On Software Verification for Sensor Nodes

Doina Bucur[a], Marta Kwiatkowska[b]

[a]*Oxford University Computing Laboratory, Oxford, OX1 3QD, UK*
[b]*Oxford University Computing Laboratory, Oxford, OX1 3QD, UK*

Abstract

We consider software written for networked, wireless sensor nodes, and specialize software verification techniques for standard C programs in order to locate programming errors in sensor applications before the software's deployment on motes. Ensuring the reliability of sensor applications is challenging: low-level, interrupt-driven code runs without memory protection in dynamic environments. The difficulties lie with (i) being able to automatically extract standard C models out of the particular flavours of embedded C used in sensor programming solutions, and (ii) decreasing the resulting program's state space to a degree that allows practical verification times.

We contribute a platform-dependent, OS-independent software verification tool for OS-wide programs written in MSP430 embedded C with asynchronous hardware interrupts. Our tool automatically translates the program into standard C by modelling the MCU's memory map and direct memory access. To emulate the existence of hardware interrupts, calls to hardware interrupt handlers are added, and their occurrence is minimized with a double strategy: a partial-order reduction technique, and a supplementary reachability check to reduce overapproximation. This decreases the program's state space, while preserving program semantics. Safety specifications are written as C assertions embedded in the code. The resulting sequential program is then passed to CBMC, a bounded software verifier for sequential ANSI C. Besides standard errors (e.g., out-of-bounds arrays, null-pointer dereferences), this tool chain is able to verify application-specific assertions, including low-level assertions upon the state of the registers and peripherals.

Verification for wireless sensor network applications is an emerging field of research; thus, as a final note, we survey current research on the topic.

Keywords: sensor, TelosB, MSP430, TinyOS, verification, bounded model checking, CBMC

1. Introduction

While small applications for basic embedded systems for a particular microcontroller can be programmed directly in machine code, sensor node platforms are typically equipped with a rather rich set of features, including a radio (and in many cases, also a wired serial) transceiver, sensing chips and external flash memory. Programming from scratch each new application for such sensor platforms is difficult and unmaintainable—be this programming done in either assembly, or the platform's own flavour of embedded C. For example, a basic *Oscilloscope* functionality (i.e., a sensor node periodically sampling a

sensor, then broadcasting a message over the radio every ten readings) in the `elf32-avr` binary form for the MicaZ mote[1] disassembles into over 12×10^3 lines of executable code in its `.text` section alone; similar program sizes are yielded from TelosB[2] MSP430 `elf` files. Programming embedded C instead (for the platforms' C compilers, e.g., `avr-gcc` and `msp430-gcc` (Underwood, 2003)) does not decrease the code's complexity—the application will have roughly the same size[3].
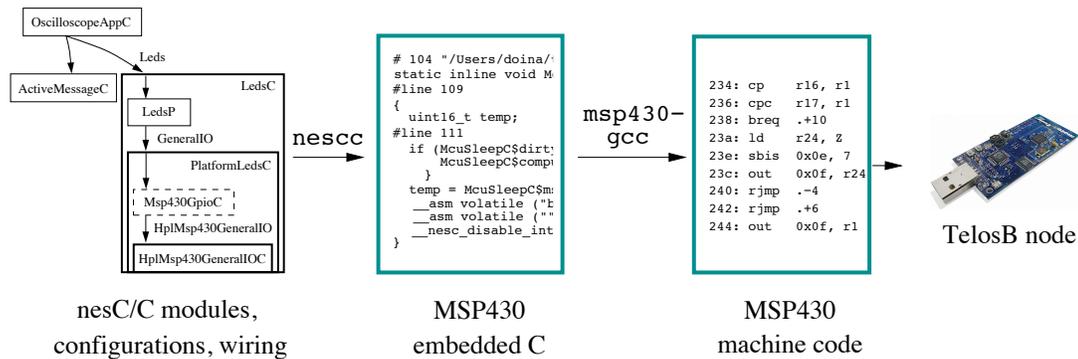

Figure 1: Example TinyOS program compilation for a TelosB mote. The nesC compiler, `nescc`, inlines nesC/C components into MSP430 embedded C; `msp430-gcc` then outputs machine code.

Instead of programming sensor applications from scratch, *operating systems* have been developed for sensor nodes, such as TinyOS (Levis et al., 2005) and Contiki. As a result, typically programmers write application logic in a high-level language, while calling scheduling and driver functionality from the operating system. Fig. 1 overviews TinyOS's tool chain of program compilation for a TelosB mote. The *Oscilloscope* application is a nesC (Gay et al., 2003) or TOSThreads (Klues et al., 2009) component interfacing with existing TinyOS components; the binary program deployable on a TelosB mote is generated by two discrete stages of program translations, from nesC to inlined MSP430 C, to MSP430 machine code.

With this multi-stage style of program compilation in mind, our task is to design *software verification* methods and tools for sensor applications. In this paper, we focus on taking as input platform-specific embedded C, such as that automatically generated from higher-level software components in the chain of compilation from Fig. 1. While we take our case studies from the standard TinyOS applications, the verification method itself is OS-independent.

Software verification per se is a compile-time method which, given a particular program implementation and set of specifications, unwinds and analyses all of the program's traces, and outputs violations of the program's specifications, if any. Specific such tools cater for specific programming (and specification) languages, and most are limited as to the program features they support, e.g.:

- complex data structures are supported by few existing software verifiers,

---

[1] Mica motes are based around the Atmel AVR ATmega128L 8-bit micro-controller (Corp., 2009).

[2] Telos platforms (e.g., TelosB (Moteiv Corporation, 2004)) are built with MSP430 (Texas Instruments, 2006), a 16-bit MCU from Texas Instruments.

[3] These examples of application complexity come from the basic *Oscilloscope* and *MultihopOscilloscope* applications, programmed for the TinyOS (Levis et al., 2005) operating system. Thus, optimizations over these numbers may be feasible.

- programs, particularly concurrent ones, give rise to sets of program states which are too large to be verified within certain time limits.

In what follows, we describe our platform-dependent, OS-independent software verification tool for OS-wide programs (on the order of magnitude of 10^3 LOC) written for `msp430-gcc` with asynchronous hardware interrupts. Our tool automatically translates such a program into standard C by modelling direct memory access and the MCU's memory map in ANSI-C. Calls to hardware interrupt handlers are inserted into the main application to emulate the existence of hardware interrupts, and the number of their occurrences is minimized with a partial-order reduction technique, in order to decrease the program's state space, while fully preserving program semantics. Safety specifications are written as C assertions embedded in the code. The resulting sequential program is then passed to CBMC (Clarke et al., 2004), a bounded software verifier for sequential ANSI C. Besides memory-related errors (e.g., out-of-bounds arrays, null-pointer dereferences), the tool chain verifies application-specific assertions, including low-level assertions upon the state of the registers and peripherals.

We first give background information essential to our approach in Section 2. This includes the syntax and semantics of MSP430 embedded C, an overview of (i) the MSP430 microcontroller, (ii) the TelosB platform, and (iii) CBMC, the software verifier for ANSI C that we specialize for our purpose.

2. Background: Platform and embedded language for sensor nodes. CBMC

A variety of hardware platforms are available as sensor nodes. TelosB motes (Moteiv Corporation, 2004) are based on the 16-bit Texas Instruments MSP430 microcontroller (Texas Instruments, 2006); Mica nodes are built around Atmel's AVR (Corp., 2009), and MITes nodes around Intel's 8051 (Atmel, 2008).

2.1. MSP430, TelosB and msp430-gcc

We consider MSP430, a microcontroller configuration featuring, on a I^2C bus, a 16-bit RISC CPU, 48kB Flash memory (and 10kB RAM), 16-bit registers, two built-in 16-bit timers, a 12-bit analogue-to-digital converter (ADC), two universal serial synchronous/asynchronous communication interfaces (USART), and 64 I/O pins (the latter, together with their connections on a TelosB mote, overviewed in Fig. 2).
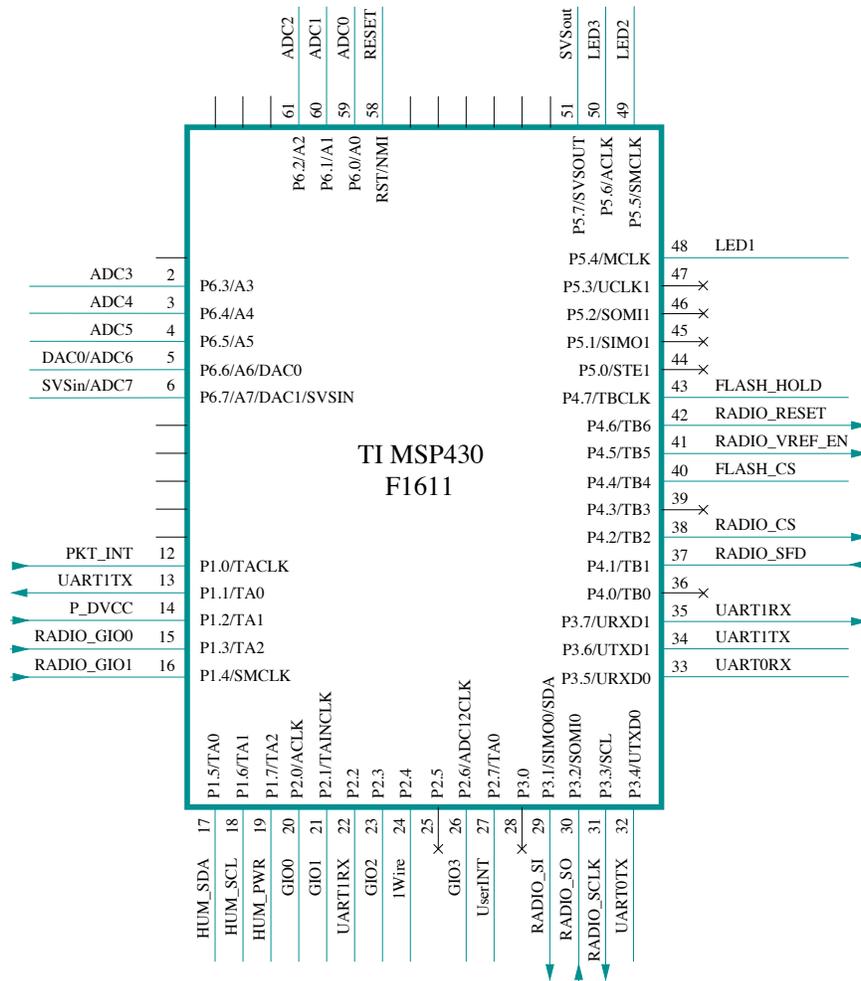
Figure 2: The MSP430 F1611 microcontroller as used on TelosB motes. Selected I/O pins and their connections on TelosB motes. Pins such as those supplying voltage (1 and 62-64) or the two crystal oscillators' input and output ports (8-9, 52-53) are not drawn.

A pin's identifier is three-fold; pin 17, for example, is pin 5 of peripheral port 1 (out of six 8-bit I/O ports), and is a general-purpose digital I/O pin or a Timer A output pin, from the microcontroller's perspective; on TelosB specifically, this pin is connected to the bidirectional serial data port, HUM_SDA, of the TelosB on-board humidity sensor (which produces a digital output). Similarly, pins 32-33 are transmit/receive pins for the first serial port, USART0, and are connected as such to the serial physical port on TelosB. Pins 48-50, or 4 to 6 on peripheral port 5 (general-purpose digital I/O pins, or clock outputs) are instead connected to, and control, the three on-board LEDs.

The software is able to access these pins, together with other peripheral modules and registers, by direct memory access. All on-board memory, including peripherals and the Interrupt Vector Table, are mapped into a unique address space, with Special Function Registers and peripheral modules at low addresses, as in Fig. 3.
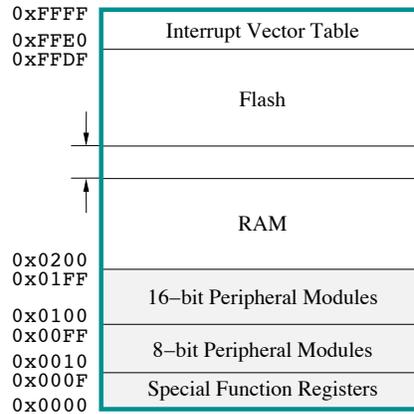
Figure 3: Memory organization for MSP430 microcontrollers.

The MCU's six 8-bit I/O ports whose pins are shown in Fig. 2 are mapped onto the `0x10-0xFF` address space for 8-bit peripheral modules. E.g., the 8-bit output register for port 5 is accessed as `0x0031`; bits 4-6 in this register control the LEDs. Thus, when a `msp430-gcc` (Underwood, 2003) embedded C program states:

```
static volatile uint8_t r __asm ("0x0031"); r ^= 0x01 << 6;
```

or in other words:

```
*(volatile uint8_t *)49U ^= 0x01 << 6;
```

this amounts to toggling a bit in the 8-bit output register `P5OUT` of peripheral port 5 at location `0x0031`, where LEDs are accessed, which toggles the yellow LED. Other essential memory-mapping examples are shown in Table 1.

Addr	Identifier	Description
0x0030	P5IN	Port 5 Input
0x0031	P5OUT	Port 5 Output
0x0074	U0BR0	USART0 Baud Rate 0
0x0075	U0BR1	USART0 Baud Rate 1
0x0076	U0RXBUF	USART0 Receive Buffer
0x0077	U0TXBUF	USART0 Transmit Buffer
0x0140	ADC12MEM[16]	ADC12 Memory 0-15
0x01A2	ADC12MCTL[16]	ADC12 Memory-Control Register 0-15
0x01A0	ADC12CTL0	ADC12 Control Register 0
0x01A2	ADC12CTL1	ADC12 Control Register 1
0x0180	TBCTL	TimerB Control Register
0x0190	TBR	TimerB Count Register
0x0192	TBCCR0-6	TimerB Compare Register 0-6

Table 1: Individual memory addresses for selected peripheral ports and MCU-internal registers.

Similarly, when a function such as `sig_ADC_VECTOR` is declared with the attributes

```
__attribute((wakeup)) __attribute((interrupt(14)))
```

the function is an *interrupt handler* for interrupt line 14 (i.e., it wakes the processor from any low power state as the routine exits).

2.2. TinyOS

TinyOS (Levis et al., 2005) is a mainstream operating system for wireless sensor network devices. We give an overview of TinyOS in order to clarify (i) program structure and (ii) terminology related to the style of concurrency, for our case studies.

The operating systems itself, as well as the applications, are implemented in the *nesC* (network embedded systems C) language (Gay et al., 2003); some new applications are coded in TOSThreads (Klues et al., 2009), TinyOS 2.x's recent thread library. NesC software comes in *components*, either *modules* or *configurations*. Fig. 4 gives a partial overview of *Oscilloscope*, a typical TinyOS nesC application; the LED driver module *LedsC* is given a degree of detail. Components have similarities to objects: they enclose the program's state and interact through *interfaces* (Gay et al., 2005). For efficiency, given the language's focus on embedded systems, an application's components, interfaces and memory use are determined at compile-time; there is no dynamic memory allocation.
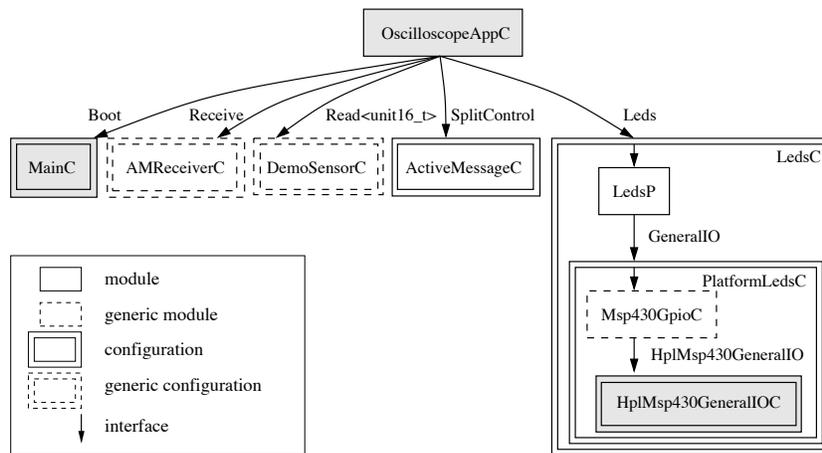

Figure 4: Overview of *Oscilloscope*, a typical TinyOS nesC application. Modules, configurations and interfaces. The *OscilloscopeAppC* module wires existing components such as the *MainC* from which `nescc` generates the program's main function, and platform-specific drivers such as that for LEDs (partly pictured).

All lengthy commands in TinyOS (e.g., the sending of a packet on the radio) are non-blocking; their completion is signalled by an *event* (some of which are triggered by a hardware interrupt), whose handler should be brief, instead posting *tasks* to the system's task queue for further execution. All threads of control in a TinyOS application are thus rooted in either an event handler or a task, in a two-level concurrency model: event handlers run with highest priority and may preempt the lower-priority tasks, which execute most of the program logic. Tasks run to completion relative to each other, however, and *synchronous code* is that which is only reachable from tasks; *asynchronous code* is reachable from at least one event handler. Whenever program variables are accessible to asynchronous code, a potential *data race* ensues.

For a particular application, nesC components are *wired* together through their interfaces to form an OS-wide program; nesC is in fact designed under the expectation that a final, inlined embedded C program will be generated from all the necessary components.

2.3. CBMC

CBMC (Clarke et al., 2004) is a bounded model-checker (Clarke et al., 2001) for ANSI-C programs, from the CProver suite. It takes safety specifications written as C `assert` statements, and is roughly geared

towards verifying embedded systems software. Unlike other C model checkers, CBMC supports a richer subset of the language in what regards data types and data representation, by modelling semantics accurately to the bit level, to the extent that this semantics is defined by the ANSI C standard[4]. Standard *static program analyses* are supported, including alias analysis[5]. Thus, the tool can pinpoint program errors related to bit-level operators and arithmetic overflow, pointer and array operations, arithmetic exceptions, user-inserted assertions and assumptions.

To derive an accurate mathematical representation of an input program, the tool translates the C input into a side-effect-free intermediate representation:

- All side-effect assignments are broken into equivalent statements by introducing auxiliary variables, and all loops (`for`, `while`, backward `goto`s) are *unwound* a user-provided number of times, by replicating the loop body.
- Function calls are inlined and recursive calls are similarly unwound.

The resulting program consists of `if` instructions, assignments, assertions, labels, and forward jumps. This is then translated into *static single assignment form* (SSA), a standard intermediate representation in which every program variable is split into "versions", i.e., a new program variable is invented for each assignment to the original (Fig. 5, center). Frequently used for compiler optimizations, the technique simplifies the analysis of the variables' definition and use.
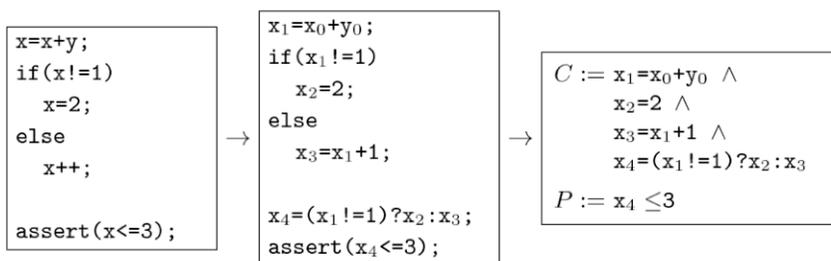

Figure 5: CBMC (Clarke et al., 2004) program transformation into a mathematical model

Two boolean propositional formulas are derived from this program in SSA form: C for the program itself, and P for the asserted expression, as in Fig. 5, right. ANSI-C variables x_i of any data type (including arrays, structures, unions, pointers, and all basic data types) are now replaced with bit-vector variables, and all mathematical operations performed over program variables with bit vector operations, by bit blasting (Kroening and Strichman, 2008). This transformation is word-width adjustable to, e.g., 16 bits, to simulate different hardware platforms.

P is then verified by converting $C \land \neg P$ into conjunctive normal form (CNF) and then passing it to a SAT solver such as MiniSAT (Sorensson and Een, 2005). If this conjuncted formula is satisfiable, there exists a violation of the assertion, and CBMC returns to the programmer a program trace leading to the violation, as a debugging tool would do; otherwise, the assertion holds.

---

[4] This fact results in a number of issues in the case of C code which relies on, e.g., the memory alignment of struct fields, which is not standardized; whenever a verification run depends on non-standard C, CBMC reports the issue and we disregard the run.

[5] The CBMC manual at http://www.cprover.org/cbmc/ gives some detail on the static program analyses included in the tool.

In verifying an ANSI-C program by bounded unwinding, CBMC thus proves a partial guarantee of program properties (i.e., that bugs are absent for a certain amount of unwinding). The process is highly automated and scales reasonably well.

3. TOS2CProver: source-to-source transformation

3.1. Overview

Software bugs stem from flaws both in the legacy OS code base (the lowest levels of which are platform-dependent), and in the programmer's applications. By a *safe* sensor application, we understand that which exhibits no memory violations, and whose programmer-inserted assertions hold, if reachable. Note that a safe application may still contain undiscovered errors, either if they require a greater number of loop unwindings, or if no appropriate assertion was specified. However, our verification approach can detect errors deeper in the program, compared to static analysis.

To achieve a homogeneous verification scheme for both legacy and newly programmed TinyOS code, we contribute, also in (Bucur and Kwiatkowska, 2010), an automated tool chain of program transformation and verification as depicted in Fig. 6.
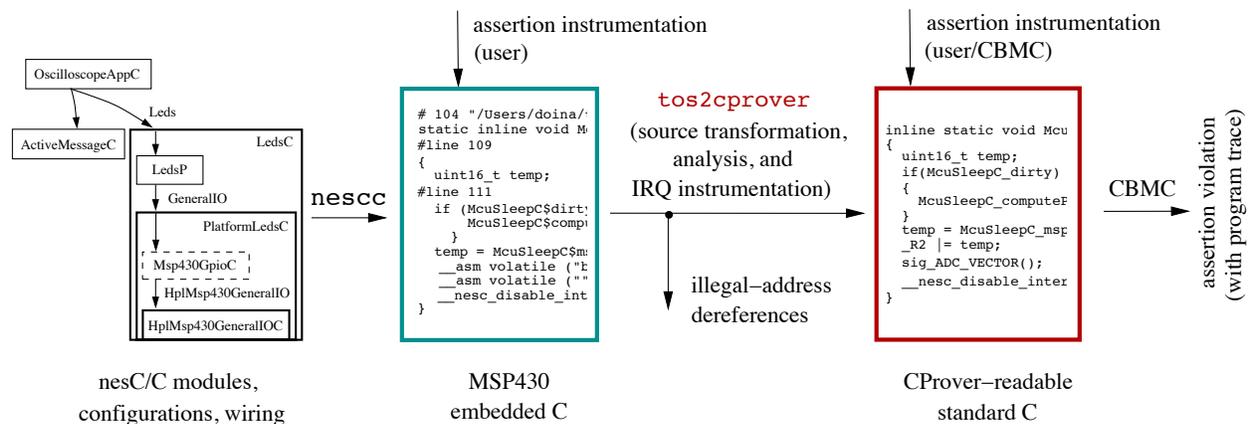

Figure 6: Our verification tool chain

In order for the tool to be independent of the OS-specific programming paradigm, an initial run of TinyOS's `nescc` compiler is used to generate an inlined, platform-specific, embedded C program out of nesC and TOSThreads components. Then, instead of employing the platform's own compiler to further build this into a binary deployable on a mote, the program is passed to our own tool, `tos2cprover`, which performs a dual task:

- In a source-to-source transformation step, it gives a precise ANSI-C model to all low-level, hardware-managing language extensions, and instruments the code so as to emulate the hardware's functionality: whenever a register's value is filled in from the hardware, the program is augmented so as to provide such values.

- Then, `tos2cprover` reads the functions' attributes and determines which functions would be called as IRQ handlers in the event of a hardware interrupt, then instruments the resulting program so that IRQ handlers may be called whenever hardware interrupts are allowed; this way, the code also becomes a hardware emulator. A specialized partial-order reduction (POR)

technique is used to minimize the number of occurrences of such calls. Following these transformation and instrumentation steps, the result is a standard-C program which both soundly preserves the functionality of the initial platform-specific program, and emulates the hardware. The program is instrumented with user-inserted properties (in the form of assertions, assumptions and nondeterministic input), and passed to CBMC (which also introduces a set of assertions) for verification; the unwinding bounds for the program loops are set individually per loop.

In the remainder of this section, we briefly describe `tos2cprover`'s source-transformation step. The instrumentation of the source with calls to IRQ handlers (and the accompanying partial-order reduction technique) are detailed in Section 4.

3.2. TOS2CProver: source-to-source transformation

Table 2 exemplifies the source transformations executed by `tos2cprover` on a TelosB, MSP430 program. While `msp430-gcc` code implicitly assumes an underlying memory map (as overviewed in Section 2) in which low, constant addresses have a defined semantics (e.g., writing at `0x0031` programs the LEDs), `tos2cprover` *models* direct memory access with a header file defining memory as a set of new, global variables. E.g., `uint8_t _P5OUT` is now the 8-bit output register for peripheral port 5. All subsequent dereferences of address `0x0031` are replaced by accesses to `_P5OUT`[6]. As a note, the Status Register `_R2` has the General Interrupt Enable (GIE) as bit 4; if GIE is set, interrupts are enabled.

MSP430-specific program feature	Example `tos2cprover` transformation	
MCU registers and memory map	Standard C global variables	
	`unsigned short _R2;`	(Status Register)
	`unsigned char  _P5OUT;`	(Port 5 at `0x0031` by Table 1)
	`unsigned short _ADC12CTL0;`	(ADC12 Register at `0x01A0`)
	`unsigned short _ADC12MEM[16];`	(ADC12 Memory 0-15)
Fixed-address dereference	Global variable access	
`*(uint8_t*)49U ^= 0x01 << 6;`	`_P5OUT ^= 0x01 << 6;`	
Fixedly allocated variables	Global variable access	
`uint16_t HplAdc12P$ADC12CTL0 __asm("0x01A0");`	(declaration removed; ADC12CTL0 previously declared)	
`HplAdc12P$ADC12CTL0 |= 0x0010;`	`_ADC12CTL0 |= 0x0010;`	
Assembly instructions	C statements	
`__asm volatile ("eint");`	`_R2 |= 0x0008;`	
`__asm volatile ("bis %0, r2"::"m"(temp));`	`_R2 |= temp;`	

Table 2: `tos2cprover`: source-to-source transformation examples for MSP430 code.

Then, `msp430-gcc`'s assembly extensions are modelled into standard C, as are all other non-standard language features (e.g., identifier names are standardized by replacing dollar signs with underscores, `struct` and `union` designated initializers are expanded).

---

[6] Note that the variables we introduce are named in accordance with the MSP430 documentation (Texas Instruments, 2006), but are preceded by an underscore, to avoid name clashes with existing program variables.

4. TOS2CProver: IRQ instrumentation and the partial-order reduction (POR) technique

4.1. Overview

The `nescc`-generated program inputted to `tos2cprover` does not explicitly call any IRQ handlers; in deployments, the calls are made from the hardware. Instead, it defines the functions and marks them as interrupt service routines; e.g., in the case of a TelosB-based *Sense*, two types of hardware interrupts are expected: one from the user timer, TimerB, and another from the 12-bit Analog-to-Digital Converter, ADC. Their handler functions have the signatures:

```
void sig_TIMERB0_VECTOR(void) __attribute((wakeup)) __attribute((interrupt(24)));
void sig_ADC_VECTOR(void)     __attribute((wakeup)) __attribute((interrupt(14)));
```

The size of the asynchronous code (i.e., code reachable from either IRQ handler) is substantial: in *Sense*, out of the 520 reachable functions in the program, 166 are reachable from the `ADC` interrupt handler and 185 from the `TIMERB0` handler (i.e., are asynchronous); 386 (both synchronous and asynchronous functions) are reachable from `main`[7].

   To simulate the presence of interrupts, `tos2cprover` needs to instrument the program with explicit calls to the handlers of the expected hardware interrupts, e.g., `sig_ADC_VECTOR()`, with each call guarded by a check of the GIE bit, and each call made atomic by disabling and enabling interrupts[8]. A listing of an IRQ instrumentation is given in Table 3.

```
if (int_enabled()) {
     disable_int();        /* _R2 &= ~0x0008;    */
     sig_ADC_VECTOR();
     enable_int();         /* _R2 |= 0x0008;     */
}
```

Table 3: The ADC IRQ instrumentation.

A correct, yet naive, approach is to instrument the program by refactoring it to use threads, and running the IRQ instrumentation (listed in Table 3) as threads alongside a `main` thread, as in Fig. 7(a). An equivalent *sequential* alternative is to add an instrumentation as every second statement in all non-atomic `main`-reachable code. Since each instrumented IRQ call amounts, at model-checking time, to the duplication of all code rooted in the call, we employ two automated minimization procedures to reduce the number of these sequential instrumentations, as described in what follows.

---

[7] As a note, the style of nesC wiring has as consequence the fact that a number of functions generated by nescc consist of a simple one-line function call.

[8] This is a somewhat artificial limitation, as TinyOS events may *nest*, instead of running to completion; this limitation can be lifted with a finer implementation of TinyOS's scheduling priorities.
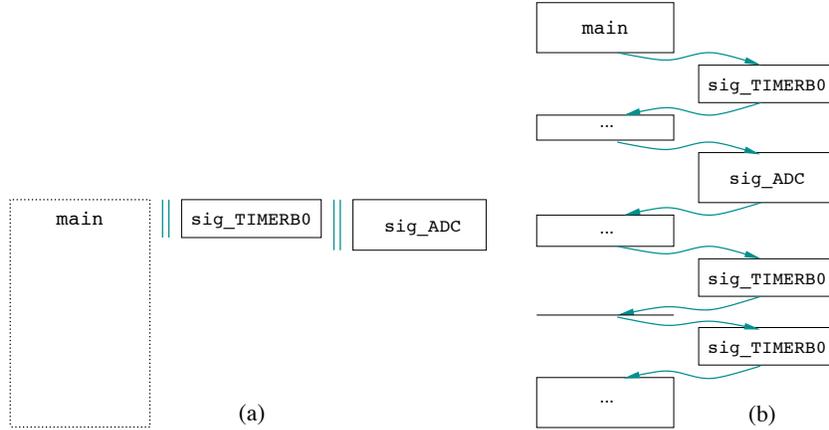
Figure 7: (a) Naive refactoring of the application to emulate hardware interrupts, by running IRQ handlers concurrently with the main program thread. Solid lines denote atomic code blocks; TinyOS events interrupt synchronous code and run to completion. (b) Efficient hardware emulation by partial-order reduction, resulting in a sequential program of reduced state space; all code blocks are now de facto atomic. After a further reduction step, this program is passed to CBMC.

4.2. Decreasing the state space with a partial-order reduction technique

The first state-space minimization procedure is a *partial order reduction* (POR) technique (or *model checking using representatives*) (Clarke et al., 2000), a general method to reduce the state space of a concurrent program to be model checked. Applied to the original embedded C code, it calculates a smaller C program for CBMC to verify; the technique reduces the number of interleavings between threads of behaviour by exploiting the fact that a number of different interleavings are equivalent and indistinguishable to the model checking algorithm, and thus it suffices to check a single interleaving representative. An image of the transformation of the program as a result of the POR technique is given in Fig. 7(b).

To achieve this, we take the concurrent C program refactoring as in Fig. 7(a) and give it a standard formalization, over which we then apply a specialized POR algorithm, as follows.

System formalization. We define by *state* $s \in S$ a valuation of all program variables before or after a C statement[9] or an explicitly atomic section. Then, we define the *transition* set T to contain state tuples, $T \subseteq S \times S$; we write $\alpha \in T$ for a single transition; a *path* is a sequence of transitions. Intuitively, transitions model C statements (including assertions), `atomic` sections, and the program's control flow: when executed from a state s, a transition α leads the program into a new state s'. As an example of this formalization, the simplified, inlined fragment of *Sense* from Fig. 8 (refactored with a concurrent ADC interrupt handler as in Fig. 7(a)) is formalized in Fig. 9 (left).

---

[9] For `tos2cprover`, the atomicity grain of the C program is at the level of C statement: a colon-terminating statement in the inlined C program is considered atomic; the statement may correspond to more than one basic assignment.

```
                    main                                              sig_ADC
       /* global variables */                            /* function SchedulerBasicP_postTask */
       uint8_t next[8U], head, tail, state;              next[tail] = AdcStreamP_readDone;        :β
       unsigned short AdcStreamP_readDone = 5U,
       NO_TASK = 0xFF, RES_IDLE, _ADC12IV, _ADC12IFG;

α₀:    assert (state == RES_IDLE);


       /* function SchedulerBasicP_taskLoop */

       __nesc_atomic_t n = __nesc_atomic_start();
         if (head != NO_TASK)
         {
           head = next[head];
α₁:        next[head] = NO_TASK;
           assert(head != NO_TASK);
         }
       __nesc_atomic_end(n);


α₂:    _ADC12IV = 0;
α₃:    _ADC12IFG = 0;
```

Figure 8: Fragment of a sensor application pre-POR, with a concurrent ADC IRQ. Transition labels are noted on the side. The fragment is extracted from code originally in the `SchedulerBasicP` nesC system component
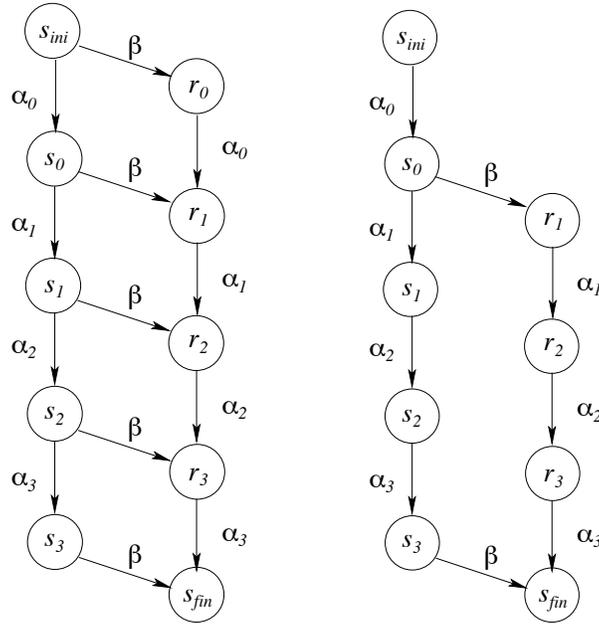

Figure 9: (left) Formalization in terms of states and transition set of the concurrent program in Fig. 8; (right) A sound reduction of the program by our method

A transition α from T is called *enabled* in state s if $\exists s' \in S.s \rightarrow s'$, with the set of transitions enabled in s denoted by *enabled* (s). E.g., *enabled* $(s_0) = \{\alpha_1, \beta\}$ in Fig. 9 (left). Then, the Kripke structure equivalent to this concurrent program is $M = (S, T, \{s_{ini}\})$, with s_{ini} the unique initial state of the program. This we call the *full-state graph*, as T models all possible inter-leavings between the program's threads.

As usual, our POR technique aims at constructing from the full-state transition relation T a second, smaller such relation by selecting for each state $s \in T$ only a subset *ample*(s) of its enabled transitions,

$$ample(s) \subseteq enabled(s)$$

This reduced set of transitions must satisfy a soundness condition: when *ample*(*s*) replaces *enabled*(*s*), the soundness of the model checking algorithm must be preserved.

Calculating the ample set. The basis for soundly calculating *ample*(*s*) relies on an *independence* relation between transitions (Clarke et al., 2000). This independence relation $I \subseteq T \times T$ is a symmetric relation satisfying, for any $s \in S$ and for any $\alpha, \beta \in I$, two conditions:

Enabledness The two transitions may execute in either order from any state *s*. I.e., if $\alpha, \beta \in enabled(s)$, then $\alpha \in enabled(\beta(s))$.

Commutativity Executing either of the two possible transition sequences starting in *any* state *s* leads to the same end state. I.e, if $\alpha, \beta \in enabled(s)$, then $\beta(\alpha(s)) = \beta(\alpha(s))$.

The *dependency* relation *D* is the complement of *I*; two transitions which are not independent are then dependent.

With these definitions, the sound reduction from *enabled*(*s*) to *ample*(*s*) must satisfy the constraints upon *ample*(*s*) (based on (Clarke et al., 2000)) listed in Table 4.

C_0	*ample*(*s*) is empty iff *enabled*(*s*) is empty.
C_1	Along every path of the full-state graph starting in *s*, a transition dependent on a transition α from *ample*(*s*) must be preceded by α. This effectively allows the deferring of transitions.

Table 4: Constraints upon reducing *enabled*(*s*) to *ample*(*s*) for a sound partial-order reduction. As program cycles are subsequently bound and unwound, a constraint upon cycles is not necessary.

We then apply the constraints from Table 4 to the reduction of the sample program in Fig. 9 (left). To start with, we note, from the listing in Fig. 8 by applying the definition of independence above, that α_0, α_2 and α_3 are independent from β; e.g., executing either $\alpha_0\beta$ or $\beta\alpha_0$ from any initial variable valuation leads to the same end values for variables `state`, `next`, etc. Intuitively, the independent pairs of transitions are those which cannot cause data races. α_1 and β are dependent, with a data race over `next`.

Take initial program state s_{ini} with $enabled(s_{ini}) = \{\alpha_0, \beta\}$. By C_0, $ample(s_{ini})$ cannot be empty; it must have at least one transition. Eliminating β (and thus, the entire branch through r_0) from $ample(s_{ini})$ is legal by C_1, with no constraints imposed on the full-state graph. Similar reasoning can be applied for s_1 and s_2. Any reduction of states r_{1-3} is illegal by C_0.

In s_0, with the dependence relation between α_1 and β discovered above, C_1 gives the constraint that β could only be eliminated from $ample(s_0)$ if, in the full-state graph starting in s_0, any occurrence of α_1 is preceded by a β. Since this is not the case for the α_1 enabled in s_0, β must remain in $ample(s_0)$.

Fig. 9 (right) gives the resulting, sound *ample* sets for all program states. This demonstrates that only two IRQ instrumentations need to be considered when verifying this program (with the second not needed in practice: since it becomes the last statement in a sequential program, it cannot influence its verification). Thus, the body of the concurrent program in Fig. 8 can soundly be rewritten as Fig. 10.

```
assert (state == RES_IDLE);
/* function SchedulerBasicP_postTask */
next[tail] = AdcStreamP_readDone;
/* function SchedulerBasicP_taskLoop */

__nesc_atomic_t n = __nesc_atomic_start();
  if (head != NO_TASK)
  {
    head = next[head];
    next[head] = NO_TASK;
    assert(head != NO_TASK);
  }
__nesc_atomic_end(n);


_ADC12IV = 0;
_ADC12IFG = 0;
```

Figure 10. Sound rewriting of the program in Fig. 8.

More intuitively, `tos2cprover` instruments the `main` program with the following reduced set of IRQ calls:

- A call for, e.g., `sig_ADC_VECTOR()` appears before each statement containing a read of a variable raced between the `ADC` interrupt and `main`. This is sound, but overapproximated: the statement may execute in an atomic context, in which case the call is not reachable.

- A call also appears (i) before the beginning of those atomic sections where a data race may happen, or (ii) in the MCU's interruptible sleep.

This first POR-based minimization step results in, e.g., in the case of *Sense*, a total number of 91 IRQ calls between the two types of IRQs, as opposed to a number two orders of magnitude higher in a naive refactoring (i.e., Fig. 7 (a)).

4.3. Improving POR's overapproximated race criterion by reachability checks

A final analysis step is used to further minimize the instrumentation count described in Section 4. To remove some of the degree of overapproximation in assessing the atomicity of code, `tos2cprover` inputs the instrumented program to CBMC for a preliminary run which checks the *reachability* of each IRQ instrumentation: the verification of a claim of the form `assert(0);` inserted in the body of the interrupt routine will fail when the assertion is reachable.

This step leaves, e.g., *Sense* with a manageable set of 8 reachable IRQ calls. This CBMC run is inexpensive, with verification times per claim in the range of 8 to 78 seconds. This order-of-magnitude reduction is due to `tos2cprover` having thus far overapproximated the race criterion—many of the IRQ instrumentations prove to be unreachable within a bounded check, e.g., in situations when they run at program points when interrupts are disabled, or when the code is explicitly atomic.

5. Assertions, nondeterminism and assumptions. Unwinding bounds for CBMC

This section describes the instrumentation of the program with assertions, assumptions and nondeterminism, and the setting of bounds for program unwinding by CBMC.

5.1. Assertions, nondeterminism and assumptions

Our tool chain verifies, for each inputted program, two types of assertions. For both existing TinyOS code base and any new applications, assertions may be manually inserted by the programmer and are preserved as such in the transformed program source. These *application-based assertions* can be either hardware-aware, e.g.

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

or high-level, (e.g., asserting upon the value of a local variable).

Furthermore, CBMC automatically inserts *memory-violation assertions* guarding both bounds of all array accesses, null-pointer dereferences, and other exceptions such as arithmetic division by zero. E.g., as function `SchedulerBasicP_pushTask(uint8_t id)` writes upon the task queue:

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

with `id` unsigned, CBMC will generate the upper-bound assertion:

```
Claim SchedulerBasicP_pushTask.1:
   array 'SchedulerBasicP_m_next' upper bound
   (unsigned int)SchedulerBasicP_m_tail < 8
```

For *Sense*, 132 memory-violation assertions are thus generated[10]. Advantageously, this generation is completely automatic, with CBMC analysing an array's declaration to find the index bounds; SafeTinyOS (Cooprider et al., 2007), for example, has programmers explicitly type-annotate arrays with access bounds instead.

Finally, a decision needs to be taken with regard to the contents of those registers and buffers whose values are filled in by the hardware and not the software. For example, reading the current time in a TelosB application takes the form of reading the user timer's count register, `_TBR` (mapped at address `0x0190`), which holds the number of clock periods elapsed since the last timer interrupt, and which is automatically incremented from the hardware at every clock period. In another example, the 8-bit `_U0RXBUF` buffer (mapped at `0x0076`) holds the latest byte received from the network. A similar discussion holds for setting `TOS_NODE_ID`, the variable which holds the node's address, and which is programmed only at deployment time.

Clearly, the actual values in such registers drive the program's further behaviour. We set their values in either of two ways:

- The register involved is assigned a nondeterministic (i.e., *any*) value, and the verification procedure explores all the ensuing possibilities.

- The register is *assumed* to have a particular value, or to have any value within a small, specified set.

5.2. Unwinding bounds for CBMC

As a final step in our tool chain, the transformed program annotated with assertions is passed to CBMC (configured for 16-bit words), and each claim is verified one at a time, for scalability. The runs need to have an unwinding depth specified; this can be either identical for all loops and recursions, or— ideally— selectively refined for each. Some of the loops are obviously of fixed iterations; e.g., out of only 16 interesting loops in *Sense*, the loop:

---

[10] The names of functions, variables and assertion identifiers in all our code examples are those generated by nescc: the original nesC function or variable name is preceded by a list of nesC component names, which helps in recovering the C code's counterpart in the original nesC code.

```
  do {
    *resultBuffer++ =
      Msp430Adc12ImplP_HplAdc12_getMem(i);
  }
  while(++i < length);
```

is always bounded at 16 iterations (the size of the ADC12 conversion memory)[11]. Other loops, on the other hand, are clearly unbounded, such as the main OS scheduler loop in function `SchedulerBasicP_Scheduler_taskLoop`. For our benchmarks, in the following, we make a visual inspection of the program's loops (as reported by CBMC), determine bounds for the loops which are clearly bounded, and experiment with unwinding depth for the rest. For the purpose of determining the reachability of instrumented IRQ calls (described in Section 4), we set the depth for the unbounded loops to the minimum, 1.

6. Verification and results

For our tests, we settle on the existing applications in the `apps` directory from TinyOS's source tree; we pick applications which wire TelosB components of different functionality, as summarized in Table 5. The coverage criteria is in the amount of different driver code which is pulled in by each application: this ranges from only the timer driver in *Blink*, to timer, ADC and communication drivers for *TestDissemination*.

	Blink	*Sense*	*TestDissemination*
functionality	timer	sensor, timer	CC2420 radio, timer
lines of code, number of loops	3340, 8	7181, 16	13388, 31
memory-violation assertions	35	132	747
expected interrupts	TIMERB0	TIMERB0, ADC	TIMERB0, PORT1, PORT2, UART0RX, UART0TX
reachable functions	total: 248, TIMERB0: 114	total: 520, TIMERB0: 185, ADC: 166	total: 1022, TIMERB0: 364, PORT1: 153, PORT2: 25, UART0RX: 268, UART0TX: 16
potentially raced global variables	TIMERB0: 6	TIMERB0: 7, ADC: 11	TIMERB0: 15, PORT1: 13, PORT2: 0, UART0RX: 19, UART0TX: 0
IRQ instrumentations	initial 21, minimized to 4	initial 92, minimized to	initial 422, minimized to 30

Table 5: TelosB-based test cases

We detail the size and complexity of the test cases in terms of (i) the number of lines of code in the cleanly reformatted program outputted by `tos2cprover`, (ii) the number of unique loops for which CBMC needs to have configured an unwinding depth, (iii) the number and type of expected hardware interrupts, together with qualitative measures of the size of the code duplication incurred during the IRQ instrumentation phase, such as the count of asynchronous functions, and that of ensuing data races. As a side note, the program generated by `nescc` and inputted to `tos2cprover` is not fully optimized; for our test cases, this input program contained code of no end functionality, such as that rooted in the IRQ handlers for the non-user timer, `TIMERA0/1`; `tos2cprover` skips instrumenting the program with such IRQ calls. On another hand, for *TestDissemination* we preserved the code for the `UART0RX/TX` interrupts, and instrumented the program with the respective calls: UART functionality may not be wired through to the top application component, but supports the CC2420 radio.

---

[11] A few loops are bounded, but not worthy of exploring, and are thus commented out. An example is the initial clock calibration, which busy waits for thousands of clock periods.

Finally, Table 5 gives the number of IRQ instrumentations calculated as in Section 4, and the number of automatically generated, memory-violation assertions; most of the assertions are array bounds checks, with a number of null-pointer dereference checks.

In the remainder of this section, we give an overview of our verification runs, and discuss the tool's scalability.

6.1. Out-of-bounds array access, null-pointer dereference, and application-based assertions

We ran our MSP430 test cases through the verification tool chain, having set to *any value* the contents of the TimerB count register `_TBR`, the 16 ADC12 sensor memory buffers `_ADC12MEM[ ]`, and the transmit and receive buffers `_U0TXBUF`/`_U0RXBUF`. Any verification run is parameterized by the following:

- The number of IRQ calls added per iteration of the scheduler main loop. While `tos2cprover` calculates the program points at which an IRQ of a certain type can be called, a verification run may include all, none, or any superset of these calls. This is settled empirically on a per-application basis: one `TIMERB0` interrupt is sufficient to explore the workings of *Blink*, and similarly for `ADC` and *Sense*; for any network communication, on the other hand, an interrupt arrives for any byte received, which induced us to allow more `UART0` transmit or receive interrupts per loop.

- The unwinding bound for the infinitely looping `SchedulerBasicP_Scheduler_taskLoop` function. Given some understanding of the task loop functionality, and the number of IRQ calls per loop, we again settle the number empirically, per application.

- The number of assertions checked in one verification run; this number can be either *one* (and CBMC is configured with the assertion's identifier) or *all*; as checking one assertion at a time scales better, we automated our tool to iterate through all of the program's assertions.

All our verification runs of the memory violations in the test cases from Table 5 came up negative, when allowed two task loops and one IRQ per loop (in the case of *Blink* and *Sense*) and up to eight loops and eight IRQs per loop for *TestDissemination*. We then artificially triggered positive runs in *TestDissemination*, simulating known bugs in TinyOS serial drivers. I.e., we sent a null pointer to a `requestData` call in the `DisseminationEngineImplP` module from TinyOS's network library (the comments are ours):

```
static void DisseminationEngineImplP_sendObject(uint16_t key)
{
  void *object;
  uint8_t objectSize = 0;
  [..]
  // send a zero instead of &objectSize
  object = DisseminationEngineImplP_DisseminationCache_requestData(key, 0);
}
```

Since the final `requestData` method does no sanity check on the pointer it receives:

```
inline static void* DisseminatorP_0_DisseminationCache_requestData(uint8_t *size)
{
  *size = sizeof(DisseminatorP_0_t);
  [..]
}
```

the assertion then generated by CBMC to check the sanity of the pointer:

```
Claim DisseminatorP_0_DisseminationCache_requestData.1:
  line 7503 function DisseminatorP_0_DisseminationCache_requestData
  dereference failure: NULL pointer
  !(SAME-OBJECT(size, NULL))
```

fails.

6.2. Constant-address dereference

A potential, secondary source of errors in embedded software is that of dereferencing constant memory addresses. While null pointers may still be erroneous (as exemplified in Section 6.1), dereferencing constant, low pointers is generally expected from embedded code. To enforce a degree of safety when dereferencing constants is involved, we state that all dereferencing of constants must be limited to constants from those memory-map sections which pertain to peripheral control (I/O locations), and not to other sections.

To this end, in the process of program transformation, `tos2cprover` reports to the programmer the list of encountered memory dereferences, and translates the constant address implicated to its section in the memory map, e.g., for the line:

```
*(volatile uint8_t *)49U ^= 0x01 << 6;
```

we report

```
  -> DEREF at 49/0x31 in the 8-bit Peripheral Module in *(volatile uint8_t *)(49U)
```

and for a fixedly allocated variable:

```
  static volatile uint8_t r __asm ("0x0019"); r|=1 <<1;
```

we report

```
  -> DEREF at 25/0x19 in the 8-bit Peripheral Module with fixed-address variable r
```

In some cases (particularly for dereferences of address `0x0`), an inspection of this report is advisable to sort any null pointers from legitimate peripheral access. A similar approach is taken by SafeTinyOS (Cooprider et al., 2007), which has programmers explicitly mark legal dereferences of constants with a *trusted type*, and thus make null-pointer dereferences visible.

6.3. Cost of verification

The cost of showing that a TinyOS application is safe lies partially in (i) inspecting the program's loops to settle on an unwinding bound for each, (ii) inspecting the list of expected hardware interrupts to decide on the number of IRQ instrumentations necessary, and (iii) inspecting the report on dereferencing constant addresses. Mostly, however, the cost lies with the verification time: the time it takes the model checker to unwind the program (i.e., the *program unwinding time*), generate and simplify its boolean formula, and have this verified by the SAT solver (i.e., the *decision procedure runtime*).

Fig. 10 exemplifies the verification times for a representative subset of memory-violation assertions from *Sense*. The *x* axis is labeled with identifiers of assertions: e.g., `SchedulerBasicP_pushTask.1` is the first assertion within the body of function `SchedulerBasicP_pushTask` (i.e., in the original nesC code, function `pushTask` from component `SchedulerBasicP`). When more than two assertions are generated for a single function, we note that verification times are similar for all these assertions, and only depict the first and the last. Two verification runs are given for each assertion (each run in terms of both program unwinding time and

of decision procedure runtime); both runs are configured with one IRQ call per OS scheduler main loop; the first run unwinds the loop once, and the second twice.

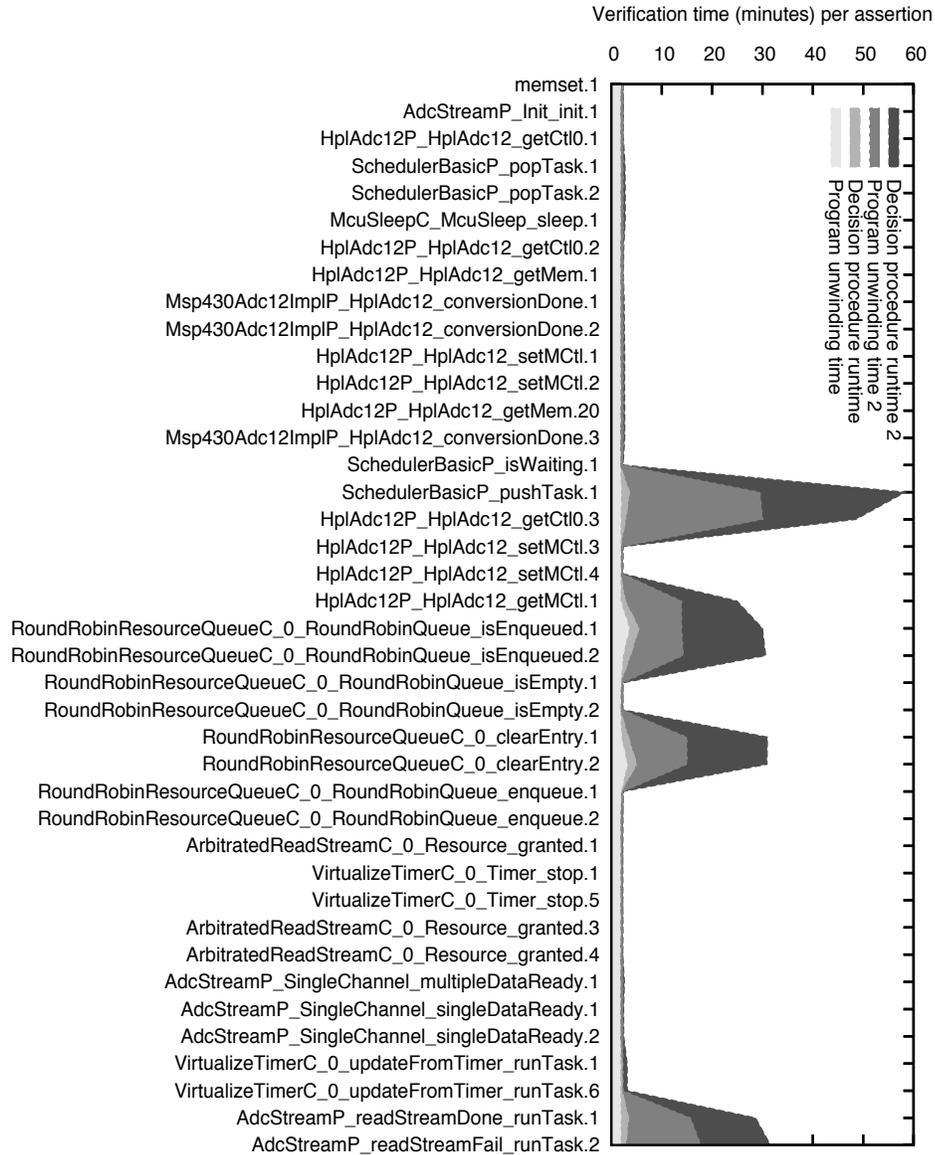Figure 11: Verification times for selected memory-violation assertions in *Sense*

The CBMC runs are configured for 16-bit words, without making use of CBMC's unwinding assertions feature, and with unwinding bounds set per loop (as described in Section 5.2, one higher than the number of loop iterations, e.g., written as a list of `loop number:loop bound`, e.g.:

```
--unwindset 0:9,1:2,2:2,3:2,4:9,6:1,7:17,8:17,9:9,[..]
```

To give an idea of the complexity of the boolean program formula, for, e.g., the verification of claim `SchedulerBasicP_popTask.1` (an assertion over array `SchedulerBasicP_m_next`'s upper bound), the program formula has 80786 assignments, and a set of 18 independent verification conditions (i.e. logical formulae) are generated by CBMC from the C program with annotated assertions. Table 6 gives a fragment of such a verification condition for the function where the above claim resides; on the left is the original C code for the function; on the right is the relevant succession of assignments of bit vectors in CBMC's program translation into verification conditions. Both listings end with giving the final claim, in C and boolean form, respectively.

```
__inline static uint8_t SchedulerBasicP_popTask(void)
{
  if(SchedulerBasicP_m_head != SchedulerBasicP_NO_TASK)      \guard#6 == !(SchedulerBasicP_m_head#8 != 255)
  {
    uint8_t id = SchedulerBasicP_m_head;                      id@2#1 == SchedulerBasicP_m_head#8
    SchedulerBasicP_m_head =                                  SchedulerBasicP_m_head#9 ==
      SchedulerBasicP_m_next[SchedulerBasicP_m_head];          SchedulerBasicP_m_next#12[SchedulerBasicP_m_head#8]

    [..]                                                     [..]

    SchedulerBasicP_m_next[id] = SchedulerBasicP_NO_TASK;    SchedulerBasicP_m_next#13 ==
                                                              (SchedulerBasicP_m_next#12 WITH [id@2#1:=255])

    return id;                                               return_value_SchedulerBasicP_popTask$2@2#1 == id@2#1
  }                                                          return_value_SchedulerBasicP_popTask$2@2#3 == 255
  else                                                       return_value_SchedulerBasicP_popTask$2@2#4 ==
    return SchedulerBasicP_NO_TASK;                           (!\guard#6 ?
                                                              return_value_SchedulerBasicP_popTask$2@2#1
}                                                            : return_value_SchedulerBasicP_popTask$2@2#3)
[..]                                                        [..]
```

```
Claim SchedulerBasicP_popTask.1:                   !\guard#4 && !\guard#8 && !\guard#10 =>
  line 5859 function SchedulerBasicP_popTask         (unsigned int)SchedulerBasicP_m_head#11 < 8
  array 'SchedulerBasicP_m_next' upper bound
  (unsigned int)SchedulerBasicP_m_head < 8
```

Table 6: Fragment of verification condition for *Sense*

When solving the overall program formula after bit blasting, the conjunctive normal form is a 357610-variable, 1058725-clause formula.

We note that most assertions are verified in a speedy manner, using close to zero time in the decision procedure, and a constant time for program unwinding. There are, however, notable exceptions for which the verification time explodes—we used these time-consuming runs to bound CBMC's unwinding depth: for, e.g., *TestDissemination*, some five-scheduler-loops verification runs took up to 55 minutes. Such costly verification runs are generally expected for OS-wide networked applications, due to the large state space added to the program by the network communication drivers.

6.4. Discussion

Support for other platforms. The tool chain is generally independent of platform support. This support is introduced through platform files, included at `tos2cprover` runs, and is essential for the source-to-source transformation described in Table 2, Section 3.2.

The platform files are:

- A header file defining new variables of appropriate types for registers and peripheral ports mapped at fixed numerical addresses, e.g.:

```
unsigned short _R2;        /* R2/SR/CG1 Status Register    */
```

```
unsigned short _R4;      /* General Purpose Register   */
unsigned char  _IE1;     /* @0x0000 Interrupt Enable 1 */
unsigned char  _P5OUT;   /* @0x0031 Port 5 Output      */
unsigned short _TBR;     /* @0x0190 Timer_B register   */
```

- A mapping file, i.e. a dictionary[12] mapping fixed numerical peripheral addresses to the new variables defined above; this is automatically extractable from the header file above:

```
MSP430_map = {
  "0x0000":'_IE1',
  "0x0190":'_TBR',
  "0x0031":'_P5OUT',
  [..]
}
```

- The latter mapping file also describes the new semantics for e.g. `__asm("eint");` in terms of the new variable `_R2` (enabling interrupts now amounts to setting a particular bit in `_R2`, as shown in Table 2).

Thus, the tool being extended to other platforms is a matter of coming up with these platform-specific files (we chose MSP430 for this prototype tool due to previous experience with the platform). While the information for these files is easily extractable from the platforms' data, other difficulties arise at the verification runs. We overview these in what follows.

Difficulties and shortcomings. At a verification run, two issues related to manual instrumentation of the program become serious and limit the usability of the tool for large TinyOS programs. The first is in the need for manually giving a (whenever possible, *sound*) bound to every loop in the program: (i) with every new nesC component linked by the application to be verified, new loops need to be manually investigated for bounds, and (ii) even when the same program loops show up in different applications, the loops may need to be allowed different upper bounds to be sound. A simple example of the latter case is the basic memory-copying function, `memset`: it is called to initialize a task queue of 8 in *Sense*, one of 16 in *TestDissemination*, and one of 2 in *Blink*.

   Other loops may remain identical among applications, yet are not trivial to bound soundly. A simple example is the busywaiting for the clock initialization: `for(calib=0, step=0x800; step!=0; step>>=1) {[..]}`. If this loop were bound to less than its true upper iteration limit, the program following the loop would be unreachable by the verification procedure. Thus, in such cases one needs to allow large loop bounds, unless the loop is not relevant to the subsequent behavior of the software and may be removed; the entire clock calibration procedure is such an example, and we comment it out for the purpose of verification.

   Manually investigating the tens of loop bounds brought up by other large TinyOS applications is time consuming; in many cases this could be automated in a program analysis step. In regard to covering larger test cases, the difficulty of generating loop bounds becomes a serious usability issue for the tool.

   The second issue is platform-specific: for a verification run, it is natural that the values of ADC and UART memories are left nondeterministic, to cover all environmental input; however, all other hardware registers and ports must also be initialized as part of the hardware emulation. Recovering appropriate values in this latter case, while keeping nondeterminism to a minimum, is not trivial. This is strictly an issue of knowledge of the platform workings; in porting the tool to different platforms and in testing on large applications, this is another bottleneck.

---

[12] The current implementation of `tos2cprover` is in Python.

7. Related Work

7.1. Runtime safety

Most existing solutions to detect software errors in sensor operating systems act at *runtime*, are (unlike our method) strictly OS-dependent and platform-independent, and are intended to be used for deployed applications: the code is instrumented such that a statement which is semantically *unsafe* under its current execution context is detected before it is executed, and a certain diagnosis or recovery measure is taken (which usually consists in reporting the error and rebooting, as summarized in Table 7). While a necessary solution to ensure safe execution in all execution contexts, runtime error detection for deployments is potentially followed by the expensive redeployment of software, and could instead be preceded by compile-time means of error detection to save on redeployment efforts.

Scheme (OS)	Scope of errors	Measure
Safe TinyOS, *Neutron* (TinyOS)	out-of-bounds array access, invalid-pointer dereference, integer-to-pointer cast	report error location, reboot; *Neutron*: with state preservation
Interface contracts (TinyOS)	pre-/postconditions for nesC interface use	LED-signal error location
NodeMD (MantisOS)	stack overflow, deadlock, livelock, application assertions	report execution trace, remote queries

Table 7. Runtime diagnosis and recovery solutions to software errors

Safe TinyOS (Cooprider et al., 2007) detects memory and type violations in deployed TinyOS code, and transfers control to a fault handler, which either reboots or powers down after sending a concise failure report to its base station. The failure report identifies the error type and its source code location (but does not give an execution trace clarifying the context which caused the error); the failure report is used to debug the code post-deployment. While the method allows the *safe execution* of existing TinyOS code, with little programmer effort and runtime overhead, debugging every error encountered involves the software's redeployment.

A drawback of Safe TinyOS's Deputy code instrumenter is the fact that, unlike our automatic instrumentation with assertions, SafeTinyOS programmers must explicitly type-annotate e.g., a `void *payload` array with access bounds `COUNT(len)`. The resulting program is first translated into C by the nesC compiler, and then into a program instrumented with, e.g., calls to a failure routine `if(i >= len) deputy_fail()`. Statements accessing a fixed address such as `0x0031` also need manual *trusted cast* annotations, `TC()`. Like SafeTinyOS though (and unlike earlier SafeTinyOS versions), we do not change the C data representation in the verification process.

Neutron (Chen et al., 2009) adds a welcome update to Safe TinyOS, for applications coded in TinyOS's TOSThreads API: instead of rebooting the node as a corrective measure to a memory violation, Neutron groups the application's threads into recovery groups, and selectively restarts the threads in certain recovery units. Furthermore, it allows the preservation of the values held in "precious" memory locations between thread restarts.

The *interface contracts* (Archer et al., 2007) write specification-like type annotations which define the "correct" use of TinyOS 1.x nesC interfaces, just as Safe TinyOS annotates memory for safe use. In cases when the semantics of the interface dictates it being stateful, the contract is expressed in terms of the state of the interface: for the interface command `Timer.start()`, the contract states as precondition the fact that the timer's state must be `IDLE`, and as a postcondition that the state of the timer changes (i.e., to `ONE SHOT`) if the command's return value is `SUCCESS`. In other cases, the interface is stateless and the pre-

and postconditions are imposed upon the command's arguments, which can themselves become stateful. This detects incorrect ordering of interface commands at runtime, e.g., a `SendMsg.send()` with a message buffer for which no `SendMsg.sendDone()` event was received to complete a previous send. On a related note, similar, stateful interface contracts are set over nesC commands and events in (Menrad et al., 2009); noting that state transitions, as programmed for interface contracts, can become lengthy, it devises an equivalent, more readable statechart notation, with the view towards a future automated (static) verification for all TinyOS interfaces.

NodeMD (Krunic et al., 2007) detects runtime errors in MantisOS multithreaded, synchronous code. AVR-based applications are instrumented to check the stack pointer `SP` for overflows at function-call time, and for the violation of application-specific assertions. The checks for deadlock and livelock involve a manually added timer to verify that each thread goes through its duty cycle.

7.2. Hybrid approaches: runtime safety by emulation

An important hybrid approach between runtime and static error detection consists of combining runtime safety systems (such as Safe TinyOS) with cycle-accurate hardware *emulators* such as Avrora (Titzer et al., 2005), MSPsim (Eriksson et al., 2007), and WSim (Fraboulet et al., 2007). A hardware emulator precisely simulates a platform in terms of its running assembly instructions; beside analyses of power and memory consumption, a precise timing analysis is also possible, given appropriate timing attached to the instructions. An overview of the current hardware support of emulators for sensor nodes is given in Table 8.

Emulator (OS)	Microcontrollers	Other chipsets
WSim (OSless)	TI MSP430, ATMega128	Chipcon CC1100, CC2420; Maxim serial ID DS2411; LEDs
MSPsim (Contiki OS)	TI MSP430	TelosB peripherals
Avrora (OSless)	Atmel AVR ATMega128	Mica2 peripherals

Table 8. Sensor platform emulators

While this combination does not provide full error traces the way a model-checking technique would, it does report the call stack at the point of the error, the way Safe TinyOS would; thus, real-time safety does not necessarily need to act at deployment-time.

7.3. Verification and simulation

A weak form of static error detection is simulation. While it does not prove any guarantee on the program's behaviour, testing allows for error detection and is particularly realistic in the case of e.g., TOSSIM (Levis et al., 2003), which accurately (yet not cycle-accurately) simulates TinyOS applications from their implementation.

TinyOS's nesC compiler has a basic built-in *data-race* detector, which warns when a global variable is updated from non-`atomic` asynchronous code without having been explicitly tagged with `norace`. While writing `atomic` asynchronous code is good practice for nesC, failure to do so only potentially causes a race, as programmers may have used other syncronization idioms (i.e., guards on variables). A suitable context model for race checking (Henzinger et al., 2004) then contributed an algorithm for the elimination of such false positives.

A degree of compile-time verification is reported by Bucur and Kwiatkowska (2009). While its scope is limited to TOSThreads applications written in C, it avoids some of the costs of system-wide verification by writing models for the interfaces to system calls (in the style of the runtime interface contracts Archer et al. (2007)). Calling `amRadioReceive(&msg, ...)` from the application is modelled so that it preserves its original behaviour: the call returns any of a set of error codes, and the `msg` variable receives

(possibly nondeterministic) data. Then, the method calls the SATABS model checker for multithreaded C programs to verify the programmer's application on its own; the errors verified pertain to interface use, or are application-specific.

SMT solvers are used as backends to ANSI-C model checkers, instead of SAT solvers in (Cordeiro et al., 2009a,b, 2010), to verify bounded instances of ANSI-C programs, such as those generated by BMC frontends. Satisfiability Modulo Theories (SMT) solvers employ decision procedures which check the satisfiability of a quantifier-free formula in a first-order logic. To make SMT-based bounded model checkers applicable to checking realistic C, (Cordeiro et al., 2009b) provides translations from ANSI-C programs to SMT formulas as precisely as bit-accurate SAT-based procedures, and positively compares the performance of their model checker to that of CBMC and a previous SMT-based CMBC. New encodings are provided into existing SMT theories from ANSI-C scalar data types (with accurate arithmetic overflow and underflow), arrays and pointers, structures and unions; the test cases include array-heavy ANSI-C benchmarks such as sorting algorithms and Linux device-controlling applications. (Cordeiro et al., 2010) adds a statespace-reducing technique for the same verification method; this looks at the modifications undergone by the system since its last verification, and submits them to a partly static, partly dynamic "continuous" verification process, guided by a set of test cases for coverage.

Closer to our domain of sensor software, (Cordeiro et al., 2009a) gives a single case study of a verification approach to part of a platform-specific embedded C monitoring software. This approach splits the 3500-LOC monitoring application, written as a set of modules, into platform-dependent and platform-independent modules; like our method, all platform-dependent syntax is translated to standard C. The method then statically verifies platform-dependent modules, one by one, using the co-verification feature of CBMC, together with a Verilog model of the microcontroller and against the standard CProver-inserted assertions to check the sanity of the interaction between software and hardware. A system-wide checking procedure is simulated on a hardware emulator. The method's advantages mostly lie in its ability to make good use of CBMC's hardware co-verification; its disadvantages lie in the amount of manual input needed to decide between the different verification schemes for different software modules, and in its potential lack of generality.

Recent contributions. Other verification or high-coverage validation methods tackle the more difficult problem of static debugging for sensor network protocols.

KleeNet (Sasnauskas et al., 2010) is a recent debugging environment for the high-coverage *testing* of *networked* sensor applications (case studied for Contiki OS), with the aim of discovering bugs that result from node interaction (by writing distributed assertions about the state of the network) and nondeterministic network events (such as loss, duplication and corruption of packets, or node failures). The latter aspects are especially difficult to observe in traditional testing mechanisms, or require manual effort by the developers to generate them.

The base technique for KleeNet's testing is the exploration of program paths in the virtual machine KLEE (Cadar et al., 2008): a distributed program is simulated in standard fashion until data flagged as 'symbolic' is reached (e.g., the contents of a network packet; this may be set to a nondeterministic value); at these points, the execution is branched (in the fashion of explicit-state model checking, with nodes forked on demand) and resumed for each such branch. This technique is then extended by KleeNet with failure models for both nodes and network communication, and assertions are written in distributed fashion. E.g., for a node *A* which just adds node *B* as a `parentID` in its routing table, the assertion:

```
if (parentID != NULL) {
   assert(NODE(parentID, ''isChild'', myID)); }
```

states that *B* should also have *A* registered as a child. `parentID` and `myID` are variables local to *A*, while `isChild` is a function called on *B*.

KleeNet itself is platform-independent, but each sensor network OS requires a frontend to KLEE. It was able to discover four bugs in Contiki's TCP/IP stack, one of which deadlocked a network node.

On a similar note to KleeNet, T-Check (Li and Regehr, 2010) uses *random walks* and execution-driven, depth-bounded *explicit-state model checking*. It builds on the TOSSIM (Levis et al., 2003) simulator for TinyOS, and inherits its emulation of hardware at the TinyOS interface, rather than at the hardware register level; this precludes safety specifications related to, e.g., register contents, timing or interrupt preemption, but gains scalability. T-Check does, however, report that it "found TOSSIM to be too high-level to support effective bug-finding", which led to its "extending the ADC, serial, and SPI subsystems to model more low-level behavior", i.e. modelling the relevant interrupts, as in our work. Network and node nondeterminism is introduced, together with a TinyOS-specific nondeterminism related to event ordering. Their model checking is a stateless, depth-bounded, depth-first search with a partial order reduction based on the static presumption that a pair of transitions on different sensor nodes is independent unless the events are a matched send/receive pair. Both safety and liveness properties are checked against, the latter heuristically, by looking for sufficiently long program traces violating the property; a number of bugs are found in the TinyOS serial driver and some tree protocols.

Anquiro (Mottola et al., 2010) is a model-checking based verification tool for Contiki applications; as a plus over our `tos2cprover`-based tool, it allows for a software engineering solution to slice the application code up to a desired level, e.g., either including code interfacing to the hardware as in our method, or remodelling network communication, similar to (Bucur and Kwiatkowska, 2009). LTL specifications are inputted to the Bogor model checker, and Anquiro is able to find a network configuration in which a dissemination protocol does not reach all nodes.

Other approaches. Insense (Sharma et al., 2009) designs a novel language for programming wireless sensor networks applications, which then compiles into Contiki C source code. With the aim of simplifying the complexity of both programming and the verification of the resulted programs, the language hides from the programmer all language constructs regarding concurrency (e.g., for the various existing WSN-programming API: processes, threads, asynchronous events) and thread synchronization. An Insense application is instead programmed as a set of components, sharing no states and communicating through typed, synchronous channels. Selected hardware is modelled (similarly to our (Bucur and Kwiatkowska, 2009)) as Insense components and matching channels. For the purpose of verification, the components and channels are translated into Promela and model checked against LTL properties by SPIN (Holzmann, 2003); their cases are limited to the verification of properties of single channel operations in their translation from Insense to SPIN, such as "A *send* operation does not return until data has been written to a receiver's buffer".

FSMGen (Kothari et al., 2008) takes another approach to error detection in TinyOS programs: it statically analyzes the program and derives automatically a finite-state machine to describe the high-level application logic, thus aiding the programmer's understanding of the application code. The method is applied over the demo applications available with TinyOS.

8. Conclusions

We have contributed a tool[13] and novel approach towards the static software verification of embedded C sensor applications. Operating-system-wide sensor programs with an added explicit emulation of hardware interrupts are automatically given precise standard C models for the MCU's direct memory access, and are then minimized in state space with the aid of a partial-order reduction technique. Safety specifications written as assertions and bounds for the program loops are inputted together with the

---

[13] Documentation and repository at `http://www.daimi.au.dk/~doina/svtos.php`.

program into CBMC, which is then able to verify the program and report error traces. While no new bugs have been found, we were able to reproduce known bugs in a TinyOS driver, and we list as our main achievement the efficient, fully automatic method and tool bridging embedded C compilers for sensor platforms with the state of the art tools in C software verification.

References

Archer, W., Levis, P., Regehr, J., 2007. Interface contracts for TinyOS. In: Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN). ACM, pp. 158–165.

Atmel, 2008. Atmel 8051 microcontrollers hardware manual. www.atmel.com.

Bucur, D., Kwiatkowska, M., 2009. Bug-free sensors: The automatic verification of context-aware TinyOS applications. In: Proceedings of the European Conference on Ambient Intelligence (AmI). Vol. LNCS 5859. Springer Verlag, pp. 101–105.

Bucur, D., Kwiatkowska, M., 2010. Software verification for TinyOS. In: Information Processing in Sensor Networks (IPSN). ACM, pp. 400–401.

Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. OSDI 2008: 209-224.

Chen, Y., Gnawali, O., Kazandjieva, M., Levis, P., Regehr, J., 2009. Surviving sensor network software faults. In: Proceedings of the Symposium on Operating System Principles (SOSP), pp. 235–246. ACM.

Clarke, E. M., Kroening, D., Lerda, F., 2004. A tool for checking ANSI-C programs. In: TACAS. Vol. 2988 of LNCS. Springer, pp. 168–176.

Clarke, E. M., Biere, A., Raimi, R., Zhu, Y., 2001. Bounded model checking using satisfiability solving. Formal Methods in System Design 19 (1), pp. 7–34. Kluwer Academic Publishers.

Clarke, E. M., Grumberg, O., Peled, D. A., 2000. Model Checking. MIT Press.

Cooprider, N., Archer, W., Eide, E., Gay, D., Regehr, J., 2007. Efficient memory safety for TinyOS. In: Proceedings of the conference on Embedded Networked Sensor Systems (SenSys). ACM, pp. 205–218.

Cordeiro, L., Fischer, B., Chen, H., Marques-Silva, J., 2009a. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In: Proceedings of the International Conference on Embedded Software and Systems, pp. 396–403. IEEE Computer Society.

Cordeiro, L., Fischer, B., Marques-Silva, J., 2009b. SMT-based bounded model checking for embedded ANSI-C software. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 137–148.

Cordeiro, L., Fischer, B., Marques-Silva, J., 2010. Continuous verification of large embedded software using SMT-based bounded model checking. In: Proceedings of IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS), pp. 160–169.

Corp., A., 2009. 8-bit AVR microcontroller with 128K bytes in-system programmable Flash. www.atmel.com, doc2467.pdf.

Eriksson, J., Dunkels, A., Finne, N., Osterlind, F., Voigt, T., 2007. MSPsim – an extensible simulator for MSP430-equipped sensor boards. In: Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session.

Fraboulet, A., Chelius, G., Fleury, E., 2007. Worldsens: development and prototyping tools for application specific wireless sensors networks. In: Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN). ACM, pp. 176–185.

Gay, D., Levis, P., Culler, D., 2005. Software design patterns for TinyOS. In: Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES). ACM, pp. 40–49.

Gay, D., Levis, P., von Behren, R., 2003. The nesC language: A holistic approach to networked embedded systems. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM, pp. 1–11.

Henzinger, T. A., Jhala, R., Majumdar, R., 2004. Race checking by context inference. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI). ACM Press, pp. 1–13.

Holzmann, G. J., 2003. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley.

Klues, K., Liang, C.-J., Paek, J., Musăloiu, R., Govindan, R., Terzis, A., Levis, P., 2009. TOSThreads: Safe and Non-Invasive Preemption in TinyOS. In: Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys). ACM, pp. 127-140.

Kothari, N., Millstein, T., Govindan, R., 2008. Deriving state machines from TinyOS programs using symbolic execution. In: Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN). IEEE, pp. 271–282.

Kroening, D., Strichman, O., 2008. Decision Procedures: An Algorithmic Point of View. Springer.

Krunic, V., Trumpler, E., Han, R., 2007. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In: Proceedings of the International Conference on Mobile Systems, Applications and Services (MobiSys). ACM, pp. 43–56.

Levis, P., Gay, D., Handziski, V., Hauer, J.-H., Greenstein, B., Turon, M., Hui, J., Klues, K., Sharp, C., Szewczyk, R., Polastre, J., Buonadonna, P., Nachman, L., Tolle, G., Culler, D., Wolisz, A., 2005. T2: A second generation OS for embedded sensor networks. Tech. Rep. TKN-05-007, Technische Universität Berlin.

Levis, P., Lee, N., Welsh, M., Culler, D. E., 2003. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In: Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys). pp. 126–137.

Li, P., Regehr, J., 2010. T-Check: bug finding for sensor networks. In: Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN). ACM, pp. 174–185.

Menrad, V., Garcia, M., Schupp, S., 2009. Improving TinyOS developer productivity with statecharts. In: Proceedings of the Workshop on Self-Organising Wireless Sensor and Communication Networks.

Moteiv Corporation, 2004. 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.

Mottola, L., Voigt, T., Osterlind, F., Eriksson, J., Baresi, L., Ghezzi, C., 2010. Anquiro: Enabling efficient static verification of sensor network software. In: Proceedings of Workshop on Software Engineering for Sensor Network Applications (SESENA) ICSE(2).

Sasnauskas, R., Landsiedel, O., Alizai, M. H., Weise, C., Kowalewski, S., Wehrle, K., 2010. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In: Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN), pp. 186–196.

Sharma, O., Lewis, J., Miller, A., Dearle, A., Balasubramaniam, D., Morrison, R., Sventek, J., 2009. Towards verifying correctness of wireless sensor network applications using Insense and SPIN. In: Model Checking Software, Vol. 5578 of LNCS, Springer, pp. 223–240.

Sorensson, N., Eén, N., 2005. MiniSat — A SAT solver with conflict-clause minimization. In: Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT). Vol. 3569 of LNCS. Springer.

Texas Instruments, 2006. MSP430x1xx family — user's guide (Rev. F). www.ti.com.

Titzer, B. L., Lee, D. K., Palsberg, J., 2005. Avrora: scalable sensor network simulation with precise timing. In: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN), Demo session. IEEE Press, pp. 67–72.

Underwood, S., 2003. Mspgcc—A port of the GNU tools to the Texas Instruments MSP430 microcontrollers. http://mspgcc.sourceforge.net/manual.