, pages 205–218. ACM, 2007.

[4] S. Underwood. Mspgcc—A port of the GNU tools to the Texas Instruments MSP430 microcontrollers. `http://mspgcc.sourceforge.net/manual`, 2003 (accessed 02/2010).

[5] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of LNCS, pages 168–176. Springer, 2004.